

# Содержание

<b>Предисловие</b>	17
<b>Часть VI. Классы и объектно-ориентированное программирование</b>	19
<b>ГЛАВА 26. Объектно-ориентированное программирование: общая картина</b>	20
Для чего используются классы?	21
Объектно-ориентированное программирование с высоты птичьего полета	22
Поиск в иерархии наследования	23
Классы и экземпляры	25
Вызовы методов	25
Создание деревьев классов	26
Перегрузка операций	28
Объектно-ориентированное программирование – это многократное использование кода	29
Резюме	32
Проверьте свои знания: контрольные вопросы	32
Проверьте свои знания: ответы	33
<b>ГЛАВА 27. Основы написания классов</b>	34
Классы генерируют множество объектов экземпляров	34
Объекты классов обеспечивают стандартное поведение	35
Объекты экземпляров являются конкретными элементами	35
Первый пример	36
Классы настраиваются через наследование	38
Второй пример	39
Классы являются атрибутами в модулях	41
Классы могут перехватывать операции Python	42
Третий пример	43
Для чего используется перегрузка операций?	45
Простейший в мире класс Python	46
Снова о записях: классы или словари	49
Резюме	51
Проверьте свои знания: контрольные вопросы	51
Проверьте свои знания: ответы	52
<b>ГЛАВА 28. Более реалистичный пример</b>	54
Шаг 1: создание экземпляров	55
Написание кода конструкторов	55
Тестирование в ходе дела	56
Использование кода двумя способами	58
Шаг 2: добавление методов, реализующих поведение	59
Написание кода методов	61
Шаг 3: перегрузка операций	63
Реализация отображения	63

Шаг 4: настройка поведения за счет создания подклассов	65
Написание кода подклассов	66
Расширение методов: плохой способ	66
Расширение методов: хороший способ	67
Полиморфизм в действии	69
Наследование, настройка и расширение	70
Объектно-ориентированное программирование: основная идея	71
Шаг 5: настройка конструкторов	72
Объектно-ориентированное программирование проще, чем может казаться	73
Другие способы комбинирования классов	74
Шаг 6: использование инструментов интроспекции	77
Специальные атрибуты класса	78
Обобщенный инструмент отображения	79
Атрибуты экземпляра или атрибуты класса	81
Размышления относительно имен в классах инструментов	82
Финальная форма классов	83
Шаг 7 (последний): сохранение объектов в базе данных	84
Модули pickle, dbm и shelve	85
Сохранение объектов в базе данных shelve	86
Исследование хранилища shelve в интерактивной подсказке	87
Обновление объектов в хранилище shelve	89
Указания на будущее	91
Резюме	93
Проверьте свои знания: контрольные вопросы	93
Проверьте свои знания: ответы	94
<b>ГЛАВА 29. Детали реализации классов</b>	<b>96</b>
Оператор class	96
Общая форма	97
Пример	97
Методы	99
Пример метода	100
Вызов конструкторов суперклассов	101
Другие возможности вызова методов	101
Наследование	102
Построение дерева атрибутов	102
Специализации унаследованных методов	104
Методики связывания классов	104
Абстрактные суперклассы	106
Пространства имен: заключение	108
Простые имена: глобальные, если не выполнено их присваивание	109
Имена атрибутов: пространства имен объектов	109
“Дзен” пространств имен: присваивания классифицируют имена	110
Вложенные классы: снова о правиле областей видимости LEGB	112
Словари пространств имен: обзор	114
Связи между пространствами имен: инструмент подъема по дереву	117
Снова о строках документации	119
Классы или модули	120

Резюме	121
Проверьте свои знания: контрольные вопросы	121
Проверьте свои знания: ответы	122
<b>ГЛАВА 30. Перегрузка операций</b>	<b>123</b>
Основы	123
Конструкторы и выражения: <code>__init__</code> и <code>__sub__</code>	124
Распространенные методы перегрузки операций	124
Индексирование и нарезание: <code>__getitem__</code> и <code>__setitem__</code>	127
Перехват срезов	127
Нарезание и индексирование в Python 2.X	129
Но метод <code>__index__</code> в Python 3.X не имеет отношения к индексированию!	130
Итерация по индексам: <code>__getitem__</code>	130
Итерируемые объекты: <code>__iter__</code> и <code>__next__</code>	131
Итерируемые объекты, определяемые пользователем	132
Множество итераторов в одном объекте	135
Альтернативная реализация: <code>__iter__</code> плюс <code>yield</code>	138
Членство: <code>__contains__</code> , <code>__iter__</code> и <code>__getitem__</code>	142
Доступ к атрибутам: <code>__getattr__</code> и <code>__setattr__</code>	145
Ссылка на атрибуты	146
Присваивание и удаление атрибутов	147
Другие инструменты управления атрибутами	148
Эмуляция защиты атрибутов экземпляра: часть 1	149
Строковое представление: <code>__repr__</code> и <code>__str__</code>	150
Для чего используются два метода отображения?	151
Замечания по использованию отображения	152
Использование с правой стороны и на месте: <code>__radd__</code> и <code>__iadd__</code>	153
Правостороннее сложение	154
Сложение на месте	157
Выражения вызовов: <code>__call__</code>	158
Функциональные интерфейсы и код, основанный на обратных вызовах	160
Сравнения: <code>__lt__</code> , <code>__gt__</code> и другие	162
Метод <code>__cmp__</code> в Python 2.X	163
Булевские проверки: <code>__bool__</code> и <code>__len__</code>	163
Булевские методы в Python 2.X	164
Уничтожение объектов: <code>__del__</code>	166
Замечания относительно использования деструкторов	166
Резюме	167
Проверьте свои знания: контрольные вопросы	168
Проверьте свои знания: ответы	168
<b>ГЛАВА 31. Проектирование с использованием классов</b>	<b>169</b>
Python и объектно-ориентированное программирование	169
Полиморфизм означает интерфейсы, а не сигнатуры вызовов	170
Объектно-ориентированное программирование и наследование: отношения “является”	171
Объектно-ориентированное программирование и композиция: отношения “имеет”	173

Снова об обработчиках потоков данных	174
Объектно-ориентированное программирование и делегирование:	
промежуточные объекты-оболочки	178
Псевдозакрытые атрибуты классов	180
Обзор корректировки имен	180
Для чего используются псевдозакрытые атрибуты?	181
Методы являются объектами: связанные или несвязанные методы	183
Несвязанные методы являются функциями в Python 3.X	185
Связанные методы и другие вызываемые объекты	187
Классы являются объектами: обобщенные фабрики объектов	190
Для чего используются фабрики?	191
Множественное наследование: “подмешиваемые” классы	192
Реализация подмешиваемых классов отображения	193
Другие темы, связанные с проектированием	214
Резюме	214
Проверьте свои знания: контрольные вопросы	215
Проверьте свои знания: ответы	215
<b>ГЛАВА 32. Расширенные возможности классов</b>	<b>216</b>
Расширение встроенных типов	217
Расширение типов путем внедрения	217
Расширение типов путем создания подклассов	218
Модель классов “нового стиля”	220
Что нового в новом стиле?	221
Изменения в классах нового стиля	222
Процедура извлечения атрибутов для встроенных операций пропускает экземпляры	224
Изменения модели типов	229
Все классы являются производными от object	232
Изменение ромбовидного наследования	234
Дополнительные сведения о MRO: порядок распознавания методов	238
Пример: отображение атрибутов на источники наследования	241
Расширения в классах нового стиля	246
Слоты: объявления атрибутов	247
Свойства: средства доступа к атрибутам	256
Метод <code>__getattr__</code> и дескрипторы: инструменты для работы с атрибутами	259
Другие изменения и расширения классов	260
Статические методы и методы классов	261
Для чего используются специальные методы?	261
Статические методы в Python 2.X и 3.X	262
Альтернативы для статических методов	264
Использование статических методов и методов класса	265
Подсчет экземпляров с помощью статических методов	267
Подсчет экземпляров с помощью методов классов	268
Декораторы и метаклассы: часть 1	271
Основы декораторов функций	272
Первый взгляд на декораторы функций, определяемые пользователем	273
Первый взгляд на декораторы классов и метаклассы	275

Дополнительные сведения	277
Встроенная функция <code>super</code> : для лучшего или для худшего?	277
Продолжительные дебаты относительно <code>super</code>	277
Традиционная форма вызова методов суперкласса: переносимая, универсальная	279
Базовое использование встроенной функции <code>super</code> и связанные с ней компромиссы	279
Положительные стороны <code>SUPER</code> : изменения деревьев и координирование	285
Изменения классов во время выполнения и <code>super</code>	286
Кооперативная координация вызовов методов при множественном наследовании	287
Сводка по <code>super</code>	299
Затруднения, связанные с классами	300
Изменение атрибутов классов может иметь побочные эффекты	301
Модификация изменяемых атрибутов классов тоже может иметь побочные эффекты	302
Множественное наследование: порядок имеет значение	303
Области видимости в методах и классах	304
Другие затруднения, связанные с классами	305
Еще раз о KISS: чрезмерно большое количество уровней	306
Резюме	307
Проверьте свои знания: контрольные вопросы	307
Проверьте свои знания: ответы	307
Проверьте свои знания: упражнения для части VI	309
<b>Часть VII. Исключения и инструменты</b>	315
<b>ГЛАВА 33. Основы исключений</b>	316
Для чего используются исключения?	316
Роли, исполняемые исключениями	317
Исключения: краткая история	318
Стандартный обработчик исключений	318
Перехват исключений	320
Генерация исключений	321
Исключения, определяемые пользователем	321
Действия при завершении	322
Резюме	325
Проверьте свои знания: контрольные вопросы	326
Проверьте свои знания: ответы	326
<b>ГЛАВА 34. Детали обработки исключений</b>	327
Оператор <code>try/except/else</code>	327
Как работают операторы <code>try</code>	328
Конструкции оператора <code>try</code>	329
Конструкция <code>else</code> оператора <code>try</code>	332
Пример: стандартное поведение	333
Пример: перехват встроенных исключений	334

Оператор <code>try/finally</code>	335
Пример: написание кода действий при завершении с помощью <code>try/finally</code>	336
Унифицированный оператор <code>try/except/finally</code>	337
Унифицированный синтаксис оператора <code>try</code>	338
Комбинирование <code>finally</code> и <code>except</code> за счет вложения	339
Пример унифицированного оператора <code>try</code>	339
Оператор <code>raise</code>	341
Генерация исключений	342
Области видимости и переменные <code>except</code> в <code>try</code>	343
Распространение исключений с помощью <code>raise</code>	344
Сцепление исключений в Python 3.X: <code>raise from</code>	345
Оператор <code>assert</code>	347
Пример: улавливание нарушений ограничений (но не ошибок!)	348
Диспетчеры контекстов <code>with/as</code>	349
Базовое использование	350
Протокол управления контекстами	351
Множество диспетчеров контекстов в Python 3.1, 2.7 и последующих версиях	353
Резюме	355
Проверьте свои знания: контрольные вопросы	355
Проверьте свои знания: ответы	355
<b>ГЛАВА 35. Объекты исключений</b>	357
Исключения: назад в будущее	358
Строковые исключения канули в лету!	358
Исключения на основе классов	359
Реализация классов исключений	360
Для чего используются иерархии исключений?	362
Встроенные классы исключений	365
Категории встроенных исключений	366
Стандартный вывод и состояние	367
Специальное отображение при выводе	369
Специальные данные и поведение	371
Предоставление деталей исключения	371
Предоставление методов исключений	372
Резюме	373
Проверьте свои знания: контрольные вопросы	373
Проверьте свои знания: ответы	374
<b>ГЛАВА 36. Проектирование с использованием исключений</b>	375
Вложение обработчиков исключений	375
Пример: вложение в потоке управления	377
Пример: синтаксическое вложение	377
Идиомы исключений	379
Прерывание множества вложенных циклов: “безусловный переход”	379
Исключения не всегда являются ошибками	380
Функции могут сигнализировать об условиях с помощью <code>raise</code>	381
Закрытие файлов и серверных подключений	382
Отладка с помощью внешних операторов <code>try</code>	383
Выполнение внутрипроцессных тестов	383

Дополнительные сведения о функции <code>sys.exc_info</code>	384
Отображение сообщений об ошибках и трассировок	385
Советы по проектированию с использованием исключений и связанные с ними затруднения	386
Что должно быть помещено внутрь операторов <code>try</code> ?	386
Перехват слишком многого: избегайте использования пустой конструкции <code>except</code> и конструкции <code>except Exception</code>	387
Перехват чересчур малого: используйте категории на основе классов	389
Сводка по базовому языку	390
Комплект инструментов Python	390
Инструменты для разработки, ориентированные на более крупные проекты	391
Резюме	395
Проверьте свои знания: контрольные вопросы	396
Проверьте свои знания: ответы	396
Проверьте свои знания: упражнения для части VII	396
<b>Часть VIII. Более сложные темы</b>	399
<b>ГЛАВА 37. Unicode и байтовые строки</b>	400
Изменения строк в Python 3.X	401
Основы строк	402
Схемы кодирования символов	402
Хранение строк Python в памяти	405
Типы строк Python	406
Текстовые и двоичные файлы	408
Написание базовых строк	409
Строковые литералы Python 3.X	410
Строковые литералы Python 2.X	412
Преобразования строковых типов	412
Написание строк Unicode	414
Написание текста ASCII	414
Написание текста, отличающегося от ASCII	415
Кодирование и декодирование текста, отличающегося от ASCII	416
Другие схемы кодирования	417
Байтовые строковые литералы: закодированный текст	418
Преобразования между кодировками	420
Кодирование строк Unicode в Python 2.X	420
Объявления кодировок в файлах исходного кода	424
Использование объектов <code>bytes</code> в Python 3.X	425
Вызовы методов	425
Операции над последовательностями	426
Другие способы создания объектов <code>bytes</code>	427
Смешивание строковых типов	428
Использование объектов <code>bytearray</code> в Python 3.X/2.6+	428
Объекты <code>bytearray</code> в действии	429
Сводка по строковым типам Python 3.X	431
Использование текстовых и двоичных файлов	431
Основы текстовых файлов	432
Текстовый и двоичный режимы в Python 2.X и 3.X	433

Несоответствия типов и содержимого в Python 3.X	434
Использование файлов Unicode	435
Чтение и запись данных Unicode в Python 3.X	436
Обработка маркера BOM в Python 3.X	437
Файлы Unicode в Python 2.X	440
Имена файлов и потоки данных Unicode	441
Другие изменения инструментов для обработки строк в Python 3.X	442
Модуль re для сопоставления с образцом	442
Модуль struct для работы с двоичными данными	444
Модуль pickle для сериализации объектов	446
Инструменты для разбора XML	447
Резюме	451
Проверьте свои знания: контрольные вопросы	452
Проверьте свои знания: ответы	452
<b>ГЛАВА 38. Управляемые атрибуты</b>	<b>455</b>
Для чего используются управляемые атрибуты?	455
Вставка кода для запуска при доступе к атрибутам	456
Свойства	457
Основы	458
Первый пример	458
Вычисляемые атрибуты	459
Реализация свойств с помощью декораторов	460
Дескрипторы	462
Основы	462
Первый пример	465
Вычисляемые атрибуты	467
Использование информации состояния в дескрипторах	468
Связь между свойствами и дескрипторами	471
__getattr__ и __getattribute__	473
Основы	474
Первый пример	477
Вычисляемые атрибуты	478
Сравнение __getattr__ и __getattribute__	480
Сравнение методик управления	481
Перехват атрибутов для встроенных операций	484
Пример: проверка достоверности атрибутов	491
Использование свойств для проверки достоверности	492
Использование дескрипторов для проверки достоверности	494
Использование __getattr__ для проверки достоверности	498
Использование __getattribute__ для проверки достоверности	500
Резюме	501
Проверьте свои знания: контрольные вопросы	502
Проверьте свои знания: ответы	502
<b>ГЛАВА 39. Декораторы</b>	<b>504</b>
Что такое декоратор?	504
Управление вызовами и экземплярами	505
Управление функциями и классами	505



Использование и определение декораторов	506
Для чего используются декораторы?	506
Основы	508
Декораторы функций	508
Декораторы классов	512
Вложение декораторов	514
Аргументы декораторов	516
Декораторы одновременно управляют функциями и классами	517
Реализация декораторов функций	518
Отслеживание вызовов	518
Варианты предохранения состояния для декораторов	519
Грубые ошибки, связанные с классами, часть I: декорирование методов	524
Измерение времени вызовов	530
Добавление аргументов к декоратору	533
Реализация декораторов классов	536
Классы-одиночки	536
Отслеживание объектных интерфейсов	538
Грубые ошибки, связанные с классами, часть II: предохранение множества экземпляров	542
Декораторы или управляющие функции	543
Для чего используются декораторы? (Еще раз)	545
Управление функциями и классами напрямую	547
Пример: “закрытые” и “открытые” атрибуты	549
Реализация закрытых атрибутов	549
Детали реализации, часть I	551
Обобщение также для открытых объявлений	553
Детали реализации, часть II	555
Нерешенные проблемы	556
Python не поощряет контроль доступа	564
Пример: проверка допустимости аргументов функций	565
Цель	565
Базовый декоратор проверки вхождения значений в диапазон для позиционных аргументов	566
Обобщение для поддержки также ключевых аргументов и стандартных значений	568
Детали реализации	571
Нерешенные проблемы	574
Аргументы декоратора или аннотации функций	576
Другие приложения: проверка типов (если вы настаиваете!)	578
Резюме	579
Проверьте свои знания: контрольные вопросы	579
Проверьте свои знания: ответы	580
<b>ГЛАВА 40. Метаклассы</b>	590
Нужно ли иметь дело с метаклассами?	591
Повышение уровней “магии”	592
Язык привязок	593
Недостаток “вспомогательных” функций	594
Метаклассы против декораторов классов: раунд 1	596

Модель метаклассов	599
Классы являются экземплярами <code>type</code>	599
Метаклассы являются подклассами <code>type</code>	601
Протокол оператора <code>class</code>	602
Объявление метаклассов	603
Объявление в Python 3.X	603
Объявление в Python 2.X	604
Координирование метаклассов в Python 3.X и 2.X	605
Реализация метаклассов	605
Базовый метакласс	606
Настройка создания и инициализации	607
Другие методики реализации метаклассов	608
Наследование и экземпляр	613
Метакласс или суперкласс	615
Наследование: вся история	617
Методы метаклассов	623
Методы метаклассов или методы классов	624
Перегрузка операций в методах метакласса	624
Пример: добавление методов в классы	626
Ручное дополнение	626
Дополнение на основе метаклассов	628
Метаклассы против декораторов классов: раунд 2	629
Пример: применение декораторов к методам	634
Трассировка с помощью декорирования вручную	635
Трассировка с помощью метаклассов и декораторов	636
Применение любого декоратора к методам	637
Метаклассы против декораторов классов: раунд 3 (и последний)	639
Резюме	641
Проверьте свои знания: контрольные вопросы	642
Проверьте свои знания: ответы	642
<b>ГЛАВА 41. Все хорошее когда-нибудь заканчивается</b>	644
Парадокс Python	644
О “необязательных” языковых средствах	645
Против тревожных усовершенствований	646
Сложность или мощь	647
Простота или элитарность	648
Заключительные размышления	648
Куда двигаться дальше?	649
На бис: распечатайте собственный сертификат об окончании!	649
<b>Часть IX. Приложения</b>	653
<b>Приложение А. Установка и конфигурирование</b>	654
Установка интерпретатора Python	654
Присутствует ли Python на компьютере?	654
Где взять интерпретатор Python	655
Шаги установки	656

Конфигурирование интерпретатора Python	658
Переменные среды Python	658
Способы установки конфигурационных параметров	660
Аргументы командной строки интерпретатора Python	663
Командные строки запускающего модуля, появившегося в Python 3.3	666
Дополнительная помощь	667
<b>Приложение Б. Запускающий модуль Windows для Python</b>	668
Наследие Unix	668
Наследие Windows	669
Введение в запускающий модуль Windows	670
Учебное руководство по запускающему модулю Windows	672
Шаг 1: использование директив версий в файлах	672
Шаг 2: использование переключателей версий командной строки	675
Выводы: чистый выигрыш для Windows	676
<b>Приложение В. Изменения в Python и настоящая книга</b>	677
Главные отличия между Python 2.X и Python 3.X	677
Отличия Python 3.X	678
Расширения, доступные только в Python 3.X	679
Общие замечания: изменения в Python 3.X	680
Изменения в библиотеках и инструментах	681
Переход на Python 3.X	682
Изменения в Python, относящиеся к пятому изданию: Python 2.7, 3.2, 3.3	682
Изменения в Python 2.7	683
Изменения в Python 3.8	683
Изменения в Python 3.7	683
Изменения в Python 3.3	685
Изменения в Python 3.2	686
Изменения в Python, относящиеся к четвертому изданию: Python 2.6, 3.0, 3.1	686
Изменения в Python 3.1	686
Изменения в Python 3.0 и 2.6	687
Удаления в языке Python 3.0	688
Изменения в Python, относящиеся к третьему изданию: Python 2.3, 2.4, 2.5	691
Более ранние и более поздние изменения в Python	691
<b>Приложение Г. Решения упражнений, приводимых в конце частей</b>	692
Часть VI, “Классы и объектно-ориентированное программирование”	692
Часть VII, “Исключения и инструменты”	700
<b>Предметный указатель</b>	709

# ОСНОВЫ ИСКЛЮЧЕНИЙ

В этой части книги мы будем иметь дело с *исключениями*, которые являются событиями, способными изменить поток управления в программе. Исключения в Python возникают автоматически при ошибках и могут генерироваться и перехватываться вашим кодом. Они обрабатываются четырьмя операторами, рассматриваемыми в данной части, первый из которых имеет две вариации (перечисленные ниже по отдельности), а последний был необязательным расширением вплоть до версий Python 2.6 и Python 3.0.

## ***try/except***

Перехватывает и производит восстановление после исключений, инициируемых Python или вами.

## ***try/finally***

Выполняет действия по очистке независимо от того, происходили исключения или нет.

## ***raise***

Генерирует исключение вручную в коде.

## ***assert***

Генерирует исключение условно в коде.

## ***with/as***

Реализует диспетчеры контекстов в Python 2.6, 3.0 и последующих версиях (необязательные в Python 2.5).

Рассмотрение этой темы перенесено ближе к концу книги, т.к. для реализации самих исключений вам необходимо было изучить классы. Однако за небольшими исключениями (получился каламбур) вы обнаружите, что обработка исключений в коде на Python проста из-за ее интеграции в сам язык как еще одного высокоуровневого инструмента.

## Для чего используются исключения?

Вкратце исключения позволяют нам перескакивать через произвольно большие порции кода программы. Возьмем обсуждаемый ранее в книге гипотетический робот по приготовлению пиццы. Предположим, что мы занялись идеей всерьез и построили

такую машину. Чтобы приготовить пищу, нашему кулинарному автомату понадобится выполнить план, который мы реализуем в виде программы на Python: она примет заказ, замесит тесто, добавит начинки, испечет основу и т.д.

Теперь представим, что во время шага “испечет основу” что-то пошло совершенно не так. Возможно, сломался духовой шкаф или робот неправильно рассчитал расстояние для перемещения к нему и самопроизвольно воспламенился. Очевидно, мы хотим иметь возможность перехода на код, который быстро обработает такие состояния. Поскольку в необычных случаях подобного рода у нас нет никакой надежды на то, что задача приготовления пищи будет доведена до конца, мы также могли бы целиком отказаться от плана.

Именно это и позволяют нам делать исключения: можно за один шаг перейти к обработчику исключений, отменяя все вызовы функций, которые начались до того, как был совершен вход в данный обработчик. Затем код в обработчике исключений надлежащим образомотреагирует на сгенерированное исключение (скажем, позвонив в противопожарную службу!).

Об исключении можно думать как о своеобразном структурированном “безусловном переходе”. *Обработчик исключений* (оператор `try`) оставляет маркер и выполняет некоторый код. Где-то намного дальше в программе генерируется исключение, заставляющее интерпретатор Python перейти обратно на этот маркер и прекратить выполнение любых активных функций, которые были вызваны после оставления маркера. Такой протокол обеспечивает согласованный способ реагирования на необычные события. Кроме того, поскольку интерпретатор Python переходит к оператору обработчика незамедлительно, ваш код становится проще — как правило, исчезает необходимость проверять коды состояния после каждого вызова функции, которая способна потерпеть неудачу.

## Роли, исполняемые исключениями

В программах на Python исключения обычно применяются для разнообразных целей. Ниже перечислены самые распространенные роли, которые они исполняют.

### *Обработка ошибок*

Интерпретатор Python генерирует исключения всякий раз, когда обнаруживает ошибки в программах во время выполнения. Вы можете перехватывать и реагировать на ошибки в своем коде либо игнорировать инициированные исключения. Если ошибка игнорируется, тогда активизируется стандартная линия поведения обработки исключений Python: она останавливает программу и выводит сообщение об ошибке. Если вас не устраивает такое стандартное поведение, то нужно предусмотреть оператор `try` для перехвата и восстановления после исключения — при обнаружении ошибки интерпретатор Python будет переходить на ваш обработчик `try` и программа возобновит выполнение после `try`.

### *Уведомление о событиях*

Исключения можно также использовать для оповещения о допустимых условиях, не заставляя вас передавать результирующие флаги внутри программы или явно их проверять. Например, процедура поиска могла бы генерировать исключение в случае неудачи, а не возвращать целочисленный результирующий код — и надеяться на то, что код никогда не окажется допустимым результатом!

## Обработка особых случаев

Иногда условие может возникать настолько редко, что оправдать запутанность кода для его обработки в многочисленных местах довольно-таки трудно. Вы часто можете устранить код для особых случаев за счет обработки необычных ситуаций в обработчиках исключений на более высоких уровнях программы. Оператор `assert` может аналогично применяться для проверки того, что условия соответствуют ожидаемым на стадии разработки.

## Действия при завершении

Как будет показано, оператор `try/finally` дает вам возможность гарантировать, что обязательные операции времени закрытия будут выполнены независимо от наличия или отсутствия исключений в ваших программах. Более новый оператор `with` предлагает в этом отношении альтернативу для объектов, которые его поддерживают.

## Редкие потоки управления

Наконец, поскольку исключения являются разновидностью высокоуровневого и структурированного “безусловного перехода”, вы можете их использовать в качестве основы для реализации экзотических потоков управления. Скажем, хотя в языке явно не поддерживается возврат к предыдущему состоянию, вы можете реализовать его на Python с применением исключений и небольшого объема вспомогательной логики для раскручивания присваиваний<sup>1</sup>. В Python не существует оператора “безусловного перехода” (к счастью!), но исключения временами способны исполнять похожие роли; например, `raise` может использоваться для выхода из множества циклов.

Некоторые из таких ролей мы кратко рассматривали ранее, и будем исследовать типичные сценарии применения исключений позже в этой части книги. А пока давайте начнем с того, что взглянем на инструменты обработки исключений Python.

# Исключения: краткая история

По сравнению с рядом других тем, которые встречаются в книге, исключения представляют собой довольно легковесный инструмент в Python. Из-за их простоты мы перейдем прямо к написанию кода.

## Стандартный обработчик исключений

Допустим, мы написали следующую функцию:

```
>>> def fetcher(obj, index):  
    return obj[index]
```

---

<sup>1</sup> Однако подлинный возврат к предыдущему состоянию не является частью языка Python. Возврат к предыдущему состоянию перед переходом отменяет все вычисления, но исключения Python этого не делают: переменные, которым присваивались значения между моментом входа в оператор `try` и моментом генерации исключения, не переустанавливаются в свои предыдущие значения. Даже генераторные функции и выражения, обсуждаемые в главе 20 первого тома, не делают полный возврат к предыдущему состоянию — они реагируют на запросы `next (G)` просто восстановлением состояния и возобновлением выполнения. Дополнительные сведения о возврате к предыдущему состоянию ищите в книгах, посвященных искусственному интеллекту или языкам программирования Prolog либо Icon.

В этой функции нет ничего особенного — она всего лишь индексирует объект с использованием переданного индекса. При нормальной работе она возвращает результат по допустимому индексу:

```
>>> x = 'spam'
>>> fetcher(x, 3)           # Подобно x[3]
'm'
```

Тем не менее, если мы запросим у функции `fetcher` индексирование за концом строки, тогда возникнет исключение, как только функция попытается выполнить `obj[index]`. Интерпретатор Python обнаруживает индексирование последовательностей, выходящее за допустимые пределы, и сообщает о нем *генерацией* встроенного исключения `IndexError`:

```
>>> fetcher(x, 4)           # Стандартный обработчик – интерфейс оболочки
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
Трассировка (самый последний вызов указан последним):
  файл <stdin>, строка 1, в <модуль>
  файл <stdin>, строка 2, в fetcher
Ошибка индекса: индекс в строке выходит за допустимые пределы
```

Из-за того, что наш код явно не перехватывает такое исключение, оно просачивается на верхний уровень программы и приводит к вызову *стандартного обработчика исключений*, который просто выводит стандартное сообщение об ошибке. К этому месту в книге вы, наверное, уже получили свою долю стандартных сообщений об ошибках. Они содержат сгенерированное исключение и *трассировку стека* — список всех строк и функций, которые были активными на момент возникновения исключения.

Текст сообщения об ошибке здесь был выведен версией Python 3.7; он может слегка варьироваться в зависимости от выпуска и даже от интерактивной оболочки, так что вы не должны полагаться на его точную форму — ни в книге, ни в своем коде. При интерактивном написании кода в базовом интерфейсе оболочки именем файла будет просто `<stdin>`, что обозначает стандартный входной поток.

При работе в интерактивной оболочке с графическим пользовательским интерфейсом IDLE именем файла является `<pyshell>` и отображаются также строки исходного кода. В любом случае номера строк в файле не особо содержательны, когда файла нет (позже в текущей части книги вы увидите более интересные сообщения об ошибках):

```
>>> fetcher(x, 4)           # Стандартный обработчик – интерфейс IDLE
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    fetcher(x, 4)
  File "<pyshell#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
Трассировка (самый последний вызов указан последним):
  файл <pyshell#5>, строка 1, в <модуль>
  файл <pyshell#3>, строка 2, в fetcher
    return obj[index]
Ошибка индекса: индекс в строке выходит за допустимые пределы
```

В более реалистичной программе, запущенной вне интерактивной оболочки, после вывода сообщения об ошибке стандартный обработчик верхнего уровня также немедленно *прекращает работу* программы. Такой курс действий имеет смысл для простых сценариев; ошибки часто должны быть фатальными, и лучшее, что вы можете предпринять, когда они возникают — изучить стандартное сообщение об ошибке.

## Перехват исключений

Однако временами это не то, что вас интересует. Скажем, серверные программы обычно должны оставаться активными даже после возникновения внутренних ошибок. Если вы не хотите иметь дело со стандартным поведением исключений, тогда поместите вызов внутрь оператора `try`, чтобы самостоятельно перехватывать исключения:

```
>>> try:
...     fetcher(x, 4)
... except IndexError:           # Перехват и восстановление
...     print('got exception')   # Получено исключение
...
got exception
>>>
```

Теперь интерпретатор Python автоматически переходит на ваш *обработчик* (блок ниже конструкции `except`, в которой указано генерируемое исключение), когда на стадии выполнения блока `try` инициируется исключение. Результатом оказывается вкладывание вложенного блока кода внутрь обработчика ошибок, который перехватывает исключения данного блока.

При интерактивном взаимодействии вроде показанного далее после выполнения конструкции `except` мы возвращаемся обратно в подсказку Python. В более реалистичной программе операторы `try` не только перехватывают исключения, но также осуществляют *восстановление* после них:

```
>>> def catcher():
...     try:
...         fetcher(x, 4)
...     except IndexError:
...         print('got exception') # Получено исключение
...         print('continuing')   # Продолжение
>>> catcher()
got exception
continuing
>>>
```

На этот раз после перехвата и обработки исключения программа возобновляет выполнение ниже полного оператора `try`, который его перехватил — вот почему отображается сообщение `continuing`. Мы не видим стандартное сообщение об ошибке, а программа продолжает нормально двигаться своим путем.

Обратите внимание, что в Python отсутствует способ *возвратиться обратно* к коду, который сгенерировал исключение (конечно, не считая повторного запуска кода, достигнувшего данной точки). Как только вы перехватили исключение, поток управления продолжается после полного оператора `try`, перехватившего исключение, но не после оператора, его инициировавшего. На самом деле Python очищает память от любых функций, которые завершили работу в результате возникновения исключения, подобных функции `fetcher` в нашем примере; они не возобновляемы. Оператор `try` перехватывает исключения и является тем местом, где программа возобновляет выполнение.





*Замечание по представлению.* В этой части для ряда операторов `try` верхнего уровня снова указываются приглашения . . . интерактивной подсказки, т.к. их код не будет работать в случае вырезания и вставки, если только он не вложен в функцию или класс (excerpt и другие строки должны быть выровнены с `try` и не иметь добавочных предваряющих пробелов, необходимых для иллюстрации структуры отступов). Для нормального выполнения просто набирайте или вставляйте операторы с приглашениями . . . по одной строке за раз.

## Генерация исключений

До сих пор мы позволяли интерпретатору Python генерировать исключения, совершая ошибки (преднамеренно!), но наши сценарии тоже могут инициировать исключения, т.е. исключения могут генерироваться Python или вашей программой и перехватываться или нет. Чтобы инициировать исключение вручную, просто запустите оператор `raise`. Генерируемые пользователем исключения перехватываются тем же способом, что и исключения, которые генерирует интерпретатор Python. Следующий код нельзя считать самым полезным кодом, когда-либо написанным на Python, но он важен тем, что инициирует встроенное исключение `IndexError`:

```
>>> try:
...     raise IndexError                # Генерация исключения вручную
... except IndexError:
...     print('got exception')        # Получено исключение
...
got exception
```

Как обычно, если генерируемые пользователем исключения не перехватываются, то они распространяются вплоть до стандартного обработчика исключений и прекращают работу программы с выводом стандартного сообщения об ошибке:

```
>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

Трассировка (самый последний вызов указан последним):
  файл <stdin>, строка 1, в <модуль>
Ошибка индекса
```

Как вы увидите в следующей главе, оператор `assert` тоже может применяться для генерации исключений — он представляет собой условный оператор `raise`, используемый главным образом при отладке на стадии разработки:

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nobody expects the Spanish Inquisition!

Трассировка (самый последний вызов указан последним):
  файл <stdin>, строка 1, в <модуль>
Ошибка утверждения: Никто не ждёт испанскую инквизицию!
```

## Исключения, определяемые пользователем

Представленный в предыдущем разделе оператор `raise` генерировал *встроенное* исключение, определенное во встроенной области видимости Python. Как вы узнаете позже в этой части книги, можно также самостоятельно определять новые исключе-

ния, специфичные для ваших программ. Определяемые пользователем исключения реализуются с помощью *классов*, унаследованных от встроенного класса исключения — обычно класса по имени `Exception`:

```
>>> class AlreadyGotOne(Exception): pass      # Исключение, определяемое
                                           # пользователем

>>> def grail():
    raise AlreadyGotOne()                   # Генерирует экземпляр

>>> try:
...     grail()
... except AlreadyGotOne:                   # Перехват по имени класса
...     print('got exception')              # Получено исключение
...
got exception
>>>
```

В следующей главе будет показано, что конструкция `as` оператора `except` может предоставлять доступ к самому объекту исключения. Исключения на основе классов позволяют сценариям формировать категории исключений, которые способны наследовать поведение, а также иметь присоединенную информацию о состоянии и методах. Вдобавок они могут настраивать текст своих сообщений об ошибках, отображаемый в ситуации, когда не был совершен перехват:

```
>>> class Career(Exception):
    def __str__(self): return 'So I became a waiter...'

>>> raise Career()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  __main__.Career: So I became a waiter...

Трассировка (самый последний вызов указан последним):
  файл <stdin>, строка 1, в <модуль>
  __main__.Career: Так я стал официантом...
>>>
```

## Действия при завершении

Наконец, операторы `try` могут содержать слово `finally`, т.е. иметь в своем составе блоки `finally`. Они выглядят похожими на обработчики `except` для исключений, но комбинация `try/finally` указывает действия при завершении, которые всегда выполняются “на выходе” независимо от того, происходили исключение в блоке `try` или нет:

```
>>> try:
...     fetcher(x, 3)
... finally:
...     print('after fetch')                # Действия при завершении
...                                         # После извлечения
...
'm'
after fetch
>>>
```

Если блок `try` завершается без исключения, то блок `finally` выполнится и программа возобновит работу после полного оператора `try`. В данном случае наличие оператора `try` кажется слегка нелепым — мы могли бы просто набрать `print` сразу после вызова функции и вообще избавиться от `try`:

```
fetcher(x, 3)
print('after fetch')
```

Тем не менее, здесь присутствует проблема: если вызов функции сгенерирует исключение, тогда поток управления никогда не доберется до `print`. Комбинация `try/finally` позволяет избежать этой ловушки — когда в блоке `try` все же возникает исключение, блоки `finally` выполняются во время раскручивания стека программы:

```
>>> def after():
    try:
        fetcher(x, 4)
    finally:
        print('after fetch')      # После извлечения
        print('after try?')      # После try?

>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
Трассировка (самый последний вызов указан последним):
  файл <stdin>, строка 1, в <модуль>
  файл <stdin>, строка 3, в after
  файл <stdin>, строка 2, в fetcher
Ошибка индекса: индекс в строке выходит за допустимые пределы
>>>
```

Здесь мы не получаем сообщение `after try?`, потому что поток управления не возобновляется после блока `try/finally`, когда возникает исключение. Взамен интерпретатор Python переходит обратно к выполнению действия `finally` и затем *распространяет* исключение вверх к предыдущему обработчику (в этой ситуации к стандартному разработчику на верхнем уровне). Если мы изменим вызов функции `fetcher` так, чтобы не инициировать исключение, то код `finally` по-прежнему выполнится, но программа продолжит выполнения после `try`:

```
>>> def after():
    try:
        fetcher(x, 3)
    finally:
        print('after fetch')
        print('after try?')

>>> after()
after fetch
after try?
>>>
```

На практике комбинации `try/except` удобны для перехвата и восстановления после исключений, а комбинации `try/finally` оказываются полезными, когда требуется гарантия того, что действия при завершении будут запускаться независимо от любых исключений, которые могут возникать в коде блока `try`. Например, вы можете применять `try/except` для перехвата ошибок, инициируемых кодом, который импортируется из сторонней библиотеки, и `try/finally` для обеспечения того, что обращения к функциям закрытия файлов или подключений к серверу всегда выполняются. Несколько практических примеров такого рода приводятся позже в текущей части книги.

Хотя конструкции `except` и `finally` служат концептуально отличающимся целям, начиная с Python 2.5, мы можем смешивать их в одном операторе `try` — конструкция `finally` выполняется при выходе независимо от того, генерировалось ли исключение, и было ли оно перехвачено конструкцией `except`.

Как выяснится в следующей главе, линейки Python 2.X и Python 3.X предлагают альтернативу `try/finally` в случае использования некоторых видов объектов. Оператор `with/as` запускает логику *управления контекстом* объекта, чтобы гарантировать выполнение действий при завершении безотносительно к любым исключениям в его вложенном блоке:

```
>>> with open('lumberjack.txt', 'w') as file:      # Всегда при выходе
                                             #  закрывать файл
    file.write('The larch!\n')
```

Несмотря на то что такой вариант требует меньше строк кода, он применим только при обработке определенных объектных типов, поэтому `try/finally` является более универсальной структурой для завершения и часто проще, чем реализации класса в случаях, где `with` еще не поддерживается. С другой стороны, `with/as` может выполнять также действия начального запуска и поддерживает определяемый пользователем код управления контекстами с доступом к полному комплекту инструментов ООП на Python.

---

### Что потребует внимания: проверка на предмет ошибок

---

Один из способов посмотреть, насколько полезны исключения, предусматривает сравнение кодовых стилей в Python и языках без исключений. Скажем, если вы хотите написать надежную программу на языке C, то обычно должны проверять возвращаемые значения или коды состояния после каждой операции, способной сбиться с пути, и распространять результаты проверок во время выполнения программы:

```
doStuff()
{
    # Программа на C
    if (doFirstThing() == ERROR) # Выявлять ошибки повсеместно,
        return ERROR;          # даже если они здесь не обрабатываются
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

На самом деле реалистичные программы на C часто содержат столько же кода, предназначенного для обнаружения ошибок, сколько и кода для выполнения фактической работы. Но в Python вам не придется быть до такой степени методичными (вплоть до паранойи!). Вы можете взамен помещать произвольно крупные фрагменты программы внутрь обработчиков исключений и просто писать код, делающий действительную работу, предполагая о том, что обычно все будет хорошо:

```

def doStuff():          # Код на Python
    doFirstThing()     # Мы не обязаны здесь заботиться об исключениях,
    doNextThing()      # поэтому нет необходимости и выявлять их
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()      # Тут нас интересуют результаты, так что
    except:             # это единственное место, где требуется проверка
        badEnding()
    else:
        goodEnding()

```

Так как при возникновении исключения управление немедленно передается обработчику, нет нужды снабжать весь код защитой от ошибок, к тому же отсутствуют добавочные накладные расходы в плане производительности, связанные с выполнением всех проверок. Кроме того, поскольку интерпретатор Python выявляет ошибки автоматически, в первую очередь вашему коду часто нет необходимости вообще осуществлять проверки на предмет ошибок. В итоге исключения позволяют почти совершенно игнорировать необычные случаи и избегать написания кода проверки на предмет ошибок, который способен отвлечь от подлинных целей программы.

## Резюме

Итак, большая часть истории об исключениях была изложена; исключения — действительно простой инструмент.

Подводя итоги, можно сказать, что исключения Python являются высокоуровневым механизмом управления потоком выполнения. Они могут генерироваться интерпретатором Python либо вашими программами. В обоих случаях исключения допускается игнорировать (для выдачи стандартного сообщения об ошибке) или перехватывать посредством операторов `try` (с целью обработки в вашем коде). Оператор `try` имеет два логических формата, которые начиная с версии Python 2.5, можно объединять — один обрабатывает исключения, а другой выполняет код финализации независимо от того, возникло исключение или нет. Операторы `raise` и `assert` инициируют исключение по требованию — как встроенные, так и новые исключения, определяемые с помощью классов. Оператор `with/as` представляет собой альтернативный способ гарантирования того, что действия при завершении будут выполнены для объектов, которые их поддерживают.

В остатке этой части книги мы рассмотрим ряд деталей о задействованных операторах, исследуем другие виды конструкций, которые могут появляться под `try`, и обсудим объекты исключений, основанные на классах. В следующей главе мы начнем с того, что пристальнее взглянем на введенные здесь операторы. Однако прежде чем двигаться дальше, ответьте на несколько контрольных вопросов.

## Проверьте свои знания: контрольные вопросы

1. Назовите три случая, для обработки которых хорошо подходят исключения.
2. Что произойдет с исключением, если вы не предпримете ничего специального для его обработки?
3. Как сценарий может восстанавливаться после исключения?
4. Назовите два способа генерации исключений в сценарии.
5. Назовите два способа указания действий, подлежащих выполнению на стадии завершения вне зависимости от того, возникало исключение или нет.

## Проверьте свои знания: ответы

1. Обработка исключений полезна для обработки ошибок, выполнения действий при завершении и уведомления о событиях. Вдобавок она упрощает поддержку особых случаев и может использоваться для реализации альтернативных потоков управления как что-то вроде структурированной операции “безусловного перехода”. В целом обработка исключений также сокращает объем кода проверки на предмет ошибок, который может требоваться в программе – из-за того, что все ошибки попадают в обработчики, исчезает необходимость в проверке исхода каждой операции.
2. Любое неперехваченное исключение, в конце концов, просачивается в стандартный обработчик исключений, который Python предоставляет на верхнем уровне программы. Этот обработчик выводит легко узнаваемое сообщение об ошибке и прекращает работу программы.
3. Если вы не хотите, чтобы выводилось стандартное сообщение, а работа программы прекращалась, тогда можете предусмотреть операторы `try/except` для перехвата и восстановления после исключений, которые генерируются внутри их вложенных блоков кода. После того, как исключение перехвачено, оно заканчивается, и программа продолжает выполнение после `try`.
4. Операторы `raise` и `assert` можно применять для генерации исключения в точности, как если бы оно инициировалось самим интерпретатором Python. В принципе исключение можно также сгенерировать, допустив программную ошибку, но обычно это не является прямой целью!
5. Оператор `try/finally` может использоваться для обеспечения того, что действия будут выполнены после выхода из блока кода, невзирая на то, было сгенерировано исключение в блоке или нет. Оператор `with/as` может также применяться для того, чтобы гарантировать выполнение действий при завершении, но только при обработке объектных типов, которые это поддерживают.