14 Исключения

Эта глава охватывает следующие темы:

- ✓ Объяснение исключений
- ✓ Обработка исключений в Python
- ✓ Использование ключевого слова `with`

В этой главе рассматриваются исключения — языковой механизм, предназначенный специально для обработки аномальных ситуаций во время выполнения программы. Исключения чаще всего используются для обработки ошибок, возникающих в ходе выполнения программ, но они также находят эффективное применение для многих других целей. Python предоставляет набор исключений для многих стандартных ситуаций, а пользователи могут определять собственные исключения для своих целей.

Концепция исключений как механизма обработки ошибок существует достаточно давно. C и Perl, самые популярные языки системного и сценарного программирования, не предоставляют средств обработки исключений, и даже программисты, работающие на таких языках с поддержкой исключений, как C++, нередко не знакомы с ними. Для понимания этой главы читателю не обязательно знать, как работают исключения; здесь приводятся подробные объяснения.

14.1. Знакомство с исключениями

В этом разделе вы узнаете об исключениях и их использовании. Если вы уже знакомы с исключениями, переходите сразу к разделу «Исключения в Python» (раздел 14.2).

14.1.1. Философия ошибок и обработки исключений

Любая программа может столкнуться с ошибками в ходе выполнения. Чтобы продемонстрировать работу с исключениями, мы рассмотрим систему форматирования

текста, которая записывает файлы на диск, а следовательно, может столкнуться с нехваткой места на диске до того, как будут записаны все данные. Есть несколько разных подходов к решению этой проблемы.

Решение 1: оставить проблему без решения

Простейшее решение проблемы дискового пространства — считать, что для любых операций записи в файлы всегда хватает дискового пространства, поэтому беспокоиться не о чем. К сожалению, этот вариант чаще всего встречается на практике. Для маленьких программ, работающих с небольшими объемами данных, он обычно приемлем, но для критических программ он совершенно неудовлетворителен.

Решение 2: все функции возвращают код успеха/неудачи

На следующем уровне обработки ошибок разработчик признает, что ошибки возможны, и определяет методологию их обнаружения и обработки с использованием стандартных языковых механизмов. Это можно делать многими разными способами, но в типичном варианте каждая функция или процедура возвращает код состояния, который показывает, успешно ли был выполнен вызов функции или процедуры. Нормальные результаты могут возвращаться в параметрах, передаваемых по ссылке.

Посмотрим, как это решение может работать в гипотетической программе форматирования текста. Допустим, программа вызывает одну высокоуровневую функцию `save_to_file` для сохранения текущего документа в файле. Эта функция вызывает подфункции для сохранения в файле разных частей документа: `save_text_to_file` для сохранения текста, `save_prefs_to_file` для сохранения пользовательских настроек этого документа, `save_formats_to_file` для сохранения форматов, определяемых пользователем, и т. д. Все эти подфункции могут, в свою очередь, вызывать другие подфункции, сохраняющие меньшие части файла. На самом нижнем уровне располагаются встроенные системные функции, которые выводят в файл примитивные данные и сообщают об успехе или неудаче операций записи в файл.

Код обработки ошибок можно включить в каждую функцию, в которой может произойти ошибка дискового пространства, но вряд ли это имеет смысл. Единственное, что может сделать обработчик ошибки, — вывести диалоговое окно, которое уведомляет пользователя о нехватке места, предлагает удалить какие-нибудь файлы и повторить попытку. Нет смысла дублировать этот код повсюду, где осуществляется запись на диск. Вместо этого следует поместить один блок кода обработки ошибок в основную функцию записи на диск: `save_to_file`.

К сожалению, чтобы функция `save_to_file` могла определить, когда вызывать этот код обработки ошибки, каждая вызываемая ею функция, записывающая данные на диск, сама должна проверять свободное место на диске и возвращать код состояния, обозначающий успех или неудачу операции. Кроме того, функция `save_to_file` должна явно проверять каждый вызов функции записи, хотя ее не интересует,

в какой именно функции произошел сбой. В синтаксисе, сходном с C, код выглядит примерно так:

```
const ERROR = 1;
const OK = 0;
int save_to_file(filename) {
    int status;
    status = save_prefs_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    status = save_text_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    status = save_formats_to_file(filename);
    if (status == ERROR) {
        ...обработка ошибки...
    }
    .
    .
    .
}
int save_text_to_file(filename) {
    int status;
    status = ...низкоуровневый вызов для записи размера текста...
    if (status == ERROR) {
        return(ERROR);
    }
    status = ...низкоуровневый вызов для записи текстовых данных...
    if (status == ERROR) {
        return(ERROR);
    }
    .
    .
    .
}
```

То же относится к `save_prefs_to_file`, `save_formats_to_file` и всем остальным функциям, которые либо записывают данные в `filename` напрямую, либо (тем или иным способом) вызывают функции, которые записывают в `filename`.

При такой методологии код обнаружения и обработки ошибок может занять существенную часть программы, потому что каждая функция и процедура, содержащая вызовы, которые могут привести к ошибке, должна содержать код проверки ошибок. Часто у программиста не хватает времени или сил, чтобы реализовать полную обработку ошибок; такие программы получаются ненадежными и в них часто происходят сбои.

Решение 3: механизм исключений

Очевидно, что большая часть кода проверки ошибок в программах предыдущего типа в основном повторяется: код проверяет ошибки при каждой попытке записи

в файл и при обнаружении ошибки передает сообщение вызывающей процедуре. Ошибки дискового пространства обрабатываются только в одном месте: в высокоуровневой функции save_to_file. Другими словами, большая часть обработки ошибок составляет служебный код, который связывает место возникновения ошибки с местом ее обработки. На самом деле хотелось бы избавиться от всего лишнего и написать код следующего вида:

```
def save_to_file(filename)
    попытаться выполнить следующий блок
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    если на диске кончится свободное пространство во время
        выполнения предыдущего блока
        ...обработать ошибку...

def save_text_to_file(filename)
    ...низкоуровневый вызов для записи размера текста...
    ...низкоуровневый вызов для записи текстовых данных...
    .
    .
    .
```

Код обработки ошибок полностью отделен от низкоуровневых функций; ошибка (если она происходит) генерируется встроенными функциями записи файлов и распространяется прямо в функцию save_to_file, где (как предполагается) ею займется код обработки ошибок. Хотя такой код невозможно написать на C, языки с поддержкой исключений реализуют именно такое поведение, и конечно, Python принадлежит к их числу. Исключения позволяют писать более элегантный код и лучше обрабатывать аномальные ситуации.

14.1.2. Более формальное определение исключений

Генерирование исключения называется *выдачей,* или *инициированием*, исключения. В предыдущем примере все исключения инициируются функциями записи на диск, однако инициировать исключения могут любые другие функции, и даже ваш собственный код. В нашем примере исключение инициируется низкоуровневыми функциями записи на диск (не показанными в листинге), если на диске закончится свободное пространство.

Реакция на исключение называется *перехватом* исключения, а код, обрабатывающий исключение, называется *кодом обработки исключения*, или просто *обработчиком исключения*. В приведенном выше примере строка если на диске... перехватывает исключение записи на диск, а код, заменяющий строку ...обработать ошибку..., станет обработчиком исключений записи на диск (нехватки дискового пространства). Также в программе могут быть обработчики для других типов

исключений, и даже другие обработчики для исключений того же типа (но в другом месте кода).

14.1.3. Обработка разных типов исключений

В зависимости от того, какое событие породило исключение, программа может выполнять разные действия. Исключение, инициируемое при нехватке свободного места на диске, должно обрабатываться совсем не так, как исключение, инициируемое при нехватке памяти, а оба этих исключения не имеют ничего общего с исключением, возникающим по ошибке деления на ноль. В одном из вариантов обработки разных типов исключений выполняется глобальная регистрация сообщения об ошибке, обозначающего причину исключения, а все обработчики исключений анализируют это сообщение об ошибке и предпринимают соответствующие действия. На практике другой метод оказался намного более гибким.

Вместо того чтобы определять один тип исключения, Python, как и многие современные языки с поддержкой исключений, определяет разные типы исключений для разных возникающих проблем. В зависимости от произошедшего события могут инициироваться разные типы исключений. Вдобавок коду, перехватывающему исключения, можно приказать перехватывать только исключения определенных типов. Кстати, эта возможность используется в псевдокоде из решения 3 `...если на диске закончится свободное пространство...`. Такой псевдокод указывает, что этот конкретный код обработки ошибок интересуется только исключениями дискового пространства. Другие типы исключений не будут перехватываться данным кодом обработки исключений. Например, они могут перехватываться обработчиком, который обрабатывает числовые исключения, или (если такой обработчик не существует) произойдет аварийное завершение программы с ошибкой.

14.2. Исключения в Python

В оставшейся части этой главы речь пойдет о механизмах обработки исключений, встроенных в Python. Весь механизм исключений Python строится на основе объектно-ориентированной парадигмы, которой он обязан своей гибкостью и расширяемостью. Впрочем, если вы не знакомы с объектно-ориентированным программированием (ООП), вы все равно сможете пользоваться исключениями.

Исключение представляет собой объект, автоматически генерируемый в функциях Python командой `raise`. После того как объект будет сгенерирован, команда `raise` изменяет нормальную последовательность выполнения программы Python. Вместо того чтобы продолжать выполнение со следующей команды после `raise` (или другой команды, породившей исключение), в текущей цепочке вызовов ищется обработчик, способный обработать сгенерированное исключение. Если такой обработчик будет найден, он вызывается и может обратиться к объекту исключения за дополнительной информацией. Если подходящий обработчик не будет найден, то программа аварийно завершается с сообщением об ошибке.

ПРОЩЕ ПРОСИТЬ ПРОЩЕНИЯ, ЧЕМ РАЗРЕШЕНИЯ

Подход к обработке ошибок в Python в целом отличается от подхода в таких языках, как, скажем, Java. Эти языки по возможности стараются выявить как можно больше возможных ошибок заранее, поскольку обработка исключений после их возникновения может обойтись достаточно дорого. Такой стиль описан в первой части этой главы; иногда он обозначается сокращением LBYL (Look Before You Leap, то есть «Смотри, прежде чем прыгать»).

С другой стороны, Python скорее полагается на то, что исключения будут обработаны после их возникновения. И хотя такой подход может показаться рискованным, при разумном использовании исключений код получается менее громоздким и лучше читается, а ошибки обрабатываются только в случае их возникновения. Подход Python к обработке ошибок часто описывается сокращением EAFP (Easier to Ask Forgiveness than Permission, то есть «Проще просить прощения, чем разрешения»).

14.2.1. Типы исключений Python

Программа может генерировать разные типы исключений в зависимости от непосредственной причины ошибки или возникшей аномальной ситуации. В Python 3.6 поддерживаются следующие типы исключений:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError
            OverflowError
           ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNotFoundError
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError
                ConnectionAbortedError
                ConnectionRefusedError
                ConnectionResetError
            FileExistsError
            FileNotFoundError
```

```
        InterruptedError
        IsADirectoryError
        NotADirectoryError
        PermissionError
        ProcessLookupError
        TimeoutError
    ReferenceError
    RuntimeError
        NotImplementedError
        RecursionError
    SyntaxError
        IndentationError
            TabError
    SystemError
    TypeError
    ValueError
        UnicodeError
            UnicodeDecodeError
            UnicodeEncodeError
            UnicodeTranslateError
    Warning
        DeprecationWarning
        PendingDeprecationWarning
        RuntimeWarning
        SyntaxWarning
        UserWarning
        FutureWarning
        ImportWarning
        UnicodeWarning
        BytesWarningException
        ResourceWarning
```

Набор исключений Python имеет иерархическую структуру, на что указывают отступы в списке исключений. Как упоминалось в предыдущей главе, алфавитный список можно получить из модуля `__builtins__`.

Каждый тип исключения представляет собой класс Python, наследующий от родительского типа исключения. Но если вы еще не знакомы с ООП, не беспокойтесь. Скажем, исключение `IndexError` также является `LookupError` (за счет наследования), `Exception` и `BaseException`.

Такая иерархия была создана намеренно: большинство исключений наследует от `Exception`, и все исключения, определяемые пользователем, должны субклассировать `Exception`, а не `BaseException`. Дело в том, что код вида

```
try:
    # Что-то сделать
except Exception:
    # Обработать исключения
```

позволит прервать код в блоке `try` клавишами Ctrl+C без активации кода обработки ошибок, потому что исключение `KeyboardInterrupt` *не* является субклассом `Exception`.