

# Содержание

Об авторе	21
Об иллюстрации на обложке	21
<b>Предисловие</b>	<b>22</b>
“Экосистема” этой книги	22
О пятом издании	23
Линейки Python 2.X и Python 3.X	25
Современная история Python 2.X/3.X	25
Раскрытие линеек Python 3.X и Python 2.X	26
Какая версия Python должна использоваться?	27
Предпосылки и усилия	28
Структура этой книги	29
Чем эта книга не является	32
Это не справочник и не руководство по специфическим приложениям	32
Это не краткая история для спешащих людей	33
Изложение последовательно до той степени, до которой позволяет Python	33
Программы в книге	34
Версии Python	34
Платформы	35
Загрузка кода примеров для книги	35
Использование кода, сопровождающего книгу	35
Соглашения, используемые в этой книге	36
Ждем ваших отзывов!	36
Благодарности	37
Предыстория	37
Благодарности Python	37
Личные благодарности	38
<b>Часть I. Начало работы</b>	<b>39</b>
<b>ГЛАВА 1. Python в вопросах и ответах</b>	<b>40</b>
Почему люди используют Python?	40
Качество программного обеспечения	42
Продуктивность труда разработчиков	42
Является ли Python “языком написания сценариев”?	43
Хорошо, но в чем недостаток?	44
Кто использует Python в наши дни?	46
Что можно делать с помощью Python?	48
Системное программирование	48
Графические пользовательские интерфейсы	49
Написание сценариев для Интернета	49
Интеграция компонентов	50
Программирование для баз данных	50
Быстрое прототипирование	51
Численное и научное программирование	51

И еще: игры, изображения, глубинный анализ данных, роботы, электронные таблицы Excel...	52
Как Python разрабатывается и поддерживается?	53
Компромиссы, связанные с открытым кодом	53
Каковы технические превосходства Python?	54
Он объектно-ориентированный и функциональный	54
Он бесплатный	55
Он переносимый	55
Он мощный	56
Он смешиваемый	57
Он относительно прост в использовании	57
Он относительно прост в изучении	58
Он назван в честь группы “Монти Пайтон”	58
Как Python соотносится с языком X?	59
Резюме	60
Проверьте свои знания: контрольные вопросы	61
Проверьте свои знания: ответы	61
<b>ГЛАВА 2. Как Python выполняет программы</b>	64
Введение в интерпретатор Python	64
Выполнение программ	66
Точка зрения программиста	66
Точка зрения Python	67
Разновидности модели выполнения	70
Альтернативные реализации Python	70
Инструменты оптимизации выполнения	73
Фиксированные двоичные файлы	75
Будущие возможности?	76
Резюме	76
Проверьте свои знания: контрольные вопросы	77
Проверьте свои знания: ответы	77
<b>ГЛАВА 3. Как пользователь выполняет программы</b>	78
Интерактивная подсказка	78
Запуск интерактивного сеанса	79
Пути поиска в системе	81
Новые возможности для Windows в версии Python 3.3:	
переменная среды PATH и запускающий модуль	81
Где выполнять: каталоги для кода	82
Что не набирать: приглашения к вводу и комментарии	83
Интерактивное выполнение кода	84
Для чего нужна интерактивная подсказка?	85
Замечания по использованию: интерактивная подсказка	87
Командная строка системы и файлы	89
Первый сценарий	90
Запуск файлов в командной строке	91
Варианты использования командной строки	92
Замечания по использованию: командная строка и файлы	93

Исполняемые сценарии в стиле Unix: #!	94
Основы сценариев Unix	94
Трюк с поиском посредством env в Unix	95
Запускающий модуль для Windows в версии Python 3.3: #! приходит в Windows	95
Щелчки на значках файлов	97
Основы щелчков на значках	97
Щелчки на значках в Windows	98
Трюк с использованием функции input в Windows	99
Другие ограничения, связанные со щелчками на значках	101
Импортирование и повторная загрузка модулей	101
Основы импортирования и повторной загрузки	101
Дополнительная история о модулях: атрибуты	103
Замечания по использованию: import и reload	106
Использование exes для выполнения файлов модулей	107
Пользовательский интерфейс IDLE	108
Детали запуска IDLE	109
Базовое использование IDLE	110
Удобные функциональные возможности IDLE	111
Расширенные инструменты IDLE	112
Замечания по использованию: IDLE	112
Другие IDE-среды	114
Другие варианты запуска	116
Встраивание вызовов	116
Фиксированные двоичные исполняемые файлы	117
Варианты запуска из текстовых редакторов	117
Прочие варианты запуска	117
Будущие возможности?	118
Какой вариант должен использоваться?	118
Резюме	120
Проверьте свои знания: контрольные вопросы	120
Проверьте свои знания: ответы	121
Проверьте свои знания: упражнения для части I	122
<b>Часть II. Типы и операции</b>	125
<b>ГЛАВА 4. Введение в типы объектов Python</b>	126
Концептуальная иерархия Python	126
Для чего используются встроенные типы?	127
Основные типы данных Python	128
Числа	130
Строки	131
Операции над последовательностями	132
Неизменяемость	134
Методы, специфичные для типа	135
Получение справки	136
Другие способы написания строк	137
Строки Unicode	138
Сопоставление с образцом	141

Списки	141
Операции над последовательностями	141
Операции, специфичные для типа	142
Контроль границ	142
Вложение	143
Списковые включения	144
Словари	146
Операции над отображениями	146
Снова о вложении	147
Недостающие ключи: проверки <code>if</code>	149
Сортировка ключей: циклы <code>for</code>	150
Итерация и оптимизация	152
Кортежи	153
Для чего используются кортежи?	154
Файлы	154
Файлы с двоичными байтами	155
Файлы с текстом Unicode	156
Другие инструменты, подобные файлам	158
Прочие основные типы	158
Как нарушить гибкость кода	160
Классы, определяемые пользователем	161
Все остальное	162
Резюме	162
Проверьте свои знания: контрольные вопросы	163
Проверьте свои знания: ответы	163
<b>ГЛАВА 5. Числовые типы</b>	165
Основы числовых типов	165
Числовые литералы	166
Встроенные инструменты для обработки объектов чисел	168
Операции выражений Python	168
Числа в действии	173
Переменные и базовые выражения	173
Форматы числового отображения	175
Сравнения: нормальные и сцепленные	176
Деление: классическое, с округлением в меньшую сторону и настоящее	178
Точность целых чисел	182
Комплексные числа	183
Шестнадцатеричная, восьмеричная и двоичная формы записи: литералы и преобразования	183
Побитовые операции	185
Другие встроенные инструменты для обработки чисел	187
Другие числовые типы	189
Десятичные типы	189
Дробный тип	192
Множества	195
Булевские значения	203
Численные расширения	204
Резюме	205

Проверьте свои знания: контрольные вопросы	205
Проверьте свои знания: ответы	205
<b>ГЛАВА 6. Кратко о динамической типизации</b>	207
Случай отсутствия операторов объявления	207
Переменные, объекты и ссылки	208
Типы обитают в объектах, не в переменных	210
Объекты подвергаются сборке мусора	210
Разделяемые ссылки	212
Разделяемые ссылки и изменения на месте	214
Разделяемые ссылки и равенство	215
Динамическая типизация вездесуща	217
Резюме	218
Проверьте свои знания: контрольные вопросы	218
Проверьте свои знания: ответы	218
<b>ГЛАВА 7. Фундаментальные основы строк</b>	219
Вопросы, раскрываемые в главе	219
Unicode: краткая история	220
Основы строк	220
Строковые литералы	222
Строки в одинарных и двойных кавычках являются одинаковыми	223
Управляющие последовательности представляют специальные символы	223
Неформатированные строки подавляют управляющие последовательности	227
Утроенные кавычки представляют многострочные блочные строки	228
Строки в действии	230
Базовые операции	230
Индексация и нарезание	231
Инструменты преобразования строк	235
Изменение строк, часть I	238
Строковые методы	239
Синтаксис вызова методов	239
Методы строк	240
Примеры строковых методов: изменение строк, часть II	241
Примеры строковых методов: разбор текста	243
Другие распространенные строковые методы в действии	244
Функции первоначального модуля <code>string</code> (изъяты из Python 3.X)	245
Выражения форматирования строк	246
Основы выражений форматирования	247
Расширенный синтаксис выражений форматирования	248
Более сложные примеры использования выражений форматирования	249
Выражения форматирования, основанные на словаре	250
Вызовы методов форматирования строк	251
Основы методов форматирования	251
Добавление ключей, атрибутов и смещений	252
Расширенный синтаксис методов форматирования	253
Более сложные примеры использования методов форматирования	254
Сравнение с выражением форматирования <code>%</code>	256
Для чего используется метод <code>format</code> ?	259

Общие категории типов	264
Типы разделяют наборы операций по категориям	264
Изменяемые типы можно модифицировать на месте	265
Резюме	266
Проверьте свои знания: контрольные вопросы	266
Проверьте свои знания: ответы	267
<b>ГЛАВА 8. Списки и словари</b>	268
Списки	268
Списки в действии	270
Базовые списковые операции	271
Итерация по спискам и списковые включения	271
Индексация, нарезание и матрицы	272
Изменение списков на месте	273
Словари	279
Словари в действии	280
Базовые словарные операции	281
Изменение словарей на месте	282
Дополнительные словарные методы	283
Пример: база данных о фильмах	285
Замечания по использованию словарей	287
Другие способы создания словарей	291
Изменения в словарях в Python 3.X и 2.7	293
Резюме	300
Проверьте свои знания: контрольные вопросы	301
Проверьте свои знания: ответы	301
<b>ГЛАВА 9. Кортежи, файлы и все остальное</b>	302
Кортежи	303
Кортежи в действии	304
Для чего используются списки и кортежи?	307
Снова о записях: именованные кортежи	307
Файлы	309
Открытие файлов	310
Использование файлов	311
Файлы в действии	313
Кратко о текстовых и двоичных файлах	314
Хранение объектов Python в файлах: преобразования	315
Хранение собственных объектов Python: модуль pickle	317
Хранение объектов Python в формате JSON	318
Хранение упакованных двоичных данных: модуль struct	320
Диспетчеры контекстов для файлов	321
Другие файловые операции	322
Обзор и сводка по основным типам	323
Гибкость объектов	324
Ссылки или копии	326
Сравнения, равенство и истинность	328
Смысл понятий “истина” и “ложь” в Python	331
Иерархии типов Python	333

Объекты <code>type</code>	333
Прочие типы в Python	335
Затруднения, связанные со встроенными типами	335
Присваивание создает ссылки, а не копии	336
Повторение добавляет один уровень глубины	336
Остерегайтесь циклических структур данных	337
Неизменяемые типы нельзя модифицировать на месте	338
Резюме	338
Проверьте свои знания: контрольные вопросы	339
Проверьте свои знания: ответы	339
Проверьте свои знания: упражнения для части II	340
<b>Часть III. Операторы и синтаксис</b>	<b>343</b>
<b>ГЛАВА 10. Введение в операторы Python</b>	<b>344</b>
Еще раз о концептуальной иерархии Python	344
Операторы Python	345
История о двух <code>if</code>	347
Что Python добавляет	347
Что Python устраняет	348
Для чего используется синтаксис с отступами?	349
Несколько специальных случаев	352
Короткий пример: интерактивные циклы	354
Простой интерактивный пример	354
Выполнение математических действий над пользовательским вводом	356
Обработка ошибок путем проверки ввода	357
Обработка ошибок с помощью оператора <code>try</code>	358
Вложение кода на три уровня в глубину	360
Резюме	361
Проверьте свои знания: контрольные вопросы	361
Проверьте свои знания: ответы	361
<b>ГЛАВА 11. Операторы присваивания, выражений и вывода</b>	<b>362</b>
Операторы присваивания	362
Формы оператора присваивания	363
Присваивание последовательности	364
Расширенная распаковка последовательностей в Python 3.X	367
Групповые присваивания	371
Дополненные присваивания	372
Правила именования переменных	375
Операторы выражений	379
Операторы выражений и изменения на месте	380
Операции вывода	380
Функция <code>print</code> в Python 3.X	381
Оператор <code>print</code> в Python 2.X	384
Перенаправление потока вывода	385
Вывод, нейтральный к версии	389
Резюме	392

Проверьте свои знания: контрольные вопросы	393
Проверьте свои знания: ответы	393
<b>ГЛАВА 12. Проверки if и правила синтаксиса</b>	394
Операторы if	394
Общий формат	394
Элементарные примеры	395
Множественное ветвление	395
Снова о синтаксисе Python	398
Ограничители блоков: правила отступов	399
Ограничители операторов: строки и продолжения	401
Несколько особых случаев	402
Значения истинности и булевские проверки	404
Тернарное выражение if/else	405
Резюме	408
Проверьте свои знания: контрольные вопросы	409
Проверьте свои знания: ответы	409
<b>ГЛАВА 13. Циклы while и for</b>	410
Циклы while	410
Общий формат	410
Примеры	411
Операторы break, continue, pass и конструкция else цикла	412
Общий формат цикла	412
Оператор pass	412
Оператор continue	414
Оператор break	414
Конструкция else цикла	415
Циклы for	417
Общий формат	418
Примеры	418
Методики написания циклов	425
Циклы с подсчетом: range	425
Просмотр последовательностей: while и range или for	426
Тасование последовательностей: range и len	427
Неполный обход: range или срезы	428
Изменение списков: range или включения	429
Параллельные обходы: zip и map	430
Генерация смещений и элементов: enumerate	433
Резюме	436
Проверьте свои знания: контрольные вопросы	437
Проверьте свои знания: ответы	437
<b>ГЛАВА 14. Итерации и включения</b>	438
Итерации: первый взгляд	438
Протокол итерации: итераторы файловых объектов	439
Ручная итерация: iter и next	442
Итерируемые объекты других встроенных типов	445



Списковые включения: первый подробный взгляд	447
Основы списковых включений	448
Использование списковых включений с файлами	449
Расширенный синтаксис списковых включений	451
Другие итерационные контексты	453
Новые итерируемые объекты в Python 3.X	457
Влияние на код Python 2.X: доводы за и против	457
Итерируемый объект range	458
Итерируемые объекты map, zip и filter	459
Итераторы с множеством проходов или с одним проходом	461
Итерируемые словарные представления	462
Другие темы, связанные с итерацией	464
Резюме	464
Проверьте свои знания: контрольные вопросы	465
Проверьте свои знания: ответы	465
<b>ГЛАВА 15. Документация</b>	466
Источники документации Python	466
Комментарии #	467
Функция dir	467
Строки документации: __doc__	469
PyDoc: функция help	472
PyDoc: отчеты в формате HTML	475
За рамками строк документации: Sphinx	483
Стандартный набор руководств	483
Веб-ресурсы	484
Изданные книги	485
Распространенные затруднения при написании кода	485
Резюме	487
Проверьте свои знания: контрольные вопросы	488
Проверьте свои знания: ответы	488
Проверьте свои знания: упражнения для части III	489
<b>Часть IV. Функции и генераторы</b>	491
<b>ГЛАВА 16. Основы функций</b>	492
Для чего используются функции?	493
Написание кода функций	494
Операторы def	496
Оператор def исполняется во время выполнения	496
Первый пример: определения и вызовы	497
Определение	497
Вызов	497
Полиморфизм в Python	498
Второй пример: пересечение последовательностей	499
Определение	500
Вызов	500
Еще раз о полиморфизме	501
Локальные переменные	502

Резюме	502
Проверьте свои знания: контрольные вопросы	503
Проверьте свои знания: ответы	503
<b>ГЛАВА 17. Области видимости</b>	<b>504</b>
Основы областей видимости в Python	504
Детали, касающиеся областей видимости	505
Распознавание имен: правило LEGB	507
Пример области видимости	510
Встроенная область видимости	511
Оператор <code>global</code>	514
Проектирование программы: минимизируйте количество глобальных переменных	515
Проектирование программы: минимизируйте количество межфайловых изменений	516
Другие способы доступа к глобальным переменным	518
Области видимости и вложенные функции	519
Детали вложенных областей видимости	519
Примеры вложенных областей видимости	519
Фабричные функции: замыкания	520
Сохранение состояния из охватывающей области видимости с помощью стандартных значений	523
Оператор <code>nonlocal</code> в Python 3.X	527
Основы оператора <code>nonlocal</code>	527
Оператор <code>nonlocal</code> в действии	529
Для чего используются оператор <code>nonlocal</code> ? Варианты сохранения состояния	531
Состояние с помощью оператора <code>nonlocal</code> : только Python 3.X	531
Состояние с помощью глобальных переменных: только одиночная копия	532
Состояние с помощью классов: явные атрибуты (предварительный обзор)	533
Состояние с помощью атрибутов функций: Python 3.X и 2.X	534
Резюме	538
Проверьте свои знания: контрольные вопросы	539
Проверьте свои знания: ответы	540
<b>ГЛАВА 18. Аргументы</b>	<b>541</b>
Основы передачи аргументов	541
Аргументы и разделяемые ссылки	542
Избегайте модификации изменяемых аргументов	544
Эмуляция выходных параметров и множественных результатов	545
Специальные режимы сопоставления аргументов	546
Основы сопоставления аргументов	547
Синтаксис сопоставления аргументов	548
Особенности использования специальных режимов сопоставления	549
Примеры ключевых слов и стандартных значений	550
Примеры произвольного количества аргументов	552
Аргументы с передачей только по ключевым словам Python 3.X	557
Функция <code>min</code>	560
Основная задача	561
Дополнительные очки	562

Заключение	563
Обобщенные функции для работы с множествами	563
Эмуляция функции <code>print</code> из Python 3.X	565
Использование аргументов с передачей только по ключевым словам	567
Резюме	569
Проверьте свои знания: контрольные вопросы	569
Проверьте свои знания: ответы	570
<b>ГЛАВА 19. Расширенные возможности функций</b>	571
Концепции проектирования функций	571
Рекурсивные функции	573
Суммирование с помощью рекурсии	574
Альтернативные варианты кода	574
Операторы цикла или рекурсия	576
Обработка произвольных структур	576
Объекты функций: атрибуты и аннотации	580
Косвенные вызовы функций: “первоклассные” объекты	580
Интроспекция функций	581
Атрибуты функций	582
Аннотации функций в Python 3.X	583
Анонимные функции: выражения <code>lambda</code>	585
Основы выражения <code>lambda</code>	586
Для чего используется выражение <code>lambda</code> ?	587
Как (не) запутать свой код на Python	589
Области видимости: выражения <code>lambda</code> также могут быть вложенными	590
Инструменты функционального программирования	591
Отображение функций на итерируемые объекты: <code>map</code>	592
Выбор элементов из итерируемых объектов: <code>filter</code>	594
Комбинирование элементов из итерируемых объектов: <code>reduce</code>	594
Резюме	596
Проверьте свои знания: контрольные вопросы	596
Проверьте свои знания: ответы	596
<b>ГЛАВА 20. Включения и генераторы</b>	598
Списковые включения и инструменты функционального программирования	598
Списковые включения или <code>map</code>	599
Добавление проверок и вложенных циклов: <code>filter</code>	600
Пример: списковые включения и матрицы	603
Не злоупотребляйте списковыми включениями: KISS	605
Генераторные функции и выражения	608
Генераторные функции: <code>yield</code> или <code>return</code>	608
Генераторные выражения: итерируемые объекты встречаются с включениями	614
Генераторные функции или генераторные выражения	618
Генераторы являются объектами с одиночной итерацией	620
Генерация во встроенных типах, инструментах и классах	623
Пример: генерация перемешанных последовательностей	626
Не злоупотребляйте генераторами: EIBTI	631
Пример: эмуляция <code>zip</code> и <code>map</code> с помощью итерационных инструментов	633

Сводка по синтаксису включений	639
Области видимости и переменные включений	639
Осмысление включений множеств и словарей	641
Расширенный синтаксис включений для множеств и словарей	642
Резюме	642
Проверьте свои знания: контрольные вопросы	643
Проверьте свои знания: ответы	643
<b>ГЛАВА 21. Оценочные испытания</b>	645
Измерение времени выполнения итерационных альтернатив	645
Модуль измерения времени: любительский	646
Сценарий измерения времени	651
Результаты измерения времени	652
Альтернативные версии модуля для измерения времени	655
Другие варианты	658
Измерение времени выполнения итераций и версий Python с помощью модуля <code>timeit</code>	659
Базовое использование <code>timeit</code>	659
Модуль и сценарий оценочных испытаний: <code>timeit</code>	664
Результаты запуска сценария оценочных испытаний	666
Продолжаем забавляться с оценочными испытаниями	668
Другие темы, связанные с оценочными испытаниями: <code>test pystone</code>	672
Затруднения, связанные с функциями	673
Локальные имена распознаются статически	673
Стандартные значения и изменяемые объекты	675
Функции без операторов <code>return</code>	677
Прочие затруднения, связанные с функциями	677
Резюме	678
Проверьте свои знания: контрольные вопросы	678
Проверьте свои знания: ответы	679
Проверьте свои знания: упражнения для части IV	679
<b>Часть V. Модули и пакеты</b>	683
<b>ГЛАВА 22. Модули: общая картина</b>	684
Для чего используются модули?	684
Архитектура программы Python	685
Структурирование программы	686
Импортирование и атрибуты	686
Стандартные библиотечные модули	688
Как работает импортирование	689
1. Поиск файла модуля	689
2. Компиляция файла модуля (возможная)	690
3. Выполнение файла модуля	691
Файлы байт-кода: <code>__pycache__</code> в Python 3.2+	691
Модели файлов байт-кода в действии	692
Путь поиска модулей	693
Конфигурирование пути поиска	696

Вариации пути поиска	696
Список <code>sys.path</code>	697
Выбор файла модуля	698
Резюме	700
Проверьте свои знания: контрольные вопросы	701
Проверьте свои знания: ответы	701
<b>ГЛАВА 23. Основы написания модулей</b>	702
Создание модулей	702
Имена файлов модулей	702
Другие виды модулей	703
Использование модулей	703
Оператор <code>import</code>	703
Оператор <code>from</code>	704
Оператор <code>from *</code>	704
Операции импортирования происходят только однократно	705
Операторы <code>import</code> и <code>from</code> являются присваиваниями	706
Эквивалентность <code>import</code> и <code>from</code>	707
Потенциальные затруднения, связанные с оператором <code>from</code>	708
Пространства имен модулей	709
Файлы генерируют пространства имен	709
Словари пространств имен: <code>__dict__</code>	711
Уточнение имен атрибутов	712
Импортирование или области видимости	712
Вложение пространств имен	713
Перезагрузка модулей	714
Основы использования <code>reload</code>	715
Пример использования <code>reload</code>	716
Резюме	718
Проверьте свои знания: контрольные вопросы	718
Проверьте свои знания: ответы	719
<b>ГЛАВА 24. Пакеты модулей</b>	720
Основы импортирования пакетов	721
Пакеты и настройки пути поиска	721
Файлы <code>__init__.py</code> пакетов	722
Пример импортирования пакетов	724
Использование <code>from</code> или <code>import</code> с пакетами	726
Для чего используется импортирование пакетов?	727
История о трех системах	727
Относительное импортирование пакетов	730
Изменения в Python 3.X	731
Основы относительного импортирования	732
Для чего используются операции относительного импортирования?	733
Границы действия операций относительного импортирования	736
Сводка по правилам поиска модулей	736
Операции относительного импортирования в действии	737
Затруднения, связанные с операциями импортирования относительно пакетов: смешанное использование	742

Пакеты пространств имен, введенные в Python 3.3	748
Семантика пакетов пространств имен	749
Влияние на обычные пакеты: необязательность <code>__init__.py</code>	750
Пакеты пространств имен в действии	751
Вложение пакетов пространств имен	752
Файлы по-прежнему имеют приоритет над каталогами	753
Резюме	756
Проверьте свои знания: контрольные вопросы	756
Проверьте свои знания: ответы	756
<b>ГЛАВА 25. Расширенные возможности модулей</b>	758
Концепции проектирования модулей	758
Соккрытие данных в модулях	760
Сведение к минимуму вреда от <code>from *: _X</code> и <code>__all__</code>	760
Включение будущих языковых средств: <code>__future__</code>	761
Смешанные режимы использования: <code>__name__</code> и <code>__main__</code>	762
Модульное тестирование с помощью <code>__name__</code>	763
Пример: код с двойным режимом	764
Символы валют: <code>Unicode</code> в действии	767
Строки документации: документация по модулям в работе	769
Изменение пути поиска модулей	770
Расширение <code>as</code> для операторов <code>import</code> и <code>from</code>	771
Пример: модули являются объектами	772
Импортирование модулей по строкам с именами	775
Выполнение строк с кодом	775
Прямые вызовы: два варианта	776
Пример: транзитивная перезагрузка модулей	777
Инструмент рекурсивной перезагрузки	777
Альтернативные реализации	780
Затруднения, связанные с модулями	784
Конфликты имен модулей: операции импортирования пакетов и относительно пакетов	784
Порядок следования операторов в коде верхнего уровня имеет значение	785
Оператор <code>from</code> копирует имена, но не ссылки на них	786
Форма оператора <code>from *</code> может сделать неясным смысл переменных	787
Функция <code>reload</code> может не оказывать влияния на результаты операторов импортирования <code>from</code>	787
<code>reload</code> , <code>from</code> и тестирование в интерактивном сеансе	788
Рекурсивные операции импортирования <code>from</code> могут не работать	789
Резюме	790
Проверьте свои знания: контрольные вопросы	791
Проверьте свои знания: ответы	791
Проверьте свои знания: упражнения для части V	792
<b>ПРИЛОЖЕНИЕ. Решения упражнений, приводимых в конце частей</b>	794
<b>Предметный указатель</b>	819

# Операторы и синтаксис

## Введение в операторы Python

Теперь, когда вы знакомы с основными встроенными типами объектов Python, мы приступаем к исследованию главных форм операторов Python. Как и в предыдущей части, мы начнем с обобщенного введения в синтаксис операторов, после чего в последующих нескольких главах более подробно рассмотрим специфические операторы.

Попросту говоря, *операторы* — это то, что вы пишете для сообщения Python о том, что должны делать ваши программы. Если согласно главе 4 программы “делают дела с помощью оснащения”, тогда операторы являются способом указания вида *дел*, которые делает программа. Более формально Python представляет собой процедурный, основанный на операторах язык; за счет комбинирования операторов вы задаете *процедуру*, которую Python выполняет для достижения целей программы.

### Еще раз о концептуальной иерархии Python

Другой способ постижения роли операторов предусматривает мысленный возврат к концептуальной иерархии, введенной в главе 4, где речь шла о встроенных объектах и выражениях, используемых для манипулирования ими. В настоящей главе иерархия поднимается на следующий уровень структуры программ Python.

1. Программы состоят из модулей.
2. Модули содержат операторы.
3. Операторы содержат выражения.
4. Выражения создают и обрабатывают объекты.

В своей основе программы, написанные на языке Python, состоят из операторов и выражений. Выражения обрабатывают объекты и встраиваются в операторы. Операторы кодируют более крупную *логику* работы программы — они применяют и направляют выражения для обработки объектов, которые вы изучали в предшествующих главах. Более того, операторы представляют собой место, где появляются объекты (например, в выражениях внутри операторов присваивания), а некоторые операторы создают совершенно новые виды объектов (функции, классы и т.д.). На верхнем уровне операторы всегда существуют в модулях, которые сами управляются с помощью операторов.



# Операторы Python

В табл. 10.1 приведена сводка по набору операторов Python. Каждый оператор Python имеет собственное специфическое назначение и собственный специфический *синтаксис* — правила, определяющие его структуру, — хотя, как мы увидим, многие разделяют общие синтаксические шаблоны, а роли ряда операторов перекрываются. В табл. 10.1 также даны примеры каждого оператора, закодированные в соответствии с его синтаксическими правилами. В ваших программах такие единицы кода могут выполнять действия, повторять задачи, делать выбор, создавать более крупные программные структуры и т.д.

В настоящей части книги обсуждаются операторы с первой строки таблицы вплоть до `break` и `continue`. Вам уже неформально было представлено несколько операторов из табл. 10.1; в этой части книги мы восполним опущенные ранее детали, ознакомим с остатком набора процедурных операторов Python и раскроем полную синтаксическую модель. Операторы, расположенные ниже в табл. 10.1, которые касаются более крупных программных единиц — функций, классов, модулей и исключений — подводят к более крупным понятиям программирования, так что каждый из них будет рассматриваться в отдельном разделе. Более специализированные операторы (вроде `del`, который удаляет разнообразные компоненты) раскрываются в других местах книги, а также описаны в стандартных руководствах по Python.

**Таблица 10.1. Операторы Python**

Оператор	Роль	Пример
Присваивания	Создание ссылок	<code>a, b = 'good', 'bad'</code>
Вызовы и другие выражения	Выполнение функций	<code>log.write("spam, ham")</code>
Вызовы <code>print</code>	Вывод объектов	<code>print('The Killer', joke)</code>
<code>if/elif/else</code>	Выбор действий	<code>if "python" in text:     print(text)</code>
<code>for/else</code>	Итерация	<code>for x in mylist:     print(x)</code>
<code>while/else</code>	Универсальные циклы	<code>while X &gt; Y:     print('hello')</code>
<code>pass</code>	Пустой заполнитель	<code>while True:     pass</code>
<code>break</code>	Выход из цикла	<code>while True:     if exittest(): break</code>
<code>continue</code>	Продолжение цикла	<code>while True:     if skiptest(): continue</code>
<code>def</code>	Функции и методы	<code>def f(a, b, c=1, *d):     print(a+b+c+d[0])</code>
<code>return</code>	Результаты функций	<code>def f(a, b, c=1, *d):     return a+b+c+d[0]</code>
<code>yield</code>	Генераторные функции	<code>def gen(n):     for i in n: yield i*2</code>

Оператор	Роль	Пример
<code>global</code>	Пространства имен	<pre>x = 'old' def function():     global x, y; x = 'new'</pre>
<code>nonlocal</code>	Пространства имен (Python 3.X)	<pre>def outer():     x = 'old'     def function():         nonlocal x; x = 'new'</pre>
<code>import</code>	Доступ к модулям	<pre>import sys</pre>
<code>from</code>	Доступ к атрибутам	<pre>from sys import stdin</pre>
<code>class</code>	Построение объектов	<pre>class Subclass(Superclass):     staticData = []     def method(self): pass</pre>
<code>try/except/finally</code>	Перехват исключений	<pre>try:     action() except:     print('action error')</pre>
<code>raise</code>	Генерация исключений	<pre>raise EndSearch(location)</pre>
<code>assert</code>	Отладочные проверки	<pre>assert X &gt; Y, 'X too small'</pre>
<code>with/as</code>	Диспетчеры контекста (Python 3.X, 2.6+)	<pre>with open('data') as myfile:     process(myfile)</pre>
<code>del</code>	Удаление ссылок	<pre>del data[k] del data[i:j] del obj.attr del variable</pre>

Формально в табл. 10.1 воспроизведены операторы Python 3.X. Хотя этого перечня операторов достаточно для обзора и справочных целей, в том виде, как есть, он не совсем полон. Ниже описано несколько тонкостей относительно его содержимого.

- Операторы присваивания имеют различные синтаксические формы, описанные в главе 11: базовая, последовательности, дополненная и др.
- Формально `print` в Python 3.X не является ни зарезервированным словом, ни оператором, а вызовом встроенной функции; но поскольку `print` почти всегда будет выполняться в виде оператора с выражением (и часто занимать отдельную строку), такой вызов в общем случае трактуется как разновидность оператора. Операции вывода рассматриваются в главе 11.
- Начиная с Python 2.5, `yield` — также выражение, а не оператор; подобно `print` оно обычно используется как оператор с выражением и потому включено в табл. 10.1, но в главе 20 мы увидим, что сценарии иногда присваивают или по-другому применяют его результат. Будучи выражением, `yield` также является ключевым словом в отличие от `print`.

Большая часть табл. 10.1 применима и к Python 2.X за исключением случаев, когда это не так — если вы используете Python 2.X, то ниже приведено несколько замечаний, которые его касаются.

- В Python 2.X оператор `nonlocal` недоступен; как будет показано в главе 17, существуют альтернативные способы достижения того же эффекта сохранения перезаписываемого состояния.
- В Python 2.X `print` представляет собой оператор, а не вызов встроенной функции, со специфическим синтаксисом, раскрываемым в главе 11.
- В Python 2.X встроенная функция выполнения кода `exec` из Python 3.X является оператором со специфическим синтаксисом; однако поскольку он поддерживает окружающие круглые скобки, вы можете применять в коде Python 2.X форму вызова `exec`, принятую в Python 3.X.
- В Python 2.5 операторы `try/except` и `try/finally` были объединены: раньше они представляли собой два отдельных оператора, но теперь `except` и `finally` можно записывать в том же самом операторе `try`.
- В Python 2.5 оператор `with/as` является необязательным расширением, которое не будет доступным до тех пор, пока вы явно не включите его, выполнив оператор `from __future__ import with_statement` (см. главу 34).

## История о двух `if`

Прежде чем мы погрузимся в детали конкретных операторов из табл. 10.1, я хочу начать обзор синтаксиса операторов Python, показав вам то, что вы *не* собираетесь вводить в коде Python. Это даст возможность сравнить синтаксис операторов Python с другими синтаксическими модулями, которые вы могли видеть в прошлом.

Взгляните на следующий оператор `if`, запрограммированный на С-подобном языке:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Оператор может относиться к С, С++, Java, JavaScript или похожему языку. А вот эквивалентный оператор в языке Python:

```
if x > y:
    x = 1
    y = 2
```

Первое, что может броситься в глаза — эквивалентный оператор Python не настолько загроможден, т.е. в нем меньше синтаксических компонентов. Так было задумано; одна из целей Python как языка написания сценариев заключается в том, чтобы облегчить жизнь программистов, требуя меньшего объема набора.

В частности, сравнив две синтаксические модели, вы заметите, что Python добавляет один элемент к смеси, а три элемента, присутствующие в С-подобном языке, в коде Python отсутствуют.

## Что Python добавляет

Новым компонентом синтаксиса в Python является символ двоеточия (`:`). Все *составные операторы* Python, т.е. операторы с вложенными другими операторами, соблюдают общий шаблон. Шаблон состоит из строки заголовка, завершаемой двоеточием, и вложенного блока кода, обычно указываемого с отступом под строкой заголовка:

```
Строка заголовка:
    Вложенный блок операторов
```

Двоеточие обязательно, а его отсутствие является, пожалуй, наиболее распространенной ошибкой, допускаемой начинающими программистами на Python — безусловно, мне тысячи раз приходилось быть свидетелем такой ошибки в группах студентов, которых я обучал. На самом деле, если вы новичок в Python, вы почти наверняка вскоре забудете о символе двоеточия. В таком случае вы получите сообщение об ошибке, а большинство дружественных к Python редакторов позволят легко выявить такое недоразумение. Со временем включение двоеточия становится неосознанной привычкой (настолько сильной, что вы можете начать вводить двоеточия также и в коде на C-подобном языке, приводя к выдаче забавных сообщений об ошибках компилятором этого языка!).

## Что Python устраняет

Несмотря на то что Python требует добавочного символа двоеточия, программистам на C-подобных языках приходится включать три элемента, которые обычно в Python не нужны.

### Круглые скобки необязательны

Первый элемент — набор круглых скобок вокруг проверок в верхней части оператора:

```
if (x < y)
```

Круглые скобки здесь требуются синтаксисом многих C-подобных языков. Тем не менее, в Python они необязательны — мы просто опускаем круглые скобки, и оператор работает аналогичным образом:

```
if x < y
```

Говоря формально, из-за того, что любое выражение может быть помещено в круглые скобки, их наличие не повредит в таком коде Python, и они не трактуются как ошибка, когда присутствуют.

*Но не поступайте так:* вы будете без нужды изнашивать свою клавиатуру и заявлять всему миру о том, что являетесь программистом на C-подобном языке, все еще изучающим Python (я знаю, потому как сам был таким). “Образ действий Python” заключается в том, чтобы просто опускать круглые скобки в операторах такого вида.

### Конец строки является концом оператора

Второй и более важный элемент синтаксиса, который вы не обнаружите в коде Python — точка с запятой. В Python вы не обязаны завершать операторы точками с запятой, как принято в C-подобных языках:

```
x = 1;
```

Общее правило в Python состоит в том, что конец строки автоматически завершает оператор, находящийся в этой строке. Другими словами, вы можете избавиться от точки с запятой, и оператор будет работать прежним образом:

```
x = 1
```

Как вы вскоре увидите, есть несколько способов обойти указанное правило (скажем, помещение кода внутрь структуры в квадратных скобках позволяет разнести ее на множество строк). Но в подавляющем большинстве кода вы будете записывать по одному оператору в строке, и никакие точки с запятой не потребуются.

Если вы тоскуете по тем дням, когда программировали на С (при условии, что такое вообще возможно), то можете продолжать использовать точки с запятой в конце каждого оператора — язык в состоянии смириться с их присутствием, поскольку точка с запятой служит также разделителем при комбинировании операторов.

*Но не поступайте так тоже (серьезно!).* Тем самым вы опять сообщаете миру о том, что являетесь программистом на С-подобном языке, который еще не полностью переключился на стиль написания кода Python. Стиль Python предусматривает отказ от точек с запятой. Судя по студентам в группах, некоторым программистам со стажем, похоже, нелегко отказаться от такой привычки. Но вы добьетесь своего; точки с запятой в роли завершения операторов — бесполезные помехи в Python.

## Конец отступа является концом блока

Третий и последний элемент синтаксиса, устранившийся в Python, отсутствие которого может выглядеть самым необычным для тех, кто недавно прекратил программировать на С-подобных языках, связан с тем, что вы не набираете в коде что-нибудь явное для синтаксической пометки начала и конца вложенного блока кода. Вам не нужно помещать вложенный блок внутрь `begin/end`, `then/endif` или фигурных скобок, как вы поступали бы в С-подобных языках:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

В Python взамен мы согласованно смещаем все операторы в заданном одиночном вложенном блоке на одно и то же расстояние вправо, и для определения начала и конца блока Python применяет физические отступы операторов:

```
if x > y:
    x = 1
    y = 2
```

Под *отступом* здесь подразумевается пустое пространство слева от двух вложенных операторов. Python вовсе не заботит то, *как* сделан отступ (можно использовать либо пробелы, либо табуляции), или то, насколько отступ *большой* (допускается применять любое количество пробелов или табуляций). На самом деле отступ одного вложенного блока может совершенно отличаться от отступа другого. Синтаксическое правило лишь гласит о том, что все операторы заданного одиночного вложенного блока должны быть смещены на то же самое расстояние вправо. В противном случае возникает синтаксическая ошибка, а код не запустится до тех пор, пока вы не приведете отступы в согласованное состояние.

## Для чего используется синтаксис с отступами?

Правило отступа может показаться на первый взгляд необычным программистам, привыкшим к С-подобным языкам, но это преднамеренная особенность Python и один из главных способов, которыми Python вынуждает программистов производить единообразный, систематический и читабельный код. Правило отступа по существу означает, что вы обязаны выравнивать свой код вертикально по столбцам в соответствии с его логической структурой. Совокупный эффект заключается в том, что код становится более согласованным и читабельным (в отличие от большинства кода на С-подобных языках).

Выражаясь более строго, выравнивание кода в соответствии с его логической структурой является значительной работой по содействию его читабельности и тем самым многократному использованию и удобству сопровождения, как вами, так и другими. В действительности, даже если вы не будете применять Python после прочтения книги, то все равно должны выработать привычку выравнивать свой код в целях читабельности в любом блочно-структурированном языке. Python акцентирует внимание на данной задаче, сделав выравнивание частью синтаксиса, но поступать так важно в любом языке программирования, что оказывает огромное влияние на полезность результирующего кода.

Ваш опыт может отличаться, но когда я все еще занимался разработкой на постоянной основе, мне главным образом платили за работу над крупными старыми программами C++, которые создавались многими программистами в течение нескольких лет. Почти у каждого программиста был свой стиль отступов в коде. Например, меня часто просили изменить цикл `while` в коде C++, который начинался примерно так:

```
while (x > 0) {
```

До того, как мы углубимся в отступы, следует отметить, что есть три или четыре способа, которыми программисты размещают фигурные скобки в C-подобном языке, и в организациях часто ведутся горячие споры и составляются руководства по стандартам, регламентирующие допустимые варианты (что выглядит более чем просто отклонением от задач, решаемых с помощью программирования). Как бы то ни было, вот вам сценарий, который я часто встречал в коде C++. Первый человек, работающий над кодом, использовал для тела цикла отступ в четыре пробела:

```
while (x > 0) {
    -----;
    -----;
```

Со временем этот человек перешел на руководящую должность, а его заменил другой человек, которому нравилось делать еще большие отступы вправо:

```
while (x > 0) {
    -----;
    -----;
        -----;
        -----;
```

Позже любитель широких отступов воспользовался возможностью и сменил работу (положив конец царствованию кодового террора в своем лице...), а его место занял человек, который предпочитал делать поменьше отступов:

```
while (x > 0) {
    -----;
    -----;
        -----;
        -----;
-----;
-----;
}
```

И так далее. В конце концов, блок завершается закрывающей фигурной скобкой (`}`), которая, конечно же, делает его “блочным-структурированным кодом” (здесь присутствует изрядная доля сарказма). Нет: в любом блочно-структурированном языке, Python или еще каком, если вложенные блоки не имеют согласованных отступов, тогда их становится очень трудно истолковывать, изменять или повторно использовать, пото-

му что код больше визуально не отражает свой логический смысл. *Читабельность имеет значение*, и отступы являются важной составной частью читабельности.

Ниже приведен пример, на котором вы могли обжечься в прошлом, если много программировали на каком-то С-подобном языке. Взгляните на следующий оператор в С:

```
if (x)
    if (y)
        оператор1;
else
    оператор2;
```

К какому оператору `if` здесь относится `else`? Как ни удивительно, но в С часть `else` относится к вложенному оператору `if` (`if (y)`), несмотря на то, что визуально выглядит связанной с внешним оператором `if (x)`. Это классическая ловушка в языке С; она может привести к тому, что читатель совершенно неправильно интерпретирует код и некорректно изменяет его способами, которые иногда обнаруживаются лишь после того, как марсоход врезался в огромную скалу!

В Python подобное произойти не может — из-за того, что отступы существенны, код работает именно так, как выглядит. Вот эквивалентный оператор Python:

```
if x:
    if y:
        оператор1
else:
    оператор2
```

В приведенном примере часть `else` логически связана с оператором `if`, с которым выровнена по вертикали (внешний `if x`). В известной степени Python является языком WYSIWYG (what you see is what you get — что видишь, то и получаешь), поскольку внешний вид кода определяет путь его выполнения независимо от того, кто его писал.

Если сказанного по-прежнему не хватает, чтобы должным образом оценить преимущества синтаксиса Python, то приведу еще один эпизод. В начале своей карьеры я работал в успешной компании, которая занималась разработкой системного программного обеспечения на языке С, где согласованные отступы не требовались. Даже в таких условиях, когда в конце дня мы сохраняли свой код в системе управления версиями исходного кода, в компании автоматически запускался сценарий, который анализировал отступы, применяемые в коде. Если сценарий замечал, что мы использовали отступы в коде несогласованно, то на следующее утро мы получали по электронной почте соответствующее сообщение — равно как и наши менеджеры!

Дело в том, что даже когда язык этого не требует, хорошим программистам известно, что согласованное применение отступов оказывает сильное влияние на читабельность и качество кода. Тот факт, что отступы в Python продвинуты до уровня синтаксиса, рассматривается большинством как особенность языка.

Также имейте в виду, что почти каждый дружественный к программистам текстовый редактор обладает встроенной поддержкой синтаксической модели Python. Скажем, в графическом пользовательском интерфейсе IDLE к строкам кода автоматически добавляется отступ при наборе вложенного блока; нажатие клавиши забота возвращает обратно на один уровень отступов, и то, насколько далеко вправо IDLE смещает операторы во вложенном блоке, можно настраивать. Универсального стандарта не существует: общепринятыми являются четыре пробела или одна табуляция, но обычно вы сами решаете, каким образом и насколько отступать (если только не работаете в компании, где отступы регламентируются политикой и внутренними стан-

дартами). Делайте большой отступ вправо для следующего вложенного блока и меньший отступ для закрытия предыдущего блока.

В качестве эмпирического правила запомните, что вероятно не стоит смешивать табуляции и пробелы в том же самом блоке в Python, если только не делать это согласованно; в отдельно взятом блоке используйте табуляции или пробелы, но не то и другое (на самом деле, как будет показано в главе 12, теперь Python 3.X сообщает об ошибке при несогласованном применении табуляций и пробелов). Более того, смешивать табуляции и пробелы в отступах, видимо, не следует в *любом* структурированном языке — такой код может вызвать крупные проблемы с читабельностью, если текстовый редактор у следующего программиста настроен на отображение табуляций не так, как у вас. С-подобные языки зачастую разрешают программистам обходить данное правило, но программисты не должны поступать так: результатом может оказаться изрядно запутанная смесь.

Независимо от языка, на котором пишется код, вы должны придерживаться согласованных отступов для обеспечения читабельности. Фактически, если вас не приучили делать это раньше, то ваши учителя оказали вам медвежью услугу. Большинство программистов (особенно те, кому приходится читать код, написанный другими) считает ценным качеством то, что Python продвигает правило, касающееся отступов, до уровня синтаксиса. Кроме того, инструментам, которые должны выводить код Python, не составляет труда генерировать табуляции вместо фигурных скобок. В целом, если вы делаете то, что в любом случае должны были бы делать в С-подобном языке, но избавляетесь от фигурных скобок, тогда ваш код будет удовлетворять правилам синтаксиса Python.

## Несколько специальных случаев

Как упоминалось ранее, в синтаксической модели Python:

- конец строки завершает оператор в этой строке (безо всяких точек с запятой);
- вложенные операторы объединяются в блок и ассоциируются согласно их физическим отступам (без фигурных скобок).

Указанные правила охватывают почти весь код Python, который вам доведется писать или встречать в реальности. Однако Python также предлагает ряд специализированных правил, которые делают возможной настройку как операторов, так и вложенных блоков операторов. Они не обязательны и должны использоваться умеренно, но программисты находят их полезными на практике.

### Специальные правила для операторов

Хотя операторы обычно располагают по одному в строке, в Python разрешено помещать несколько операторов в одну строку, разделяя их точками с запятой:

```
a = 1; b = 2; print(a + b)    # Три оператора в одной строке
```

Это единственное место в Python, где требуются точки с запятой: в качестве *разделителей между операторами*. Тем не менее, такое применение допускается, только если объединяемые подобным образом операторы сами не являются составными. Другими словами, вы можете выстраивать в цепочку лишь простые операторы вроде присваиваний, вызовов `print` и обращений к функциям. Составные операторы, такие как проверки `if` и циклы `while`, по-прежнему должны находиться в собственных строках (иначе вы могли бы втиснуть всю программу в одну строку, что вряд ли добавило бы вам популярности среди коллег по работе!).



Другое специальное правило для операторов по существу является противоположностью: вы можете разнести одиночный оператор на *множество строк*. Для этого понадобится лишь поместить часть оператора внутрь пары скобок — круглых (`()`), квадратных (`[]`) или фигурных (`{}`). Любой код, заключенные в такие скобки, может занимать несколько строк: оператор не закончится до тех пор, пока Python не достигнет закрывающей скобки из пары. Например, вот как записать списковый литерал в нескольких строках:

```
mylist = [1111,
          2222,
          3333]
```

Поскольку код заключен в пару квадратных скобок, Python просто переходит на следующую строку до тех пор, пока не встретит закрывающую квадратную скобку. Фигурные скобки, окружающие словари (а также литералы множеств и включения словарей и множеств в Python 3.X/2.7), позволяют им распространяться на несколько строк, а круглые скобки делают это для кортежей, вызовов функций и выражений. Отступы в строках продолжения роли не играют, хотя здравый смысл подсказывает, что строки должны каким-то образом выравниваться ради читабельности.

Круглые скобки представляют собой универсальное средство — из-за того, что в них можно помещать любое выражение, вставка открывающей круглой скобки дает возможность продолжить оператор в следующей строке:

```
X = (A + B +
     C + D)
```

Кстати, такая методика работает также и с составными операторами. Везде, где нужно записать крупное выражение, просто поместите его в круглые скобки, чтобы перенести на следующую строку:

```
if (A == 1 and
    B == 2 and
    C == 3):
    print('spam' * 3)
```

Более старое правило позволяет продолжать оператор в следующей строке, если предыдущая строка заканчивается на обратную косую черту:

```
X = A + B + \
    C + D # Подверженная ошибкам более старая альтернатива
```

Однако такая альтернативная методика вышла из употребления и в наши дни не одобряется, потому что замечать и сохранять обратные косые черты нелегко. Она также довольно хрупкая и подвержена ошибкам. Дело в том, что после обратной косой черты не должно быть пробелов, а случайный пропуск обратной косой черты может приводить к неожиданным эффектам, если следующая строка будет ошибочно воспринята как новый оператор. (В показанном выше примере `C + D` — сам по себе допустимый оператор, если он не имеет отступа.) Такое правило также является еще одним отголоском языка C, где оно обычно используется в макросах `#define`; если вы находитесь в мире Python, то и поступайте, как принято в Python, а не в C.

## Специальные правила для блоков

Как упоминалось ранее, операторы во вложенном блоке кода обычно ассоциируются по их отступам на одно и то же расстояние вправо. В качестве одного специального случая здесь тело составного оператора может взамен находиться в той же самой строке, что и строка заголовка оператора Python, после двоеточия:

```
if x > y: print(x)
```

В результате у нас появляется возможность записывать однострочные операторы `if`, однострочные циклы `while` и `for` и т.д. Тем не менее, это будет работать, только если тело составного оператора само не содержит каких-либо составных операторов. То есть после двоеточия разрешено указывать лишь простые операторы – присваивания, вызовы `print`, обращения к функциям и т.п. Более крупные операторы по-прежнему должны находиться в собственных строках. Дополнительные части составных операторов (такие как часть `else` оператора `if`, который мы рассмотрим в следующем разделе) также обязаны располагаться в отдельных строках. Тела составных операторов могут состоять из множества простых операторов, разделенных точками с запятой, но обычно такой подход не одобряется.

В общем, хотя временами это необязательно, если вы будете размещать все операторы в отдельных строках и всегда делать отступы для вложенных блоков, то ваш код станет легче читать и изменять в будущем. Более того, некоторые инструменты для профилирования и покрытия кода могут быть не в состоянии проводить различия между множеством операторов, втиснутых в одну строку, или заголовком и телом однострочного составного оператора. Практически всегда в ваших интересах поддерживать в Python простоту. Вы можете применить специальные правила для написания кода Python, который трудно читать, но это требует немало работы, и у вас наверняка найдется, куда лучше потратить свое время.

Однако чтобы увидеть в действии простое и распространенное исключение из указанных правил (использование однострочного оператора `if` для выхода из цикла с помощью `break`), а также представить дополнительный синтаксис Python, в следующем разделе будет написан реальный код.

## Короткий пример: интерактивные циклы

Мы увидим все описанные правила синтаксиса в действии, когда начнем тур по специфическим составным операторам Python в последующих нескольких главах, но повсюду в языке Python они работают одинаково. Давайте пока рассмотрим короткий, но реалистичный пример, который продемонстрирует практический способ совместного применения синтаксиса операторов и вложения операторов, а также попутно ознакомит с несколькими операторами.

### Простой интерактивный пример

Предположим, что вас попросили написать на Python программу, которая взаимодействует с пользователем в окне консоли. Возможно, необходимо принимать ввод для отправки в базу данных или читать числа с целью использования в вычислении. Независимо от предназначения, вам придется написать код цикла, который читает одну или большее количество строк, набираемых пользователем на клавиатуре, и выводит для каждой результат. Другими словами, вы должны написать программу с классическим циклом чтение/оценка/вывод.

Типичный шаблонный код для такого цикла взаимодействия может выглядеть в Python следующим образом:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

В коде применяется несколько новых идей и ряд тех, что вы уже видели ранее.

- В коде задействован цикл `while` языка Python — наиболее универсальный оператор цикла. Позже мы детально исследуем оператор `while`, а пока достаточно знать, что он состоит из слова `while`, за которым следует выражение, дающее истинный или ложный результат, а за ним вложенный блок кода, повторяющийся до тех пор, пока выражение остается истинным (слово `True` здесь считается всегда истинным).
- Встроенная функция `input`, которую мы встречали ранее в книге, используется для универсального консольного ввода — она выводит необязательный аргумент в качестве приглашения и возвращает то, что пользователь набрал, в виде строки. Согласно приведенной далее врезке “На заметку!” в Python 2.X взамен применяется `raw_input`.
- В коде также присутствует однострочный оператор `if`, который использует специальное правило для вложенных блоков: тело оператора `if` находится в строке заголовка после двоеточия вместо того, чтобы располагаться с отступом в строке ниже. В любом случае оператор будет работать одинаково, но так мы экономим одну строку.
- Наконец, оператор `break` языка Python применяется для немедленного выхода из цикла — происходит просто переход из цикла и программа продолжает выполнение сразу после оператора цикла. Без такого оператора выхода цикл `while` выполнялся бы бесконечно, потому что его проверка всегда истинна.

В сущности, приведенная комбинация операторов означает “читать строки, введенные пользователем, и выводить их в верхнем регистре до тех пор, пока пользователь не введет слово `stop`”. Существуют и другие способы написания такого цикла, но использованная здесь форма очень часто встречается в коде Python.

Обратите внимание, что все три строки, вложенные под строку заголовка `while`, смещены на одно и то же расстояние вправо — поскольку операторы подобным образом выровнены вертикально по столбцам, они являются блоком кода, который ассоциирован с проверкой `while` и повторяется. Для завершения блока с телом цикла будет достаточно либо конца файла исходного кода, либо оператора с меньшим отступом.

Когда код запускается, интерактивно или как файл сценария, вот какое взаимодействие мы получаем (весь код примера находится в файле `interact.py` из пакета примеров для этой книги):

```
Enter text: spam
SPAM
Enter text: 42
42
Enter text: stop
```



*Примечание, касающееся нестыковки версий.* В примере написан код для Python 3.X. Если вы работаете в Python 2.X, то код работает так же, но во всех примерах текущей главы вместо `input` придется применять `raw_input`, и можно опустить внешние круглые скобки в операторах `print` (хотя никакого вреда они не наносят). В действительности, если вы исследуете файл `interact.py` из пакета примеров, то увидите, что в нем это делается автоматически — для поддержки совместимости с Python 2.X имя `input` переустанавливается, если старшим номером версии функционирующего Python является 2 (вызов `input` приводит к выполнению `raw_input`):

```
import sys
if sys.version[0] == '2': input = raw_input # Совместимость
                                         # с Python 2.X
```

В Python 3.X имя `raw_input` было изменено на `input`, а `print` представляет собой встроенную функцию, а не оператор (более подробно `print` обсуждается в следующей главе). В Python 2.X также имеется оператор `input`, но он пытается выполнить входную строку, как если бы она было кодом Python, что вероятно не будет работать в данном контексте; в Python 3.X того же эффекта можно достичь посредством `eval(input())`.

## Выполнение математических действий над пользовательским вводом

Наш сценарий работает, но теперь предположим, что вместо преобразования текстовой строки в верхний регистр мы хотим выполнить какие-то математические действия над числовым вводом — скажем, возвести его в квадрат, возможно в ошибочной попытке программы ввода возраста подразнить своих пользователей. Для получения желаемого результата мы могли бы использовать код вроде показанного ниже:

```
>>> reply = '20'
>>> reply ** 2
...текст сообщения об ошибке не показан...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Ошибка типа: неподдерживаемые типы операндов для ** или pow(): str и int
```

Тем не менее, прием в нашем сценарии работать не будет, потому что (как обсуждалось в предыдущей части книги) Python не преобразует типы в выражениях, если только все они не являются числовыми, а ввод от пользователя всегда возвращается сценарию в виде *строки*. Мы не сможем возвести строку цифр в степень до тех пор, пока вручную не преобразуем ее в целое число:

```
>>> int(reply) ** 2
400
```

Вооружившись этой информацией, мы можем переписать наш цикл для выполнения необходимого математического действия. Поместите следующий код в файл для его тестирования:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

Как и ранее, в сценарии применяется однострочный оператор `if` для выхода из цикла при вводе `stop`, но также осуществляется преобразование ввода для выполнения требуемого математического действия. В конце еще добавляется прощальное сообщение. Поскольку оператор `print` в последней строке не имеет такого же отступа, как у вложенного блока кода, он не считается частью тела цикла и будет выполнен только раз после выхода из цикла:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```



*Замечание по использованию.* С этого момента я буду предполагать, что код хранится в файле сценария и запускается из него через командную строку, пункт меню IDLE или любым другим способом запуска файлов, которые обсуждались в главе 3. В пакете примеров файл называется `interact.py`. Однако если вы вводите этот код интерактивно, тогда удостоверьтесь в том, что включили пустую строку (т.е. два раза нажали клавишу <Enter>) перед финальным оператором `print`, чтобы завершить цикл. Это также подразумевает невозможность вырезания и вставки кода в интерактивную подсказку: добавочная пустая строка требуется в интерактивном режиме, но не в файле сценария. Хотя в наборе последнего оператора `print` в интерактивном режиме мало смысла — вы будете вводить его после взаимодействия с циклом!

## Обработка ошибок путем проверки ввода

Пока все хорошо, но обратите внимание, что происходит в случае ввода недопустимой строки:

```
Enter text:xxx
...текст сообщения об ошибке не показан...
ValueError: invalid literal for int() with base 10: 'xxx'
Ошибка значения: недопустимый литерал для int() с основанием 10: xxx
```

Столкнувшись с ошибкой, встроенная функция `int` генерирует исключение. Если нужно, чтобы сценарий был надежным, тогда можно заранее проверить содержимое строки с помощью метода `isdigit` строкового объекта:

```
>>> s = '123'
>>> t = 'xxx'
>>> s.isdigit(), t.isdigit()
(True, False)
```

Это также дает повод к дальнейшему вложению операторов в примере. В приведенной далее обновленной версии нашего интерактивного сценария с применением полнофункционального оператора `if` исключение при ошибках обходится:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

Оператор `if` подробно рассматривается в главе 12, но сейчас достаточно знать, что он представляет собой довольно легковесный инструмент для кодирования логики в сценариях. В своей полной форме оператор состоит из слова `if`, за которым следует проверка и связанный блок кода, одна или более необязательных проверок `elif` (“else if”) и блоков кода, а также необязательная часть `else` с ассоциированным блоком кода в качестве принимаемого по умолчанию. Python выполняет блок кода, связанный с первой проверкой, которая дает истину, работая сверху вниз, или часть `else`, если все проверки оказались ложными.

Части `if`, `elif` и `else` в предыдущем примере относятся к одному и тому же оператору, потому что все они выровнены по вертикали (т.е. используют одинако-

вый уровень отступа). Оператор `if` простирается от слова `if` до начала оператора `print` в последней строке сценария. В свою очередь весь блок `if` является частью цикла `while`, поскольку он располагается с отступом под строкой заголовка цикла. Подобного рода вложение операторов станет выглядеть естественным, как только вы освоитесь с ним.

После запуска нового сценария его код перехватывает ошибки до того, как они произойдут, и выводит сообщение, прежде чем продолжить работу (которое вероятно имеет смысл улучшить в более позднем выпуске), но ввод `stop` по-прежнему приводит к завершению, а допустимые числа возводятся в квадрат:

```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
Bye
```

## Обработка ошибок с помощью оператора `try`

Предыдущее решение работает, но позже в книге вы увидите, что самый универсальный способ обработки ошибок в Python предусматривает перехват и полное восстановление после них с применением оператора `try` языка Python. Мы будем исследовать этот оператор в части VII книги, а сейчас отметим, что использование `try` здесь может в итоге дать код, который многие сочтут более простым, чем предыдущая версия:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(num ** 2)
print('Bye')
```

Новая версия кода работает в точности как предыдущая, но явную проверку на предмет ошибки мы заменили кодом, который предполагает успешное выполнение преобразования, и поместили его внутрь обработчика исключений для случаев, когда это не так. Другими словами, вместо обнаружения ошибки мы просто реагируем, когда она возникает.

Оператор `try` является еще одним составным оператором и следует такому же шаблону, как `if` и `while`. Он состоит из слова `try`, за которым находится основной блок кода (действие, подлежащее выполнению), затем часть `except` с кодом обработчика исключений и часть `else`, выполняемая в случае, если никакие исключения в части `try` не генерировались. Python сначала выполняет часть `try`, после чего либо часть `except` (если исключение произошло), либо часть `else` (если исключений не было).

С точки зрения вложения операторов, поскольку слова `try`, `except` и `else` имеют отступы на том же самом уровне, они считаются частью одного оператора `try`.

Обратите внимание, что часть `else` здесь ассоциирована с `try`, а не с `if`. Как вы увидите, в Python часть `else` может появляться в операторах `if`, но также может присутствовать в операторах `try` и циклах — ее отступ сообщает, к какому оператору она относится. В данном случае оператор `try` начинается со слова `try` и охватывает код, расположенный с отступом под словом `else`, потому что `else` имеет такой же отступ, как у `try`. Оператор `if` в этом коде однострочный и заканчивается после `break`.

## Поддержка чисел с плавающей точкой

Позже в книге мы еще вернемся к оператору `try`. Пока достаточно знать, что поскольку оператор `try` можно применять для перехвата любой ошибки, он сокращает объем кода проверки на предмет ошибок, который приходится писать, и представляет собой очень универсальный подход к работе с необычными случаями. Скажем, при наличии уверенности в том, что `print` не потерпит неудачу, тогда пример мог бы стать еще короче:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

И если мы хотим поддерживать ввод чисел с плавающей точкой вместо только целых, например, то использовать `try` было бы гораздо легче, чем вручную проверять на предмет ошибок — мы могли бы просто вызвать функцию `float` и перехватить ее исключения:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(float(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

На сегодняшний день функции вроде `isfloat` для строк не предусмотрено, поэтому такой подход на основе исключений избавляет нас от необходимости анализировать весь возможный синтаксис чисел с плавающей точкой посредством явной проверки на предмет ошибок. Когда код написан таким способом, мы можем вводить более разнообразные числа, но выявление ошибок и выход по-прежнему работают, как и ранее:

```
Enter text:50
2500.0
Enter text:40.5
1640.25
Enter text:1.23E-100
1.5129e-200
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```



Здесь на месте `float` также работал бы вызов функции `eval` языка Python, который мы применяли в главах 5 и 9 для преобразования данных в строках и файлах. Тогда появилась бы возможность вводить произвольные выражения (`2 ** 100` было бы допустимым, хотя и необычным вводом, особенно с учетом того, что программа обрабатывает возраст!). Это мощная концепция, которая подвержена тем же самым проблемам с безопасностью, о которых шла речь в предшествующих главах. Если вы не можете доверять источнику строки с кодом, тогда используйте более ограничивающие инструменты преобразования наподобие `int` и `float`.

Функция `exec` языка Python, применяемая в главе 3 для выполнения кода из файла, похожа на `eval` (но предполагает, что строка является оператором, а не выражением, и не возвращает результат), а вызов `compile` предварительно компилирует часто используемые строки кода в объекты байт-кода ради достижения высокой скорости. Дополнительные сведения можно получить, запустив `help` для любого из указанных средств; как упоминалось ранее, `exec` представляет собой оператор в Python 2.X, но функцию в Python 3.X, поэтому в Python 2.X ищите ее описание в руководстве. Мы также будем применять `exec` в главе 25 для импортирования модулей, используя строку с именем (пример более динамических ролей данного инструмента).

## Вложение кода на три уровня в глубину

Давайте рассмотрим последнее изменение кода. При необходимости вложение можно продолжить; скажем, мы могли бы расширить приведенный ранее сценарий, обрабатывающий только целые числа, для перехода к одной из набора альтернатив на основе относительной величины допустимого ввода:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

В последней версии сценария добавлен оператор `if`, вложенный в часть `else` другого оператора `if`, который в свою очередь вложен в цикл `while`. Когда код является условным или повторяется как здесь, мы просто смещаем его дальше вправо. Совокупный эффект похож на предшествующие версии, но теперь для чисел, меньших 20, будет выводиться слово `low`:

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```



## Резюме

На этом краткий обзор синтаксиса операторов Python завершен. В главе были представлены общие правила для написания операторов и блоков кода. Вы узнали, что в Python мы обычно записываем по одному оператору в строке и смещаем все операторы во вложенном блоке на одно и то же расстояние вправо (отступ — часть синтаксиса Python). Кроме того, мы также взглянули на ряд исключений из этих правил, в том числе строки продолжения и однострочные проверки и циклы. Наконец, мы воплотили все рассмотренные идеи на практике в сценарии с интерактивным циклом, где продемонстрировали несколько операторов и показали синтаксис операторов в действии.

В следующей главе мы начнем более тщательно исследовать каждый базовый процедурный оператор Python. Как вы увидите, все операторы соблюдают те же самые общие правила, которые были описаны в настоящей главе.

## Проверьте свои знания: контрольные вопросы

1. Какие три элемента синтаксиса обязательны в C-подобном языке, но опущены в Python?
2. Как обычно завершается оператор в Python?
3. Как операторы во вложенном блоке обычно ассоциируются в Python?
4. Как можно было бы разместить одиночный оператор в нескольких строках?
5. Как можно было бы записать составной оператор в одной строке?
6. Есть ли веские причины набирать точку с запятой в конце оператора в Python?
7. Для чего предназначен оператор `try`?
8. Какую ошибку чаще всего допускают новички в Python?

## Проверьте свои знания: ответы

1. C-подобные языки требуют наличия круглых скобок вокруг выражений проверки в некоторых операторах, точки с запятой в конце каждого оператора и фигурных скобок вокруг вложенного блока кода.
2. Конец строки завершает оператор, находящийся в этой строке. Если в одной строке присутствует несколько операторов, то их можно завершать с помощью точек с запятой; подобным же образом, если оператор охватывает множество строк, тогда его придется завершать путем помещения внутрь пары квадратных скобок.
3. Все операторы во вложенном блоке имеют отступы с одинаковым количеством табуляций или пробелов.
4. Разместить одиночный оператор в нескольких строках можно, поместив его часть в круглые, квадратные или фигурные скобки; оператор заканчивается, когда Python встречает строку, в которой содержится закрывающая скобка из пары.
5. Тело составного оператора может быть перемещено в строку заголовка после двоеточия, но только если тело состоит из несоставных операторов.
6. Лишь тогда, когда необходимо втиснуть несколько операторов в одну строку кода. И то это работает, только если операторы являются несоставными, к тому же не одобряется, поскольку может давать в итоге код, который трудно читать.
7. Оператор `try` применяется для перехвата и восстановления после исключений (ошибок) в сценарии Python. Он обычно выступает в качестве альтернативы ручной проверки на предмет ошибок в коде.
8. Самая распространенная ошибка новичков связана с тем, что они забывают набрать двоеточие в конце строки заголовка составного оператора. Если вы начали изучать Python и пока ее не допустили, то вероятно это скоро произойдет!