Содержание

ГЛАВА 33

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 пиццу, 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 пиццы 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 образом отреагирует 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[1]. 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]
```

[1] 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>
    fetcher(x, 4)
  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 (`except` 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 методы. 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:                                 # Действия при завершении
...     print('after fetch')                 # 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, бо́льшая 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, возникало 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.