**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 5**

***Часть 7. Исключения и инструменты.*** *стр. 917*

**Глава 32. Основы исключений.** стр. 919

Исключения, по сути, явл-ся событиями, способными изменить ход выполнения программы. Исключения в Питон возбуждаются автоматически, когда код допускает ошибку, а также могут возбуждаться и перехватываться самими кодом.

Обрабатываются исключения четырьмя инструкциями. Эти инструкции мы и изучим. Первая инструкция имеет два вида ,а последняя явл-лась доп. расширением до выхода 2.6 и 3.0:

**try/except** - Перехватывает исключения, возбужденные инетр-ром или вашим программным кодом, и выполняет восстановительные операции.

**try/finally** - Выполняет заключительные операции независимо от того, возникло исключение или нет.

**raise** - Дает возможность возбудить исключение программно.

**assert** - Дает возможность возбудить исключение программно, при выполнении определенного условия.

**with/as** - Реализует менеджеры контекста в версиях 2.6 и 3.0 (в 2.5 явл-ся доп. расширением).

**Зачем нужны исключения?**

Исключения позволяют перепрыгнуть через фрагмент программы произвольной длины. Вспомним про нашу мащину по изготовлению пиццы и допустим мы её даже сделали.

Представим, что что-то пошло не так при "выпекании основы". Печь сломалась или робот ошибся. Понятно, что нам нужно предусмотреть быстрый переход к коду, которй быстро обрабатывает такие ситуации. А т.к. в таких условиях пиццы нам точно не получит нам нужно совсем остановить приготовление.

Это именно то, что позволяют делать исключения: программа может перейти к обработчику исключения за один шаг, отменив все вызовы ф-ций. После этого обработчик исключения может вып-ть действия, соответствующие ситуации.

Исключения - это своего рода "супер-goto". Обработчик исключений (инструкция try) ставит метку и вып-ет некий код. Если затем, где-то в программе, возникнет исключение, интер-р немедленно возвращается к метке, отменяя все активные вызовы ф-ций, которые были произведены после установки метки.

Это всё позволяет соотв. образом реагировать на необычные события. Кроме того, переход к обработчику исключения выполняется немедленно, поэтому нет нужды проверять коды возврата каждой вызванной ф-ции, которая могла потерпеть неудачу.

**Назначение исключений.**

Исключения в программах могу тиграть разные роли:

- *Обработка ошибок.* Интер-р возбуждает исключение каждый раз, когда обнаруживает ошибку во время выполнения программы. Прога может перехватывать такие ошибки и обрабатывать их или просто игнорировать. Если ошибка игнорируется, интер-р выполняет действия, предусмотренные по умолчанию - он останавливает выполнение проги и выводит сообщение об ошибке. Если такое поведение по умолч. явл-ся нежелательным, можно добавит ьинструкцию try, которая позволит перехватывать обнаруженные ошибки и продолжить выполнение проги после инструкции try.

- *Уведомление о событиях.* Исключения могут также использ-ся для уведомления о наступлении некоторых условий, что устраняет необходимость передавать куда-либо флаги рез-та или явно проверять их. Напр. ф-ция поиска может возбуждать исключение в случае неудачи, вместо того, чтобы возвращать целочисленный признак в виде рез-та.

- *Обработка особых ситуаций.* Некоторые условия могут наступать так редко, что было бы слишком расточительно предусматривать проверку наступления таких условий с целью их обработки. Нередко такие проверки обычных ситуаций можно заменить обработчикам исключений.

- *Заключительные операции.* Как будет показано далее, инструкция try/finally позволяет гарантировать выполнение завершающих операций независимо от наличия исключений.

- *Необычное управление потоком выполнения.* И, наконец, т.к. исключения - это своего рода оператор "goto", их можно использовать как основу для экзотического управления потоком выполнения программы. Напр. обратная трассировка не явл-ся частью самого языка, но она может быть реализована с пом. исключений и некоторой логики поддержки, выполняющей раскручивание операций присваивания. В Питоне отсутствует оператор goto, но исключения могут с успехом заменить его.

**Обработка исключений: краткий обзор.** стр. 921

**Обработчик исключений по умолчанию.**

Предположим у нас есть ф-ция:

>>> def fetcher(obj, index):

return obj[index]

Ф-ция просто извлекает элемент из объекта по заданному индексу. Если всё в порядке она возвращает рез-т:

>>> x = 'spam'

>>> fetcher(x, 3) # То же, что x[3]

'm'

Но если ф-ции передать индекс, выходящий за пределы строки, то получим встр. исключение IndexError:

>>> fetcher(x, 4) # Обработчик по умолч. - интерактивная оболочка

Traceback (most recent call last):

File "<pyshell#7>", line 1, in <module>

fetcher(x, 4)

File "<pyshell#3>", line 2, in fetcher

return obj[index]

IndexError: string index out of range

(выход индекса за пределы диапазона)

Т.к. наш код не перехватывает это исключение явно, оно возвращает выполнение на верх. уровень программы и вызывает обработчик исключений по умолч., который просто выводи тстандартное сообщение об ошибке. К этому времени мы видели уже много разных сообщений об ошибках. Они включают тип исключения, а также диагностическую информацию - список строк и ф-ций, которые были активны в момент появления исключения.

Текст ошибки может отличаться от версии интер-ра. При работе в интерактив. оболочке файлом явл-ся "<stdin>", т.е. стандарт. поток ввода:

>>> fetcher(x, 4)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in fetcher

IndexError: string index out of range

При рабоет в IDLE файлом явл-ся "<pyshell>" и доп. выводятся номера строк (так всё у нас и есть в примере, т.к. мы в основном работаем в IDLE).

В настоящей проге, запущенной не в оболочке, после сообщения об ошибке обработчик по умолч. сразу же завершает работу программы. Это имеет смысл для простых сценариев - как правило, ошибки в них должны быть фатальными и лучшее, что можно сделать - это ознакомиться с текстом ошибки.

**Обработка исключений.**

Но завершение работы не всегда, то, что нам нужно. Напр. серверные программы обычно должны быть аткивными даже после появления ошибок. Для этого нам нужн оперехватить исключение, обернув вызов ф-ции инструкцией try:

>>> try:

fetcher(x, 4)

except IndexError:

print('got exception')

...

got exception

>>>

Т.о. видим, что когда исключение возникает при выполнении инструкций в блоке try, интер-р автоматич. переходит к нашему обработчику (блок под предложением except, в котором указано имя исключения). Пр иработе в оболочке, после выполнения блока except происходит возврат в приглашение к вводу. В настоящих прогах инструкции try не тольк оперехватывают исключения, но и вып-ют действия по восстановлению после ошибки:

>>> def catcher():

try:

fetcher(x, 4)

except IndexError:

print('got exception')

print('continuing')

>>> catcher()

got exception

continuing

>>>

Здесь, после того как исключение было перехвачено и обработано, программа продолжила выполнение инже всей инструкции try - именно поэтому в данном примере было выведено сообщение "continuing". Стандартное сообщение об ошибке не появилось на экране, и программа продолжила работу как ни в чем не бывало.

**Возбуждение исключений.**

Все эти исключения возбуждались интер-ром когда он встречал наши ошибки. Однако наши сценарии также способны возбуждать исключения - т.е. исключения могут возбуждаться интер-ром или самой прогой, и могут перехватываться или нет.

Чтобы возбудить исключение вручную, нужно просто выполнить инструкцию raise. Исключения, определяемые прогой, перехватываются также как и встроенные. Пример:

>>> try:

raise IndexError

except IndexError:

print('got exception')

…

got exception

Если исключение, определяемое прогой не перехватывается, оно будет передано обработчику по умолч., что приведет к завершению проги и сообщению об ошибке:

>>> raise IndexError

Traceback (most recent call last):

File "<pyshell#26>", line 1, in <module>

raise IndexError

IndexError

Исключения также могут возбуждаться с пом. инструкции assert - это условная форма инструкции raise, которая использ-ся в основном для отладки в процессе разработки:

>>> assert False, 'Nobody expects the Spanish Inquisition!'

Traceback (most recent call last):

File "<pyshell#28>", line 1, in <module>

assert False, 'Nobody expects the Spanish Inquisition!'

AssertionError: Nobody expects the Spanish Inquisition!

**Исключения, определяемые пользователем.**

Инструкция raise возбуждает встр. исключения, которые определены во встр. области видимости. Далее мы узнаем, что можно определять новые исключения для внутр. нужд своих программ.

Польз. исключения создаются в виде классов, наследующих один из классов встр. исключений, - обычно класс с именем Exception. Польз. исключения позволяют создавать категории исключений, наследовать поведение и добавлять к ним инфу о состоянии:

>>> class Bad(Exception): # Пользовательское исключение

pass

…

>>> def doomed():

raise Bad() # Возбудит экземпляр исключения

…

>>> try:

doomed()

except Bad: # Перехватить исключение по имени класса

print('got Bad')

…

got Bad

>>>

**Заключительные операции.**

Наконец, инструкции try могут включать блоки finally. Эти блоки выглядят точно так же, как обработчики except. Комбинация try/finally определяет завершающие действия, которые всегда выполняются "на выходе", независимо от того, возникло исключение в блоке try или нет:

>>> try:

fetcher(x, 3)

finally:

print('after fetch')

…

'm'

after fetch

>>>

Здесь, если блок try выполнится без ошибок, будет выполнен блок finally и прога продолжит свою работу дальше. В этом случае данная инструкция окажется бессмысленной - мы могли бы просто добавить print сразу вслед за вызовом ф-ции и вообще убрать try:

fetcher(x, 3)

print('after fetch')

Однако в таком подходе имется одна проблема: если в ф-ции возникнет исключение, инструкция print не будет выполнена. Комбинация try/finally позволяет ликвидировать эту проблему - когда в блоке try действительн опроизойдет исклюение, блок finally будет выполнен, пока программа будет раскручиваться:

>>> def after():

try:

fetcher(x, 4)

finally:

print('after fetch')

print('after try?')

…

>>> after()

after fetch

Traceback (most recent call last):

File "<pyshell#58>", line 1, in <module>

after()

File "<pyshell#57>", line 3, in after

fetcher(x, 4)

File "<pyshell#3>", line 2, in fetcher

return obj[index]

IndexError: string index out of range

Здесь мы не получили сообщение "after try?" потмоу что работа проги не была продолжена после блока try/finally, когда возникло исключение. Вместо этого интер-р выполнил действия, предусмотренные блоком finally, после чего исключение достигло предыдущего обработчика (в данном случае - обработчик по умолч.). Если изменить вызов внутри ф-ции after, чтобы он не вызывал исключение, блок finally все равно будет выполнен, но прога продолжит работу после выхода из try:

>>> def after():

try:

fetcher(x, 3) # Изменили вызов с (х, 4) на (х, 3)

finally:

print('after fetch')

print('after try?')

>>> after()

after fetch

after try?

На практике, комбинацию try/except удобно использ. для перехвата и восстановления после исключений, а комбинацию try/finally - в случаях, когда нужно гарантировать выполнение заключительных действий независимо было исключение в try или нет.

Напр. try/except можно было бы ипольз. для перехвата ошибок, возникающих в импортированной библиотеке, созданной сторонним разработчиком, а комбинацию try/finally - чтобы гарантировать закрытие файлов и соединений с сервером.

Хоть эти две комбинации служат разным целям, начиная с Питон 2.5 появилась возможность смешивать предложения except и finally в одной и той же инструкции try - блок finally будет выполняться всегда, независимо от того, было ли перехвачено исключение предложением except.

Как узнаем далее, в 2.6 и 3.0 есть альтернатива try/finally, используемая с некоторыми типами объектов. Инструкция with/as выполняет логику, предусмотренную реализацией объекта, гарантируя выполнение заключительных операций:

>>> with open('spam.txt', 'w') as file: # Всегда закрывает файл при выходе

file.write('The larch!\n')

...

11

- теперь у нас в файле spam.txt строка "The larch!".

Такой подход позволяет сократить объем программного кода, но может прмиеняться лишь с некоторыми типами объектов, поэтому try/finally - более универсальный способ заключ. операций.

**Проверка ошибок.**

Полезность исключений можно увидеть просто сравнив программирование на Питон и на языке, не имеющим исключений. Напр. на С нам нужно проверят ьвозвращаемые значения или коды состояния после выполнения каждой операции, которая может быть выполнена с ошибкой и передавать рез-ты проверок в ходе выполнения проги:

doStuff()

{ # Программа на С

if (doFirstThing() == ERROR) # Проверить наличие ошибки

return ERROR; # даже если здесь она не обрабатывается

if (doNextThing() == ERROR)

return ERROR;

...

return doLastThing();

}

main()

{

if (doStuff() == ERROR)

badEnding();

else

goodEnding();

}

Фактич. в настоящих прогах на С значительная доля всего кода выполняет проверку наличия ошибок. Но в Питон достаточно обернуть участки кода обработчиками исключений и писать эти участки в ожидании, что ошибок не будет:

def doStuff():

# Код на Питон

doFirstThing() # Нас не беспокоят возможные исключения

doNextThing() # поэтому мы не выполняем проверку

...

doLastThing()

if \_\_name\_\_ == '\_\_main\_\_':

try:

doStuff() # Здесь нас интересуют возможные рез-ты,

except: # поэтомоу это единственное место, где нужна проверка

badEnding()

else:

goodEnding

Т.к. в случае исключения управление немедленно перейдет обработчику, здесь нет необходимости разбрасывать проверки по всему коду. Т.о. образом исключения позволяют игнорировать возможные необычные ситуации и отказаться от кода, выполняющего проверки на ошибки (как выше на С).

**Глава 33. Особенности использования исключений.** стр. 929

**Инструкция try/except/else.** стр. 930

Сначала исследуем try/except/else и try/finally как разные инструкции, т.к. они не могут комбинироваться в Питон инже 2.5, а затем уже рассмотрим их комбинации.

Инструкция try - составная инструкция. Начинается с строки заголовка try, за которой идет блок инструкци (обычно с отступами), затем следует одно или более предложений except, которые определяют обрабатываемые исключения, и затем следует необязательное предложение else. Слова try, except и else должны располагаться на одном уровне. Пример полного формата:

try:

<statements> # Сначала выполняются эти действия

except <name1>:

<statements> # Запускается, если в блоке try возникло исключение name1

except (name2, name3):

<statements> # Запускается, если возникло любое из этих исключений

except <name4> as <data>:

<statements> # Запускается в случае исключения name4

# и получает экземпляр исключения

except:

<statements> # Запускается для всех (остальных) возникших исключений

else:

<statements> # Запускается, если в блоке try не возникло исключения

Блок под заголовком try представляет основное действие инструкции - код, который следует попытаться выполнить. Предложения except определяют обработчики исключений, возникших в ходе выполнения блока try, а предложение else (если оно есть) определяет обработчик для случая отсутствия исключений. Элемент <data> имеет отношение к особенности инструкции raise, которую обсудим позднее.

Опишем принцип действия инструкции try. Когда запускается инструкция try, интер-р помечает текущий конекст программы, чтобы вернуться к нему, если возникнет исключение. В первую очередь вып-ся инструкции, располож. под заголовком try. Дальше всё зависит будет ли возбуждено исключение в блоке try или нет:

- Если исключение в блоке try возникнет, интер-р вернется к инструкции try и выполнит первое предложение except, соответствующее возбужденному исключению. После выполнения блока except управление будет передано первой инструкции, находящейся за всей инструкцией try (при условии, что в блоке except не возникло другое исключение).

- Если в блоке try возникло исключение и не было найдено ни одного соответствия среди предложений except, исключение будет передано инструкции try, стоящей выше в программе, или на верх. уровень процесса (что вынудит интер-р аварийно завершить работу программы и вывести сообщение об ошибке по умолч.).

- Если в процессе выполнения блока try не возникло исключение, интер-р выполнит инструкции в блоке else (если они есть) и затем выполнение продолжится с первой инструкции, находящейся за всей инструкцией try.

Др. словами except перехватывают любые исключения, которые могут возникнуть при выполнении блока try, а блок else вып-ся тольк ов случае отсутствия исключений в блок try.

В предложениях except нах-ся обработчики исключений - они перехватывают исключения, которые возникли тольк ов инструкциях блока try. Однако инструкции блока try могут вызывать ф-ции, расположенные в разных частях программы, поэтому сам источник исключения может располагаться за пределами самой инструкции try.

**Предложения инструкции try.**

**Таблица 33.1 Различные формы предложений в инструкции try.** стр. 931

Некоторые мы уже знаем, напр. except, finally, else. В инструкции может присутствовать несколько предложений except, но только одно предложение else. Вплоть до 2.4 предложение finally должно было быть единственным (без else и except). На самом деле try/finally - это отдельная инструкция. Однако начиная с 2.5 предложение finally может присутствовать в той же инструкции, что и предложения except и else.

Есследованием доп. значения as value мы займемся, когда будем рассматривать инструкцию raise. Оно обеспечивает доступ к объекту, который играет роль исключения.

Новыми для нас явл-ся первая и четвертая строки в табл. 33.1:

- Предложения except, в которых отсутствуют имена исключений (except:), перехватывают все исключения, ранее не перечисленные в инструкции try;

- Предложения except, где в круглых скобках перечислены имена исключений (except (e1, e2, e3):), перехватывают любое из перечисленных исключений.

Интер-р просматривает предложения except сверху вниз в поисках соответствия, поэтому версию предложения с круглыми скобками можно рассматривать как аналог нескольким отдельным выражениям except, по одному для каждого исключения из списка, только в этом случае тело обработчика явл-ся общим для всех указанных исключений. Пример использ. нескольких except:

try:

action()

except NameError:

...

except IndexError:

...

except KeyError:

...

except (AttributeError, TypeError, SyntaxError):

...

else:

...

Здесь, если при вып-ии ф-ции action возникнет исключение, интер-р возвращается к инструкции try и пытается отыскать первое предложение except, в котором указано возникшее исключение. Поиск среди except ведется сверху вниз, слева направо, и вып-ся инструкции в первом найденном совпадении. Если совпадений не найдено, исключение продолжит распространение выше этой инструкции try. Помним, что блок else вып-ся только при отсутствии исключения в ф-ции action - этот блок не вып-ся при наличии исключения, которому не было найдено соответствующее предложение except.

Для перехвата всех исключений используйте пустое предложение except:

try:

action()

except NameError:

...

except IndexError:

...

except: # Обраб-ть все остальные исключения, если нет NameError и IndexError

...

else:

...

Предложение except без имени исключения - это, своего рода, шаблонный символ, т.к. перехватывает любые исключения, что позволяет создавать и универсальные и специфич. обработчики. В некоторых случаях эта форма более удобна, чем перечисление всех возможных исключений. Пример:

try:

action()

except:

... # перехватить все возможные исключения

Однако применение пустых except несет и некоторые проблемы проектирования. Они могут перехватывать, напр., системные исключения не связанные с работой нашего кода, и случайн опрерывать распространение исключений, предназначенных для других обработчиков. Напр. даже выход из проги в Питон возбуждает исключение, и поэтому желательно такое исключение пропустить. Кроме того такая конструкция будет перехватывать исключения, вызванные обычными ошибками разработчика, которые нам навернка хотелось бы обнаружить и исправить. Т.е. предложение except требует внимательного отношения.

В 3.0 была введена альтернатива, - предложение except Exception имеет практич. тот же эффект, что и пустое except, но оно не перехватывает исключения, имеющие отношение к завершению программы (это один плюс, остальные минусы остались те же, что и у простого except):

try:

action()

except Exception:

... # Перехватывает все исключения, кроме завершения программы

***Примечание.*** В 3.0 нужно использ. форму предложения except E as V:, вместо старой формы except E, V:. Это сделано, чтобы не путать эту форму с указанием двух и более альтернативных исключений вида except (E1, E2):. Кроме того, при использовании ключевого слова as переменная V автоматич. удаляется в конце блока except (очищается память).

**Предложение try/else.** Стр. 934

Нзначение предложения else в инструкции try, для новичков Питона, не всегда очевидно. Но все-таки без этого предложения нет никакого способа узнать (без установки и проверки флагов) - выполнение проги продолжилось потому, что исключение в блоке try не было возбуждено, или потому, что исключение перехвачено и обработано:

try:

... выполняемый код ...

except IndexError:

... обработка исключения ...

# Программа оказалась здесь потому, что исключение было обработано

# или потому, что его не возникло?

Так же как и в операторах цикла, else в try однозначно и очевидно сообщает о произошедшем:

try:

... выполняемый код ...

except IndexError:

... обработка исключения ...

else:

... исключение не было возбуждено ...

То же самое поведение можно имитировать, переместив содержимое блока else в блок try:

try:

... выполняемый код ...

... исключение не было возбуждено ...

except IndexError:

... обработка исключения ...

Но такой ход может привести к некорректной классификации исключения. Если какая-либо инструкция в блоке "исключение не было возбуждено" приведет к появлению исключения IndexError, оно будет зарегистрировано как ошибка в блоке try и соответственно, ошибочно будет передано обработчику исключения ниже (тонко, но верно!).

Прия вном использ. else логика выполнения становится более очевидной и гаранитруется, что обработчики исключений будут вызываться только для обработки истинных ошибок в блоке, обернутом инструкцией try, а не при выполнении действий, предусматриваемых в блоке else.

**Пример: поведение по умолчанию.**

Мы знаем, что исключения, не перехваченные try, распространяются до самого верхнего уровня процесса и запускают логику обработки исключений по умолч. (т.е. интер-р аварийно замершает работу проги и выводит сообщение об ошибке). Пример, с делением на ноль:

def gobad(x, y):

return x / y

def gosouth(x):

print(gobad(x, 0))

gosouth(1)

Т.к. прога сама не обрабатывает это исключение, интер-р завершает её и выводит сообщение (в книге сообщение немного длиннее, там они запускали под 3.0, а у нас уже 3.4):

C:\Python34>bad.py

Traceback (most recent call last):

File "C:\Python34\bad.py", line 8, in <module>

gosouth(1)

File "C:\Python34\bad.py", line 6, in gosouth

print(gobad(x, 0))

File "C:\Python34\bad.py", line 3, in gobad

return x / y

ZeroDivisionError: division by zero

Сообщение состоит из содержимого стека вызовов ("Traceback") и имени (с доп. данными) исключения. В содержимом стека перечислены все строки, которые были активны в момент появления исключения, в порядке от более старых к более новым.

Обратите внимание, что в командной строке (не в оболочке) имена файлов и номера строк содержат для нас полезную инфу. Например, здесь видно, что ошибка произошла в 3-й строке в файле bad.py в инструкции return.

Замечание. В моей IDLE версии 3.4 инфа та же самая, т.е. у командной строки, по состоянию инфы, особого преимущества не видно. Ну это просто замечание, даже сам автор подчеркивает, что может оболочка выводить и полезную инфу.

Т.к. интер-р определяет и сообщает обо всех ошибках, появившихся во время выполнения программы, возбуждая исключения, эти исключения тесно связаны с идеями обработки ошибок и отладки вообще. Да мы и сами встречались с сообщениями об ошибках и даже пытались по ним искать причину. Естественно большинство из них были синтаксическими, но были и функциональные конечно.

**Пример: перехват встроенных исключений.**

Обработка интер-ром по умолч. чаще всего удовлетворяет всем потребностям, особенно для программного кода верхнего уровня, где ошибки должны приводить к немедленному завершению программы. Для большинства программ нет нужды особой обработки ошибок.

Однако иногда нужно перехватить ошибку и вып-ть восстановительные действия после неё. Или если не хотим, чтобы программа завершалась после ошибки. Это очень важно для серверов сети, например, которые должны работать не смотря ни на что.

Пример, перехват и обработка исключения TypeError, при попытке конкатенации списка и строки (а должны быть последовательности одного типа):

def kaboom(x, y):

print(x + y) # Это место возбуждает исключение TypeError

try:

kaboom([0, 1, 2], "spam")

except TypeError: # Исключение перехватывается и обрабатывается здесь

print('Hello World!')

print('resuming here') # Прога работает дальше,

# не зависимо было или нет исключение

Когда в ф-ци иkaboom возникает исключение, управление передается предложению except в инструкции try, где выводится текст сообщения. После того как исключение перехв-ся оно становится неактуальным, поэтому прога продолжает выполнение ниже инструкции try вместо того, чтобы завершиться:

C:\Python34>kaboom.py

Hello World!

resuming here

Внимание, - как только ошибка будет перехвачена, вып-ие продолжается с того места, где ошибка была перехвачена (т.е. после инструкции try), - нет никакой возможности вернуться к тому месту, где возникла ошибка (т.е. в ф-цию kaboom). Это делает исключения похожими на инструкцию перехода, - нет никакой возможности вернуться к коду, вызвавшему ошибку.

**Инструкция try/finally.** стр. 936

Др. разновидность инструкции try специализируется на выполнении заключ. операций. Если в try включено предложение finally, интер-р всегда будет выполнять блок инструкций при "выходе" из инструкции try независимо от того, произошло ли исключение во время вып-ия инструкций в блоке try. Общая форма:

try:

<statements> # Выполнить эти действия первыми

finally:

<statements> # Всегда выполнять этот блок кода при выходе

В первую очередь вып-ся инструкции в блоке try. Дальнейшее зависит от того возникло ли исключение в блоке try:

- Если в блоке try исключение не возникло, интер-р переходит к вып-ию блока finally и затем продолжает вып-ть программу ниже инструкции try.

- Если в блоке try возникло исключение, интер-р также вып-ит инструкции в блоке finally, но, после этого, исключение продолжит своё распространение до вышестоящей инструкции try или до обработчика исключений по умолч. - прога не будет вып-ся вслед за try. Т.е. инструкци ив блоке finally будут выполнены, даже если исключение будет, но в отличие о except, предложение finally не завершает распространение исключения - оно остается актуальным после вып-ия блока finally.

Форму try/finally удобно использ. когда нужно гаранитровать вып-ие некоторых действий. С практич. стороны, эта форма инструкции позволяет определять завершающие действия, которые должны вып-ся всегда, напр. закрытие файлов или закрытие соединений с сервером.

В Питон 2.4 и ниже finally не может использ. в той же инструкции try, где уже использ. предложение except или else, поэтому форму try/finally лучше считать отдельной формой инструкции при работе со старыми версиями.

Позднее узнаем, что в 2.6 и 3.0 инструкция with и конектстные менеджеры обеспечивают объектно-ориентированный подход к вып-ию аналогичных завершающих действий. Но, в отличие от finally, эта новая инструкция поддерживает возможность вып-ия действий по инициализации, хотя и ограничивается областью видимости объектов, которые реализуют протокол менеджеров контекста.

**Пример: реализация завершающих действий с помощью инструкции try/finally.**

Приведем типичный пример использ. инструкции try/finally:

class MyError(Exception): pass

def stuff(file):

raise MyError()

file = open('data', 'w') # Открыть файл для вывода

try:

stuff(file) # Возбуждает исключение

finally:

file.close() # Всегда закрывать файл, чтобы вытолкнуть буферы

print('not reached') # Продолжить с этого места,

# только если не было исключения

Итак, здесь мы обернули вызов ф-ции в инструкцию try с предложением finally, чтобы гарантировать закрытие файла при любых обстоятельствах.

Как мы знаем, объекты файлов автоматич. закрываются на этапе сборки мусора, что очень удобно с временными файлами. Однако не всегда можно предсказать время сборки мусора, особенно в больших прогах. Инструкция try делает закрытие файла явной операцией, предсказуемой. Она гарантирует, что файл будет закрыт при выходе из блока, не зависимо было исключение или нет.

Ф-ция здесь у нас просто генерирует исключение. А мы еще раз напоминаем - интер-р всегда выполняет код в блоке finally, независимо было исключение в блоке try или нет.

Когда ф-ция возбуждает исключение, управление перед-ся инструкции try и начинает вып-ся блок finally, в котором производится закрытие файла. После этого исключение продолжает своё распространение либо пока не встретит др. инструкцию try, либо пока не буде тдостигнут обработчик по умолч., который выведет сообщение об ошибке и остановит прогу - инструкция, находящаяся ниже инструкции try никогда не будет достигнута.

Если бы ф-ция в этом примере не возбуждала исключение, прога точно так же выполнила бы блок finally, чтобы закрыть файл, и затем продолжила бы своё выполнение ниже инструкции try. Также обратите внимание - исключение опять определено как класс.

**Объединенная инструкция try/except/finally.** Стр. 939

В Питон 2.6 и 3.0 можно объединять инструкции finally и except/else в одной конструкции. Т.е. посмотрим пример:

try:

основное действие

except Exception1:

обработчик\_1

except Exception2:

обработчик\_2

...

else:

блок else

finally:

блок finally

Первым, конечно же, выполняется код в блоке "основное действие". Если в нем возбуждается исключение, вып-ся проверка всех блоков except, одно за другим, в поисках соотествия исключению. Если возбуждено исключение Exception1 то вып-ся блок "обработчик\_1", если исключение Exception2, то вып-ся блок "обработчик\_2". Если в блоке try не возникло исключение, то буде твыполнен "блок else".

И вот уже затем, после всех этих действий, будет выполнен "блок finally". Причем будет выполнен даже если исключение возникнет в самом обработчике или блоке else.

Как всегда, finally не прекращает распространение исключения - если к моменту вып-ия блкоа finally есть активное исключение, оно продолжит своё распространение после блока finally и управление передается куда-то в др. место проги. Если к моменту вып-ия блкао finally нет исключений, выполнение проги продолжится сразу же вслед за инструкцией try.

Т.о. блок finally выполняется всегда, когда:

- В блоке основного действия возникло исключение и было обработано;

- В блоке основного действия возникло исключение и не было обработано;

- В блоке основного действия не возникло исключение;

- В одном из обработчиков возникло новое исключение.

Напомним, предложение finally служит, чтобы организовать выполнение завершающих действий, которые должны вып-ся всегда при выходе из инструкции try независимо от того, было исключение или нет.

**Синтаксис объединенной инструкции try.** стр. 940

Инструкция try, как минимум, должна содержать либо предложение except, либо предложение finally и порядок таков:

try -> except -> else -> finally

где else и finally явл-ся необязательными и можеть быть ноль и более предложений except, но в случае присутствия предложения else должно быть указано хотя бы одно предложение except. В действительности инструкция try состоит из двух частей: из предложений except с необязательным предложением else и/или предложения finally.

Опишем синтаксис объединенной инструкции (квадртные скобки означают необязательную формпу, а звездочка значит "ноль и более раз"):

try: # Формат 1

statements

except [type [as value]]:

statements

[except [type [as value]]:

statements]\*

[else:

statements]

[finally:

statements]

try: # Формат 2

statements

finally:

statements

Согласно этим правилам else может присутствовать, только если в инструкции есть хоть одно предложение except, и всегда можно одновременно указывать except и finally, независимо от наличия else.

Кроме того, допускается одновременно указывать finally и else, но только если в инструкции присутствует предложение except. А если порядок следования предложений будет нарушен, то интер-р возбудит исключение о синтаксической ошибке.

**Объединение finally и except вложением.**

До 2.5 существовала возможность объединять finally b except в инструкции try за счет вложения инструкции try/except в блок try инструкции try/finally (более полно рассмотрим этот прием в гл. 35). На самом деле фрагмент ниже имеет тот же эффект, что и новая форма инструкции, представленная в начале этого раздела:

try:

try:

основное действие

except Exception1:

обработчик\_1

except Exception2:

обработчик\_2

...

else:

нет ошибок

finally:

завершающее действие

Здесь блок finally всегда вып-ся при выходе из инструкции try независимо от того, что произошло в блоке основного действия, и независимо от того, выполнялись ли обработчики исключений во вложенной инструкции try. Т.к. else всегда требует наличия хотя бы одного предложения except, эта вложенная форма тмеет те же ограничения, что и объединенная форма инструкции.

Но этот эквивалент выглядит менее понятно, чем новая, объединенная, форма инструкции, и для её записи требуется больше программного кода (минимум на одну строку). Смешанная форма инструкции проще в написании и выглядит понятнее, поэтому такая форма записи считается в настоящее время предпочтительной.

**Пример использования объединенной инструкции try.**

В файле merged.py представлены четыре типичных варианта с инструкциями print, описывающими значение каждого из них:

sep = '-' \* 32 + '\n'

print(sep + 'EXCEPTION RAISED AND CAUGHT')

try:

x = 'spam'[99]

except IndexError:

print('except run')

finally:

print('finally run')

print('after run')

print(sep + 'NO EXCEPTION RAISED')

try:

x = 'spam'[3]

except IndexError:

print('except run')

finally:

print('finally run')

print('after run')

print(sep + 'NO EXCEPTION RAISED, EITH ELSE')

try:

x = 'spam'[3]

except IndexError:

print('except run')

else:

print('else run')

finally:

print('finally run')

print('after run')

print(sep + 'EXCEPTION RAISED BUT NOT CAUGHT')

try:

x = 1/0

except IndexError:

print('except run')

finally:

print('finally run')

print('after run')

Запустим наш код:

C:\Python34>merged.py

--------------------------------

EXCEPTION RAISED AND CAUGHT

except run

finally run

after run

--------------------------------

NO EXCEPTION RAISED

finally run

after run

--------------------------------

NO EXCEPTION RAISED, EITH ELSE

else run

finally run

after run

--------------------------------

EXCEPTION RAISED BUT NOT CAUGHT

finally run

Traceback (most recent call last):

File "C:\Python34\merged.py", line 33, in <module>

x = 1/0

ZeroDivisionError: division by zero

Этот пример в основном действии использует встр. операции и полагается на тот факт, что интер-р всегда определяет появление ошибок во время выполнения кода.

**Инструкция raise.** стр. 943

Чтобы явно возбудить исключение, можно использ. инструкцию raise. Синтаксис - инструкция raise состоит из слова raise, за которым может следовать имя класса или экземпляр возбуждаемого исключения:

raise <instance> # Возбуждает экземпляр класса-исключения

raise <class> # Создает и возбуждает экземпляр класса-исключения

raise # Повторно возбуждает самое последнее исключение

В Питон 2.6 и 3.0 исключение - это всегда экземпляр класса. Поэтому первая форма инструкции raise явл-ся наиболее типичной - ей непосредственно передается экземпляр класса, котоырй создается перед вызовом инструкции raise или внутри неё.

Если raise передается класс, интер-р вызовет конструктор класса без аргументов, а полученный экземпляр передаст инструкции raise - если после имени класса добавить круглые скобки, мы получим эквивалентную форму.

Третья форма raise просто повторно возбуждает текущее исключение - это удобно, когда возникает необходимость передать перехваченное исключение другому обработчику.

Примеры, эквивалентные друг другу:

raise IndexError # Класс (экземпляр создается неявно)

raise IndexError() # Экземпляр (создается в инструкции)

Мы можем и заранее создать экземпляр - raise принимает ссылки на объекты любого типа, примеры:

exc = IndexError()

raise exc

excs = [IndexError, TypeError]

raise excs[0]

При исключении интер-р отправляет возбужденный экземпляр вместе с исключением. Есди try включает предложение вида except name as X:, переменной Х будет присвоен экземпляр, переданной инструкции raise:

try:

...

except IndexError as X: # Переменной Х будет присвоен экземпляр исключения

Ключ. слово as явл-ся необязательным в обработчиках инструкции try (если оно опущено, интер-р просто не будет присваивать экземпляр переменной), но с его помощью можно получить доступ к данным экземпляра и методам класса исключения.

Так же действуют и исключения определяемые пользователем в виде классов. Пример, передача аргумента конструктору класса исключения, знач. которого становится доступным в обработчике через экземпляр, присвоенный переменной:

class MyExc(Exception): pass

...

raise MyExc('spam') # Вызов конструктора класса с аргументом

...

try:

...

except MyExc as X: # Атрибуты экземпляра доступны в обработчике

print(X.args)

Независимо какие исключения используются, они всегда идентифицируются обычными объектами и только одно исключение может быть активным в каждый конкретный момент времени. Как тольк оисключение перехватывается except, находящимся в любом месте проги, исключение деактивируется (т.е. не будет передано др. инструкции try), если не будет повторно возбуждено при помощи raise или простой ошибки.

**Пример: возбуждение и обработка собственных исключений.**

Итак, собственное исключение будет представлено объектом экземпляра класса (как и должно быть), как напр. MyBad у нас:

class MyBad: pass

def stuff():

raise MyBad() # Возбудит исключение вручную

try:

stuff() # Возбуждает исключение

except MyBad:

print('got it') # Здесь вып-ся обработка исключения

... # С этого места продолжается выполнение программы

Здесь исключение происходит внутри ф-ции, но это не имеет значения - управление немедленно передается блоку except. Видим, что инструкция try перехватывает собственные исключения программы так же, как и встр. исключения.

**Пример: повторное возбуждение исключений с помощью инструкции raise.**

Инструкция raise, в которой нет имени исключения или нет доп. данных, просто повторно возбуждает текущее исключение. Это делают, например, когда нужно перехватить и обработать исключение, но при этом не требуется деактивировать исключение:

>>> try:

raise IndexError('spam') # Исключения сохраняют аргументы

except IndexError:

print('propagating')

raise # Повторное возбуждение последнего исключения

...

propagating

Traceback (most recent call last):

File "<pyshell#6>", line 2, in <module>

raise IndexError('spam')

IndexError: spam

Видим как здесь raise повторно возбуждает исключение, которое затем передается обработчику более высокого уровня (или обработчику по умолч., который останавливает выполнение проги и выводи тсообщение об ошибке). Обратим внимание, как отображается значение аргумента в тексте сообщения об ошибке, который был передан конструктору класса - почему это происходит узнаем в след. главе.

**Измнения в Python 3.0: raise from.**

В 3.0 (но не в 2.6) raise может включать доп. предложение from:

raise exception from otherexception

При использ. from второе выражение определяет еще один класс исключения или экземпляр, который будет присвоен атрибуту \_\_cause\_\_ возбуждаемого исключения. Если исключение не будет перехвачено, интер-р выведет инфу об обоих исключениях:

>>> try:

1/0

except Exception as E:

raise TypeError('Bad!') from E

Traceback (most recent call last):

File "<pyshell#12>", line 2, in <module>

1/0

ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

(Исключение выше стало прямой причиной следующего исключения:)

Traceback (most recent call last):

File "<pyshell#12>", line 4, in <module>

raise TypeError('Bad!') from E

TypeError: Bad!

Когда исключение возбуждается в обработчике исключения, подобная процедура вып-ся неявно: предыдущее исключение присваивается атрибуту \_\_context\_\_ нового исключения и также выводится к стандартный поток ошибок, если исключение не будет перехвачено.

***Примечание.***

В 3.0 больше неподдерживается форма raise Exc, Args, которая все еще есть в 2.6. Вместо неё в 3.0 нужно использ. описанную выше форму raise Exc(Args), которая создает экземпляр исключения.

**Инструкция assert.** Стр. 946

Инструкция assert действует в качестве особого случая возбуждения исключений на этапе отладки. Это сокращенная форма типичного шаблона raise, которая представляет собой условную инструкцию raise. Инструкция вида:

assert <test>, <data> # Часть <data> является необязательной

представляет собой эквивалент следующего фрагмента:

if \_\_debug\_\_:

if not <test>:

raise AssertionError(<data>)

Др. словами, если условное выражение возвращает ложное значение, интер-р возбуждает исключение: элемент данных (если он есть) играет роль аргумента конструктора исключения. Как и все исключения, AssertionError приводит к завершению проги, если не будет перехвачено инструкцией try, и в этом случае элемент данных отображается как часть сообщения об ошибке.

Есть доп. возможность удалить все инструкции assert из скомпилированного байт-кода проги за счет использ. флага командной строки -0 при запуске интер-ра и тем самым оптимизировать прогу. Исключение AssertionError явл-ся встроенным исключением, а мя \_\_debug\_\_ - встр. флагом, который автоматич. получает знач. True (истина), когда не использ. флаг -0. Использ. команду фида python -0 main.py, чтобы запустить прогу в оптимизированном режиме и отключить все инструкции assert.

**Пример: проверка соблюдений ограничений (но не ошибок).**

Обычно assert использ. для проверки условий выполнения проги во время разработки. При отображении в текст сообщений об ошибках, полученных в рез-те вып-ия инструкции assert, автоматич. выключается инфа из строки исходного кода и значения, перечисленные в инструкции. Пример файла asserter.py:

def f(x):

assert x < 0, 'x must be negative'

return x \*\* 2

>>> import asserter

>>> asserter.f(1)

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

asserter.f(1)

File "C:\Python34\asserter.py", line 2, in f

assert x < 0, 'x must be negative'

AssertionError: x must be negative

Нужно не забывать, что инструкция assert глав. образом предназначена для проверки соблюдения ограничений, накладываемых программистом, а не для перехвата настоящих ошибок. Т.к. интер-р сам может находить ошибки, обычно нет нужды использ. assert для выявления таких проблем, ка квыход индекса за пределы, несоответствие типов или деление на ноль:

def reciprocal(x):

assert x != 0 # Бесполезная инструкция assert!

return 1/x # Интер-р автоматич. проверит х на равенство нулю

Такие инструкции assert явл-ся лишними, т.к. с подобными ошибками интер-р разберется и сам. Еще один пример использ. assert приводится в примере абстрактного суперкласса в гл. 28 - там assert использ. для того, чтобы вызов неопределенных методов приводил к исключению с определенным текстом сообщения.

**Конекстные менеджеры with/as.** стр. 948

Инструкция with, с необязательным предложением as, предназначена для работы с объектами контекстных менеджеров, которые поддерживают новый протокол взаимодействия, основанный на использовании методов. Эта особенность доступна и в 2.5 в виде расширения, которое активируется инструкцией:

from \_\_future\_\_ import with\_statement

Инструкция with/as может использ. как альтернатива известной идиомы try/finally; подобно этой инструкции она предназначена для вып-ия заключительных операций независимо, возникло ли исключение на этапе вып-ия основного действия. Но, в отличие от try/finally, инструкция with поддерживает более богатый возможностями протокол, позволяющий определять как предварительные, так и заключительные действия для блока кода.

Питон дополняет некоторые встр. средства контекстными менеджерами, напр. файлы, которые закрываются автоматич., или блокировки потоков выполнения, которые автоматич. запираются и отпираются. Но мы можем создавать с классами и свои контекстные менеджеры.

**Основы использования.**

Основная форма with:

with выражение [as переменная]:

блок with

Здесь предполагается, что выражение возвращет объект, поддерживающий протокол конекстного менеджера. Этот объект может возвращать значение, которое будет присвоено переменной, если присутствует необязательное предложение as.

Заметьте, что переменной необязательно будет присвоен результат выражения - результатом выражения явл-ся объект, который поддерживает контекстный протокол, а переменной может быть присвоено некоторое др. значение, предназначенное для использ. внутри ф-ции. Объект, возвращаемый выражением, может затем выполнять предварительные действия после того, как этот блок будет выполнен, независимо от того. было ли исключение при его выполнении.

Некоторые встр. объекты Питон были дополнены поддержкой протокола управления контекстом и потому могут использ. в инструкции with. Напр. объекты файлов имеют менеджер контекста, который автоматич. закрывает файл после вып-ия блока with независимо от того, было ли исключение:

with open(r'C:\misc\data') as myfile:

for line in myfile:

print(line)

... остальной программный код ...

Здесь, ф-ция open возвращает объект файла, который присваивается имени myfile. К переменной myfile можем применять обычные средства, предназначенные для работы с файлами, - в данном случае с пом. итератора вып-ся чтение строки за строкой в цикле for.

Но данный объект поддерживает протокол управления контекстом, используемый инструкцией with. После того как with начнет выполнение, механизм управления контекстом гарантирует, что объект файла, на который ссылается переменная myfile, будет закрыт автоматич., даже если в for во время обработки файла произойдет исключение.

Мы не знаем точно в какой момент сборщик мусора закроет файл. Инструкция with дает альтернативу, позволяющую гаранитровать, что файл будет закрыт сразу после выполнения определенного блока кода. Конечно можно использ. try/finally, но кода будет больше:

myfile = open(r'C:\misc\data')

try:

for line in myfile:

print(line)

... остальной программный код ...

finally:

myfile.close()

Многопоточную модель мы здесь изучать не будем, но блокировка и средства синхронизации посредством условных переменных также поддерживаются инструкцией with за счет обеспечения поддержки протокола управления контекстом:

lock = threading.lock()

with lock:

# Критическая секция программного кода

... доступ к совместно используемым ресурсам ...

Здесь механизм управления контекстом гарантирует, что блокировка автоматич. будет приобретена до того, как начнет вып-ся блок, и освобождена по завершении работы блока независимо от того, было ли исключение.

Модуль decimal, из гл. 5, также использ. менеджеры контекста для упрощения сохранения и восстановления текущего контекста вычислений, определяющего параметры точности и округления, используемые в вычислениях:

with decimal.localcontext() as ctx:

ctx.prec = 2

x = decimal.Decimal('1.00') / decimal.Decimal('3.00')

После вып-ия этой инструкции менеджер локального контекста текущего потока вып-ия автоматич. восстановит его в прежнее состояние, предшествовавшее началу вып-ия инструкции.

**Протокол управления контекстом.**

Некоторые встр. типы уже имеют менеджер контекстов, и мы сами можем добавлять менеджеры контекстов в свои классы. Для реализации использ. спец. методы классов, т.н. методы перегрузки операторов и они обеспечивают взаимодействие с with. Интерфейс, который должны реализовать объекты для использ. совместно с with, достаточно сложен, хотя чаще достаточно знать, как использ. существующие контексты менеджеров.

Коротко рассмотрим основные принципы, опишем работу инструкции with:

1. Производится вычисление выражения, возвращающего объект, известный как менеджер контекста, который должен иметь методы \_\_enter\_\_ и \_\_exit\_\_.

2. Вызывается метод \_\_enter\_\_ менеджера контекста. Возвращаемое значение метода присв-ся переменной в предложении as, если оно есть, в ином случае значение просто уничтожается.

3. Затем вып-ся блок кода, вложенный в инструкцию with.

4. Если, при вып-ии блока, возбуждается исключение, вызывается метод \_\_exit\_\_ (тип, значение, диагностическая\_инфа), которому передается подробная инфа об исключении. Заметьте, что это те же самые значения, которые возвращает ф-ция sys.exec\_info, описываемая в руководстве Питона и в книге. Если этот метод возвращает ложное значение, исключение возбуждается повторно, в противном случае исключение деактивируется. Обычно исключение следует возбуждать повторно, чтобы оно могло выйти за пределы инструкции with.

5. Если в блоке with исключение не возникнет, метод \_\_exit\_\_ всё равно вызывается, но в аргументах тип, значение и диагностическая\_информация ему передается значение None.

Рассмотрим небольшой пример, демонстрирующий работу протокола. Следующий фрагмент определяет объект менеджера контекста, который сообщает о входе и выходе из блока программного кода любой инструкции with, с которой он используется:

class TraceBlock:

def message(self, arg):

print('running', arg)

def \_\_enter\_\_(self):

print('starting with block')

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_tb):

if exc\_type is None:

print('exited normally\n')

else:

print('raise an exception!', exc\_type)

return False # Повторное возбуждение

with TraceBlock() as action:

action.message('test 1')

print('reached')

with TraceBlock() as action:

action.message('test 2')

raise TypeError

print('not reached')

Обратите внимание, что метод \_\_exit\_\_ должен возвращать False, чтобы разрешить дальнейшее распространение исключения - отсутствие инструкции return обеспечивает тот же самый эффект, потому что в этом случае по умолчанию возвращается знач. None, которое по сути явл-ся ложным.

Следует заметить, что метод \_\_enter\_\_ возвращает сам объект self, который присваивается переменной в предложении as; при желании этот метод может возвращать совершенно другой объект.

При запуске этого фрагмента менеджер контекста с пом. своих методов \_\_enter\_\_ и \_\_exit\_\_ отмечает моменты входа и выхода из блока инструкции with.

Приводим работу этого сценария:

C:\Python34>withas.py

starting with block

running test 1

reached

exited normally

starting with block

running test 2

raise an exception! <class 'TypeError'>

Traceback (most recent call last):

File "C:\Python34\withas.py", line 20, in <module>

raise TypeError

TypeError

Менеджеры контекста явл-ся новыми механизмами, предназначенными для разработчиков инструментальных средств, поэтому мы не будем рассматривать доп. подробности. В более простых случаях try/finally обеспечивает достаточную поддержку для выполнения завершающих действий.

В версии 3.1 (ну наверное и в нашей 3.4) в инструкции with можно будет определять сразу несколько (иногда их называют "вложенными") менеджеров контекста через запятую. В примере, напр. при выходе из блока инструкции with автоматич. вып-ся заключительные операции для обоих файлов независимо от наличия исключений:

with open('data') as fin, open('res', 'w') as fout:

for line in fin:

if 'some key' in line:

fout.write(line)

В одной инструкции with может быть перечислено любое количество менеджеров контекста, которые будут действовать как влож. инструкции with. Вообще реализация в 3.1 (и выше)Ж

with A() as a, B() as b:

... инструкции ...

эквивалентна следующей реализации, которая будет работать в 3.1, 3.0 и 2.6:

with A() as a:

with B() as b:

... инструкции ...

**Глава 34. Объекты исключений.** стр. 954

Мы знаем, что в 2.6 и 3.0 встроенные и пользовательские исключения идентифицируются объектами экземпляров классов. Это как-бы означает, что нам нужны будут приемы ООП для создания исключений, но и не забываем, что классы и ООП имеют свои преимущества. Определим некоторые преимущества исключений на классах:

- **Они могут быть организованы в категории.** Классы исключений поддерживают изменения в будущем - добавление новых исключений в будущем вообще не будет требовать изменений в инструкциях try.

- **Они могут нести в себе информацию о состоянии.** Классы исключений предосталяют естественное место для хранения инфы, доступной для обработчиков в инструкции try.

- **Поддерживают наследование.** Напр. могут иметь методы отображения, чтобы был единый стиль сообщений об ошибках.

В Питон 3.0 пользовательские исключения наследуют суперклассы встр. исклюений. Поэтмоу для успешного создания собственных сиключений нам нужно хорошее понимание встроенных.

В 2.6 и 3.0 исключения, должны определяться как классы. Также в 3.0 нужно, чтобы классы исключений наследовали суперкласс встр. исключения BaseException, прямо или косвенно. Обычно большинство польз. исключений наследуют подкласс Exception этого суперкласса для поддержки универсальных обработчиков исключений обычных типов.

**Исключения: назад в будущее.**

Начнем исследования исключений с прошлого, т.к. мы не можем совсем отказаться от старых программ и знать почему изменения в Питоне оказались именно такими какие они есть сейчас.

**Строковые исключения ушли в прошлое!**

До 2.6 и 3.0 была возможность определять исключения в виде экземпляров классов и в виде объектов строк. Строковые исключения были удалены в 2.6 и 3.0, поэтому сейчас использ. только классовые исключения. Тем не менее мы все еще можем столкнуться со строковыми исключениями, поэтому ознакомимся с ними.

Строк. исключения были просты с использовании - можно использ. любые строки, а сопоставление выполнялось по идентичности объекта, но не по назначению (т.е. с пом. оператора is, а не ==):

# Представим, что мы запустили этот сценарий в оболочке Питон 2.5

myexc = "My exception string"

try:

raise myexc

except myexc: print('caught')

...

caught

**Исключения на основе классов.** стр. 956

Важным преимуществом классов явл-ся возможность организовывать исключения в категории и большая гибкость чем у строк. Кроме того хранение инфы и поддержка наследования.

Помимо отличий в коде, главное различие между строковыми и исключениям ина базе классов заключается в способе идентификации возбужденных исключений в предложениях except инструкции try:

- Строковые исключения идентифицируются по идентичности объекта: идентификация возбужденного исключения в предложении except выполняется с пом. оператора is.

- Исключения на основе классов идентифицируются отношением наследования: возбужденное исключение считается соответствующим предложению except, если в данном предложении указан класс исключения или любой из его суперклассов.

Т.е. когда в инструкции try предложение except содержит суперкласс, оно будет перехватывать экземпляры этого суперкласса, а также экземпляры всех его подклассов, ниже в дереве. Благодаря этому исключения на основе классов поддерживают иерархию исключений: суперклассы превращаются в имена категорий, а подклассы соответствуют различным видам исключений внутри категории. Используя имя общего суперкласса, except может перехватывать целую категорию исключений - каждый подкласс будет соответствовать этому предложению.

**Создание классов исключений.**

Рассмотрим пример. Файл classexc.py, определяем суперкласс General и два подкласса Specific1, Specific2. Здесь у нас General - это имя категории, а подклассы - типы исключений внутри категории. Обработчики которые перехватывают исключения General, будут перехватывать и его подклассы:

class General(Exception): pass

class Specific1(General): pass

class Specific2(General): pass

def raiser0():

X = General() # Возбуждает экземпляр суперкласса исключения

raise X

def raiser1():

X = Specific1() # Возбуждает экземпляр подкласса исключения

raise X

def raiser2():

X = Specific2() # Возбуждает экземпляр другого подкласса исключения

raise X

for func in (raiser0, raiser1, raiser2):

try:

func()

except General: # Перехватывает исключения General и любые его подклассы

import sys

print('caught:', sys.exc\_info()[0])

C:\Python34>classexc.py

caught: <class '\_\_main\_\_.General'>

caught: <class '\_\_main\_\_.Specific1'>

caught: <class '\_\_main\_\_.Specific2'>

Отметим несколько примечаний к реализации этого сценария:

- **Суперкласс Exception.**

К классам, используемым для построения дерева категорий исключений предъявляется не так много требований. Фактич. все классы в примере - пустые, в них только инструкции pass. Но заметьте, наш суперкласс наследует встр. класс Exception. Это явл-ся обязательным требованием в 3.0 - классич. классы в 2.6 тоже могут играть роль исключений, но если класс исключения наследует класс встр. исключения, он автоматич. становится классом нового стиля (как все классы в 3.0). Класс Exception даёт массу полезных особенностей, как мы увидим далее.

- **Возбуждение экземпляров.**

В примере мы создаем экземпляры классов в инструкциях raise. В модели исключений на классах, мы всегда возбуждаем и перехватываем объекты экземпляров классов. Если в raise имена классов указываются без круглых скобок, интер-р будет автоматически создавать экземпляры, вызывая их конструкторы без аргументов. Экземпляры исключений могут создаваться до вызова инструкции raise, как в нашем примере, или внутри неё.

- **Перехватывание категорий.**

В этом примере определены ф-ции, которые возбуждают экземпляры всех трех классов исключений, а на верхнем уровне модуля определена инструкция try, которая вызывает ф-ции и перехватывает исключения класса General. Та же инструкция try перехватывает и два более специфических исключения, потому что они явл-ся подклассами класса General.

- **Информация об исключении.**

В этом примере, обработчик исключений использ. ф-цию sys.exc\_info - эта ф-ция обеспечивает обощенный способ получить последнее возбужденное исключение. Первый элемент, в полученном рез-те, - это класс возбужденного исключения, а второй - фактический экземпляр исключения. В подобных предложениях except, перехватывающих все исключения, принадлежащие некоторой категории, как в нашем примере, ф-ция sys.exc\_info явл-ся единственным способом точно определить, что произошло. В нашем случае её можно определять как эквивалент обращения к атрибуту \_\_class\_\_ экземпляра. Далее увидим, что ф-ция sys.exc\_info часто использ. в обработчиках пустых предложений except.

Поясним последний пункт. Внутри обработчика можно быть уверенным, что возбужденный экземпляр явл-ся экземпляром класса, указанного в предложении except, или одного из его подклассов. Благодаря этому тип исключения можно также получить из атрибута \_\_class\_\_ экземпляра. Пример, действующий так же как и предыдущий:

class General(Exception): pass

class Specific1(General): pass

class Specific2(General): pass

def raiser0(): raise General()

def raiser1(): raise Specific1()

def raiser2(): raise Specific2()

for func in (raiser0, raiser1, raiser2):

try:

func()

except General as X: # X - возбужденный экземпляр

print('caught:', X.\_\_class\_\_) # То же, что и sys.exc\_info()[0]

Т.е. ф-цию sys.exc\_info() удобнее использ. в обработчиках пустых предложений except, где нет другого способа получить доступ к экземпляру или к его классу. Кроме того, в действующих прогах обычно не нужно беспокоиться о конкретном типе возбужденного исключения - вызывая методы экземпляра, мы автоматич. получаем поведение, присущее возбужденному исключению.

**В чем преимущества иерархий исключений?**

Т.к. в прошлом примере всего три исключения мы не можем полностью показать перимущества классов исключений. На самом деле можно было то же самое, указав в предложении except список имен исключений в круглых скобках:

try:

func()

except (General, Specific1, Specific2): # Перехватывает все эти исключения

...

Однако, в случае разветвленных или глубоких иерархий исключений, может быть проще перехватывать категории, используя классы. Или расширять существующий код, адаптировать и т.д.

Напр. мы делаем библиотеку, реализующую ф-цию обработки числовой инфы, которая широко используется. Во время работы над библиотекой мы видим две ситуации, которые могут привести к таким ошибкам, как деление на ноль и переполнение. Мы описываем их как исключения:

# mathlib.py

class Divzero(Exception): pass

class Oflow(Exception): pass

def func():

...

raise Divzero()

Теперь те, кто будет польз. библиотекой, станут стремиться обертывать вызовы ваших функций или классов инструкцией try, чтобы перехватывать наши два исключения:

# client.py

import mathlib

try:

mathlib.func(...)

except (mathlib.Divzero, mathlib.Oflow):

... обработка и восстановление после ошибки ...

Теперь всё ок и многие использ. нашу библиотеку. Но позднее, мы, просматривая код, видим еще одну ситуацию, др. ошибку - потерю значимых разрядов, и добавляем новое исключение:

# mathlib.py

class Divzero(Exception): pass

class Oflow(Exception): pass

class Uflow(Exception): pass

Выпуская эту версию библиотеки мы создаем проблему уже существующим пользователям. Если они явно указали имена наших исключений, то им придется вернуться к своим прогам и внести соответствующие изменения везде, где приводятся обращения к нашей библиотеке, чтобы включить новое исключение:

# client.py

import mathlib

try:

mathlib.func(...)

except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):

... обработка и восстановление после ошибки ...

Конечно всё зависит от масшатаба, кол-ва клиентов библиотеки и т.п. Мы даже могли бы написать сценарий, который будет ликвидировать проблему автоматически. Но если многим людям придется изменять инструкции try каждый раз когда мы меняем набор исключений в библиотеке, то это будет очень печально. Также мы могли бы создать в библиотеке кортеж исключений, чтобы клиенты использовали его, а мы, в случае необходимости просто обновляли бы кортеж. Но этот прием обладает недостаточной гибкостью, хотя и простой.

Клиенты могут пытаться избежать ловушки, определяя пустые except, перехватывая все исключения:

# client.py

try:

mathlib.func(...)

except:

... обработка и восстановление после ошибки ...

Но теперь у нас будут перехватываться не ошибки библиотеки, а все возможные ошибки программы, т.е. всего приложения, что для нас нежелательно. Да и вообще, чем больше определенности при обработке исключений, тем лучше.

Так вот, исключения на основе классов полностью ликвидируют эту проблему. Сформируем исключения в дерево классов с одним общим суперклассом:

# mathlib.py

class NumErr(Exception): pass

class Divzero(NumErr): pass

class Oflow(NumErr): pass

...

def func():

...

raise Divzero()

При таком подходе, пользователям нашей библиотеки нужно будет указать лишь общий суперкласс (т.е. категорию), чтобы перехватывать все исключения в библиотеке, как имеющиеся, так и возможные в будущем:

# client.py

import mathlib

...

try:

mathlib.func(...)

except mathlib.NumErr:

... обработка и восстановление после ошибки ...

Теперь новые исключения можно добавлять просто как новые подклассы от общего суперкласса:

# mathlib.py

...

class Uflow(NumErr): pass

В итоге, прогарммный код пользователей, перехватывающий исключения вашей библиотеки, останется работоспособным без каких-либо изменений. Мы свободно сможем добавлять, удалять и изменять исключения произвольным образом - пока клиенты используют имя суперкласса, они могут не беспокоиться об изменениях в нашем наборе исключений. Т.о. исключения на основе классов лучше отвечают требованиям сопровождения, чем строки.

**Классы встроенных исключений.** стр. 962

Все встр. исключения, которые могут возбуждаться интер-ром, явл-ся объектами предопределенных классов. Они организованы в иерархию с общими суперклассами категорий и подклассами определенных типов исключений.

В Питон 3.0 все исключения, которые мы уже видели, явл-ся обычными классами, доступными в виде встр. имен в модуле builtins (в 2.6 \_\_builtins\_\_) и в виде атрибутов модуля exceions, в стандарт. библиотеке.

Небольшой обзор иерархии исключений:

- **BaseException.** Корневой суперкласс исключений. Не предназначен для наследования польз. классами (для этого есть Exception). Он соедржит реализацию п оумолчанию вывода сообщений и обеспечивает сохранение инфы о состоянии. Если встр. ф-ции str передать экземпляр этого класса (напр. спом. print), класс вернет строку с аргументами, которые передавались конструктору при создании экземпляра (или пустую строку, если конструктор вызывался без аргументов). Если подклассы не переопределяют конструктор этого класса, все аргументы, передаваемые ему при создании экземпляра, сохр-ся в атрибуте args экземпляра в виде кортежа.

- **Exception.** Корневой суперкласс всех прикладных исключений. Он прямой потомок суперкласса BaseException и суперкласс для всех др. встр. исключений, кроме классов, связанных с событиями хавершения проги (SystemExit, KeyboardInterrupt и GeneratorExit). Почти все польз. классы исключений должны наследовать этот класс, а не BaseException. При соблюдении этого соглашения, предложения except инструкции try, в которых указано исключение Exception, буду тперехватывать все исключения, кроме событий завершения программы, которые обычно обрабатывать не требуется.

- **ArithmeticError.** Суперкласс всех арифметических ошибок (и подкласс класса Exception).

- **OverflowError.** Подкласс класса ArithmeticError, идентифицирующий конкретную арифметическую ошибку.

Смотрите структуру дерева классов в справочных руководствах или в тексте справки для модуля exceptions (только в 2.6, в 3.0 этого модуля нет).

**Категории встроенных исключений.**

Дерево встр. классов позволяет определить, насколько конкретными или универсальными будут ваши обработчики исключений. Напр. встр. исключение ArithmeticError - это суперкласс для более конкретных исключений, как OverflowError и ZeroDivisionError. Указав имя ArithmeticError в инструкции try, мы будем перехватывать все арифметические ошибки, а указав имя OverflowError, мы будем перехватывать только ошибки определенного типа и никакие другие.

Также можно использ. исключение Exception - суперкласс всех прикладных исключений в 3.0, чтобы организовать обработку всех исключений. Использ. этого класса похоже на пустой except, но дает игнорировать исключения, связанные с завершением проги:

try:

action()

except Exception:

... обработать все прикладные исключения ...

else:

... обработать ситуациюб отсутствия исключений ...

Но этот прием не так универсален в 2.6, т.к. классич. классы исключений не явл-ся подклассами Exception. Но и в 3.0 есть минусы, - этому приему характерны все потенциальные ловушки, что и для пустого предложения except (см. ранее).

Но, тем не менее, при использовании классов мы можем реализовывать гибкие наборы исключений, которые легко изменять.

**Операция вывода по умолчанию и сохранение информации.**

Встр. исключения реализуют вывод сообщения, по умолч., и сохранение инфы о состоянии, что дает значительную долю логики работы исключений, необходимую для польз. классов. Если в своих классах вы не переопределяете встр. конструктор суперкласса исключения, любые аргументы, передаваемые конструктору, будут сохр-ся в атрибуте args экземпляра и автоматич. включаться в текст сообщения при выводе экземпляра (если без аргументов, будет выведен пустой кортеж и стандарт. строка сообщения).

Это объясняет, почему аргументы, передаваемые конструкторам классов встр. исключений, включаются в текст сообщения об ошибке - все аругменты присоед-ся конструктором к экземпляру и отображаются при попытке вывести его:

>>> raise IndexError # То же, что и IndexError(): нет аргументов

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

raise IndexError

IndexError

>>> raise IndexError('spam') # Конструктор присоединит аргумент

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

raise IndexError('spam')

IndexError: spam

>>> I = IndexError('spam') # Аргументы доступны в виде атрибута

>>> I.args

('spam',)

То же относится и к польз. исключениям, т.к. они наследуют от встр. суперклассов конструктор и методы вывода:

>>> class E(Exception): pass

...

>>> try:

raise E('spam')

except E as X:

print(X, X.args) # Выведет аргументы, сохраненные конструктором

...

spam ('spam',)

>>>

>>> try:

raise E('spam', 'eggs', 'ham')

except E as X:

print(X, X.args)

...

('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')

>>>

Видим, что объекты экземпляров исключений сами по себе не явл-ся строками, но они используют протокол перегрузки операторов, который мы изучали в гл. 29, и реализуют метод \_\_str\_\_, обеспечивающий преобразование экземпляра в строку. Чтобы выполнить конкатенацию экземпляра с настоящей строкой, его нужно вручную преобразовать в строковое представление: str(X) + "string".

Автоматич. поддержка вывода и сохр. инфы удобна сама по себе, тем не менее, чтобы вып-ть сохр-ие доп. инфы и вывод спец. сообщений, мы всегда можем переопределить унаследованные методы, такие как \_\_str\_\_ и \_\_init\_\_, в подклассах класса Exception.

**Определение текста исключения.**

По умолч. исключения на основе классов выводят знач. всех аргументов, которые были переданы конструктору класса, если их перехватить и вывести:

>>> class MyBad(Exception): pass

...

>>> try:

raise MyBad('Sorry--my mistake!')

except MyBad as X:

print(X)

...

Sorry--my mistake!

>>>

Эта же модель использ., когда исключение отображается в составе сообщения об ошибке, если оно не будет перехвачено прогой:

>>> raise MyBad('Sorry--my mistake!')

Traceback (most recent call last):

File "<pyshell#32>", line 1, in <module>

raise MyBad('Sorry--my mistake!')

MyBad: Sorry--my mistake!

Чаще этого достаточно. Но, чтобы улучшить сообщение, нужно переопределить в классе исключения один из двух методов перегрузки операции вывода (\_\_repr\_\_ или \_\_str\_\_). Строка, возвращаемая методом, будет отображаться при выводе экземпляра исключения вручную или когда исключение буде тперехвачено обработчиком по умолч.:

>>> class MyBad(Exception):

def \_\_str\_\_(self):

return 'Always look on the bright side of life...'

>>> try:

raise MyBad()

except MyBad as X:

print(X)

...

Always look on the bright side of life...

>>>

>>> raise MyBad()

Traceback (most recent call last):

File "<pyshell#42>", line 1, in <module>

raise MyBad()

MyBad: Always look on the bright side of life...

Нужно отметить, что в подобных ситуациях предпочтительнее переопределять метод \_\_str\_\_, т.к. встр. суперклассы уже имеют метод \_\_str\_\_, который в большинстве случаев (вкл. операцию вывода) польз-ся преимуществом перед \_\_repr\_\_. Если переопределить \_\_repr\_\_, операция вывода благополучно вызовет метод \_\_str\_\_ вместо нашей версии метода \_\_repr\_\_!

Независимо, что вернет наш метод, это знач. будет включено в текст ошибки для неперхваченных исключений и использовано при попытке вывести экземпляр явно. В нашем примере метод возвращает жестко определенную строку, однако он может вып-ть любую обработку текста и использ. инфу о состоянии, присоединенную к объекту экземпляра.

**Передача данных в экземплярах и реализация поведения.** стр. 966

Помимо гибкой иерархии, классы исключений могут хранить доп. инфу в виде атрибутов экземпляров. Также можно писать свои конструкторы, и определять доп. методы для использ. в обработчиках, предоставляя предопределенную логику обработки исключений.

**Передача дополнительной информации об исключении.**

Когда возбуждается исключение, оно способно пересекать границы модулей - инструкция raise, запускающая исключение, и инструкция try, перехватывающая его, могут находиться в разных модулях. Глобал. переменные не подходят для сохр. доп. инфы, т.к. код в try может не знать, в каком модуле находятся переменные. Передча инфы внутри самого экземпляра исключения явл-ся более надежным способом получить её в инструкции try.

При использ. классов это может быть почти автоматич. Как мы видели, при возбуждении исключения вместе с ним интер-р передате экземпляр класса. Обработчики в try могу тполучить доступ к экземплярам исключений, если в except указать имя переменной после ключевого слова as. Этот прием обеспечивает естественный способ передачи данных и поведения обработчику.

Прога, выпоняющая анализ файлов, может, например, сообщать об ошибке форматирования, возбуждая экземпляр исключения, котоырй заполняется доп. инфой об ошибке:

>>> class FormatError(Exception):

def \_\_init\_\_(self, line, file):

self.line = line

self.file = file

>>> def parser():

raise FormatError(42, file='spam.txt') # если обнаружена ошибка

>>> try:

parser()

except FormatError as X:

print('Error at', X.file, X.line)

...

Error at spam.txt 42

>>>

Здесь, переменой Х в предложении except присв-ся ссылка на экземпляр, который был сгенерирован во время возбуждения исключения. Благодаря этой переменной мы получаем доступ к атрибутам, присоединенным к экземпляру нашей реализации конструктора. Конечно, мы могли бы положиться на реализацию сохранения инфы, имеющуюся во встр. суперклассах, но такой способ хуже подходит для нашего приложения:

>>> class FormatError(Exception): pass # Наследует конструктор по умолч.

>>> def parser():

raise FormatError(42, 'spam.txt') # Именованные аргументы недопустимы!

>>> try:

parser()

except FormatError as X:

print('Error at', X.args[0], X.args[1]) # Не так удобно для

... # данного приложения

Error at 42 spam.txt

>>>

**Предоставление методов исключений.**

Адаптированные классы исключений могут использ-ся и для реализации специфического поведения объектов исключений. Т.е. класс исключения может определять доп. методы для использ. в обработчиках.

Пример, доп. метод, использующий инфу о состоянии для регитсрации ошибки в файле:

class FormatError(Exception):

logfile = 'formaterror.txt'

def \_\_init\_\_(self, line, file):

self.line = line

self.file = file

def logerror(self):

log = open(self.logfile, 'a')

print('Error at', self.file, self.line, file=log)

def parser():

raise FormatError(40, 'spam.txt')

try:

parser()

except FormatError as exc:

exc.logerror()

Если сценарий запустить, в ответ на вызов метода внутри обработчика исключения он запишет сообщение об ошибке в файл:

C:\Python34>parse.py

C:\Python34>type formaterror.txt

Error at spam.txt 40

При использ. подобных классов методы (такие как logerror) могут наследоваться подклассами, а атрибуты экземпляра (Line и file) предоставляют возможность сохр-ия инфы о состоянии, обеспечивая доп. контекст для последующих вызовов методов. А т.к. исключения опред-ся в виде классов, все преимущеста ООП, о которых мы говорили ранее, доступны и при работе с исключениями.

**Глава 35. Использование исключений.** стр. 971

**Вложенные обработчики исключений.**

А что же будет если вложить одну инструкцию try в другую? И что будет, например, если внутри инструкции try вызывается ф-ция, которая вып-ет др. инструкцию try?

Оба эти варианта проще понять, если мы узнаем, что интер-р складывает инструкции try стопокй во время вып-ия. Когда возникает исключение, интер-р возвращается к самой последней инструкции try, содержащей соответствующее, предложение except. Т.к. каждая инструкция try оставляет метку, интер-р может возвращаться к более ранним инструкциям try, двигаясь по стопке меток. Такое вложение активных обработчиков и есть то, что подразумевается, когда говорится о распространении исключений вверх, к обработчикам "более высого уровня". Эти обработчики явл-ся обычными инструкциями try, в которых поток управления ходом вып-ия проги вошел раньше.

**Рисунок 35.1 Вложенные инструкции try/except.** стр. 972

Рис. 35.1 показывает, что бывает, когда возникает вложение инструкций try/except во время вып-ия. Объем кода в try, может быть существенным (напр. вызовы ф-ций) и нередко вызывает др. код, который готов перехватить те же самые исключения.

Когда исключение возбуждается интер-р переходит к самой последней инструкции try, в которой указано имя исключения , запускает блок except и продолжает вып-ие проги ниже этой инструкции try.

Как только такое исключение будет перехвачено, его жизнь заканчивается - управление не передается всем соответствующим инструкциям try, содержащим имя исключения , - только первая из них получает возможность обработать исключение. Напр. на рис. 35.1 инструкция raise в ф-ции func2 возвращает управление обработчику в ф-ции func1, после чего прога продолжает вып-ие внутри func1.

И напротив, когда исключение возникает во влож. инструкциях try/finally, вып-ся каждый блок finally по очереди - интер-р продолжает передават ьисключение вверх по цепочке влож. инструкций try, пока не будет достигнут обработчик по умолч. верхнего уровня (стандарт. сообщения об ошибках).

**Рисунок 35.2 Вложенные инструкции try/finally.** стр. 972

Как показано на рис. 35.2, предложения finally не останавливают распространение исключений - они лишь определяют код, который должен вып-ся на выходе из инструкции try в процессе движения исключения. Если к моменту возникновения исключения имелось несколько активных инструкций try/finally, они все будут вып-ны, если только где-то на пути исключения не встретится инструкция try/except, которая перехватит его.

Др. словами, куда будет переход при возникновении исключения, зависит от того, где оно возникло - это определяется ходом вып-ия проги, а не только синтаксисом. Распространение исключения, по сути, происходит в порядке, обратном порядку вхождения в инструкции try. Это движение останавл-ся, когда управление переходит к соотв. блоку except, и продолж-ся когда управление проходит через предложения finally.

**Пример: вложение в потоке управления.**

В файле nestexc.py определяются две ф-ции. Ф-ция action2 возбуждает исключение (нельзя складывать числа и послед-ти), ф-ция action1 обертывает вызов action2 в инструкцию try:

def action2():

print(1 +[]) # Возбуждает исключение TypeError

def action1():

try:

action2()

except TypeError: # Самая последняя соответствующая инструкция try

print('inner try')

try:

action1()

except TypeError: # Этот обработчик будет выполнен, только если

print('outer try') # action1 повторно возбудит исключение

C:\Python34>nestexc.py

inner try

Мы видим, что на верх. уровне модул, внизу файла, вызов action1 также обернут в try. В момент, когда action2 возбуждает исключение TypeError, сущ-ют две активные инструкции try - одна в ф-ции action1 и одна в коде верх. уровне модуля. Интер-р выбирает и запускает самую последнюю инструкцию try с соответствующим предложением except, который в данном случае явл-ся инструкция try в ф-ции action1.

Как уже знаем, место, куда вып-ся переход исключения, зависит от того, в каком месте проги находится поток управления. Поэтому, чтобы знать, куда будет выполнен переход, нужно знать место, где находится поток управления. В данном случае выбор места, где будет обработано исключение, больше зависит от того, где находится поток управления, чем от синтаксиса.

**Пример: синтаксическое вложение.**

В гл. 33, когда изучали объединеную инструкцию try/except/finally, мы узнали, что можно вкладывать инструкции try синтаксически, задавая вложение в коде:

try:

try:

action2()

except TypeError: # Самая последняя соотв. инструкция try

print('inner try')

except TypeError: # Этот обработчик будет выполнен, только если

print('outer try') # вложенный обработчик повторно возбудит исключение

Этот код задает ту же влож. структуру обработчиков, что и предыдущий. Фактич., инструкции, влож. синтаксически, работают так же как показано на рис. 35.1 и 35.2, одно отличие только в том, что влож. обработчики физически соединены в блоке try, а не находятся в разных ф-циях. Напр. исключение пройдет через все блоки finally независимо, вложены они синтаксически или в ходе вып-ия проги идет вложение физически отдельных фрагментов кода:

>>> try:

try:

raise IndexError

finally:

print('spam')

finally:

print('SPAM')

...

spam

SPAM

Traceback (most recent call last):

File "<pyshell#74>", line 3, in <module>

raise IndexError

IndexError

Порядок вып-ия показан на рис. 35.2 - рез-т тот же самый, но сама логика вып-ия в данном случае образована влож. инструкциями. Более интересный пример синтаксического вложения, файл exc-fin.py:

def raise1(): raise IndexError

def noraise(): return

def raise2(): raise SyntaxError

for func in (raise1, noraise, raise2):

print('\n', func, sep=' ')

try:

try:

func()

except IndexError:

print('caught IndexError')

finally: # Нах-ся на уровне первого try

print('finally run')

Этот фрагмент перехватывает исключение, если оно буде твозбуждено, и выполняет завершающие действия в блоке finally независимо от того, возникло исключение или нет. Понять это не легко, но рез-т очень напоминает объединение except и finally в единственной инструкции try в Питон 2.5 или выше:

C:\Python34>exc-fin.py

<function raise1 at 0x017B2588>

caught IndexError

finally run

<function noraise at 0x017B25D0>

finally run

<function raise2 at 0x017B2540>

finally run

Traceback (most recent call last):

File "C:\Python34\exc-fin.py", line 9, in <module>

func()

File "C:\Python34\exc-fin.py", line 3, in raise2

def raise2(): raise SyntaxError

SyntaxError: None

Мы знаем, что с 2.5 есть возможность исполь. except и finally в одной инструкции try. Это делает описанный выше прием ненужным, но он по-прежнему работает, его можно встретить в коде в Питон ниже 2.5 и это т.н. альтернатива.

**Идиомы исключений.** стр. 975

Мы рассмотрели внутр. механизм исключений. Теперь рассмотрим некоторые др. способы их использования.

**Исключения не всегда являются ошибками.**

В Питон все ошибки явл-ся исключениями, но не все исключения явл-ся ошибками. Напр. в гл. 9 мы видели, что по достижении конца файла метод чтения объекта файла возвращает пустую строку. И напротив, встр. ф-ция input читает по одной строке текста при каждом вызове из стандарт. потока ввода sys.stdin и возбуждает исключение EOFError по достижении конца файла (в 2.6 ф-ция наз-ся raw\_input).

В отличие от методов объекта файла, данная ф-ция не возвращает прустую строку; пустая строка, полученная от input, ознает лишь пустую строку. Исключение EOFError, в данном контексте, - это всего лишь сигнал, а не ошибка. Поэтому, чтобы избежать завершения сценария ф-цию input обертывают инструкцией try, которую вкладывают в цикл:

while 1:

try:

line = input()

except EOFError:

break

else:

... обработка следующей строки ...

Есть и др. встр. исключения, которые явл-ся сигналами, а не ошибками, - вызов ф-ции sys.exit() и нажатие комбинации клавиш Ctrl-C, например, возбуждают исключение SystemExit и KeyboardInterrupt соответственно. В Питон есть ряд встр. исключений, которые явл-ся скорее предупреждениями, чем ошибками. Некоторые сообщают о нежелательности использования некоторых особенностей языка (которые скоро будут удалены, напр.).

**Передача сигналов из функций по условию.**

Исключения, определяемые прогой, могут быть сигналами об условиях, которые не явл-ся ошибками. Напр., поиск может ожидать исключение в случае нахождения соответствия вместо того, чтобы возвращать флаг состояния, который должен интерпретироваться вызывающей прогой.

Пример, инструкция try/except/else играет роль инструкции if/else, предназначенной для проверки возвращаемого значения:

class Found(Exception): pass

def searcher():

if ...успех...:

raise Found()

else:

return

try:

searcher()

except Found: # Исключение, если элемент найден

...успех...

else: # иначе: элемент не найден

...неудача...

В широком смысле такая организация кода может с успехом использ-ся для любой ф-ции, которая не может вернуть спец. признак, свидетельствующий об успехе или неудаче. Напр., если любое возвращаемое значение явл-ся допустимым, невозможно выбрать какое-то одно значение, которое сигнализировало бы о необычных состояниях. Исключения обеспечивают такой способ подать сигнал, не возвращая значение:

class Failure(Exception): pass

def searcher():

if ...успех...:

return ...найденный элемент...

else:

raise Failure()

try:

item = searcher()

except Failure:

...сообщение о неудаче...

else:

...обработка найденного элемента...

Т.к. Питон явл-ся динамически типизированным и в своей основе поддерживает полиморфизм, исключения, а не возвращение спец. признака, явл-ся более предпочтительным способом сообщать о таких состояниях.

**Закрытие файлов и соединений с сервером.**

Еще раз повторим, инстурменты обработки исключений также часто использ. с целью обеспечить освобождение системных ресурсов независимо от того, возникло исключение в процессе работы или нет.

Напр., некоторые серверы требуют, чтобы по завершении сеанса работы соединение было закрыто. Аналогично, после операции вывода в файл, может потребоваться закрыть его, чтобы вытолкнуть содержимое буферов на диск, а неиспользуемые файлы, открытые для чтения, могут понапрасну занимать файловые дескрипторы - объекты файлов автоматич. закрываются сборщиком мусора, но иногда бывает очень сложно знать, когда это произойдет на самом деле.

Простой и очевидный способ гарантировать выполнение заключительных операций для какого-то блока кода заключ-ся в использ. инструкции try/finally:

myfile = open(r'C:\miscscript', 'w')

try:

...обработать myfile...

finally:

myfile.close()

Как знаем из гл. 33, некоторые объекты в 2.6 и 3.0 еще больше упрощают такую возможность, предоставляя менеджеры контекстов, которые могут использ-ся совместно с инструкцией with/as, позволяющие автоматич. выполнять заключительные операции:

with open(r'C:\miscscript', 'w') as myfile:

...обработать myfile...

Какой вариант лучше зависит, как обычно, от вашей программы. В сравнении с try/finally менеджеры контекста менее очевидны, что противоречит философии Питона - максимум простоты и понятности. Кроме того, менеджеры контекста менее универсальны - доступны лишь для некоторых типов объектов, к тому же создание собств. менеджеров контекста, реализующих заключит. операции, вообще явл-ся более трудной задачей, чем использ-ие try/finally.

С др. стороны, использ. существующих менеджеров контекста требует меньше программного кода, чем применение инструкции try/finally, как видно из предыдущих примеров. Также протокол менеджеров контекста кроме заключ. операций предусматривает также возможность реализации начальных операций, выполняемых на входе. Инструкция try/finally, возможно, использ-ся более широко, но менджеры контекста могут оказаться предпочтительнее там, где они доступны или где сложность их создания оправдывается удобством использования.

**Отладка с помощью внешних инструкций try.**

Обработчики исключений можно использ. и как замену обработчика по умолч. Обернув всю программу (или её вызов) во внешнюю инструкцию try, можно перехватывать любые исключения, отменяя этим способ завершения проги, заднный по умолч.

Пример, пустое except перехватывает любые необработанные исключения во время вып-ия проги. Чтобы получить доступ к самому исключению, вызовем встр. ф-цию sys.exc\_info из модуля sys - она возвращает кортеж, в котором первые два элемента - это имя исключения и экземпляр класса исключения:

try:

...запуск программы...

except:

import sys

print('uncaught!', sys.exc\_info()[0], sys.exc\_info()[1])

Этот прием часто использ-ся во время разработки, т.к. он позволяет сохранить программу активной даже после ошибки - с его помощью можно делать доп. проверки без необходимости перезапускать программу. Также этот прием можно использ. для тестирования др. кода, см. след. раздел.

**Запуск тестов в рамках единого процесса.**

Некоторые приемы можно было бы объединить в тестовом приложении, которое позволяет тестировать другой код в рамках одного и того же процесса:

import sys

log = open('testlog', 'a')

from testapi import moreTests, runNextTest, testName

def testdriver():

while moreTests():

try:

runNextTest()

except:

print('FAILED', testName(), sys.exc\_info()[:2], file=log)

else:

print('PASSED', testName(), file=log)

testdriver()

Здесь, ф-ция testdriver выполняет в цикле серию тестов (модуль testapi - некая абстракция в этом примере). Т.к. обычно необработанное исключение приводило бы к завершению самого тестового приложения, можно обернуть вызовы очередного теста инструкцией try, чтобы тестирование продолжалось после неудачного завершения любого из тестов.

Здесь, пустое except перехватывает любые исключения, в ходе тестов, и регистрирует в файле инфу об исключении, полученную с пом. sys.exc\_info. Предложение else вып-ся в случае отсутствия исключений - т.е. тестирование завершилось хорошо.

Этот подход типичен для систем тестирования ф-ций, модулей и классов. На практике тестирование может быть гораздо сложнее. Напр., чтобы протестировать внеш. прогу, может потребоваться проверять коды состояния или вывод, создаваемый такими средствами запуска программ, как os.system и os.popen (такие инструменты вообще не возбуждают исключений в случае появления ошибок во внеш. проге - фактич. тест вып-ся параллельно с прогой, выполяющей тестирование).

**Подробнее о функции sys.exc\_info**

Ф-ция sys.exc\_info позволяет обработчикам исключений получить доступ к последнему возбужденному исключению. Её удобно использ. в пустых except, которые перехватывают все исключения, чтобы определить, что именно произошло:

try:

...

except:

# sys.exc\_info()[0:2] - класс исключения и экземпляр

Если в момент её вызова никакое исключение не обрабатывается, ф-ция возвращает кортеж с тремя объектами None. В противном случае возвр-ся (тип, значение, трассировочная\_информация), где:

- Тип - это класс обрабатываемого исключения.

- Значение - это экземпляр класса возбужденного исключения.

- Трассировочная информация - это объект, который представляет стек вызовов в точке, где возникло исключение (в документации к модулю traceback описываются инструменты, которые могут использ-ся вместе с этим объектом для создания сообщений об ошибках вручную).

Как мы видели, иногда ф-ция sys.exc\_info может также использ-ся, чтобы определить конкретный тип исключения, когда вып-ся перехват по имени суперкласса категории исключений. Однако, в подобном случае тип исключения можно определить и с пом. атрибута \_\_class\_\_ экземпляра, который можно получить с пом. ключевого слова as, поэтому ф-ция sys.exc\_info чаще всего использ-ся в пустых except:

try:

...

except General as instance:

# instance.\_\_class\_\_ - класс исключения

Кроме того, используя интерфейсы объектов экземпляров и опираясь на полиморфизм, часто бывает лучше просто использ. методы класса исключения, чем проверять его тип:

try:

...

except General as instance:

# instance.method() выполнит действия, ожидаемые от этого экземпляра

Как обычно, проверять типы объектов в Питон означает ограничивать гибкость кода, что не есть хорошо. Реализации на полиморфизме обычно обеспечивают лучшую поддержку возможных изменений в будущем.

В 2.6 для извлечения типа и значения последнего исключения можно использ. стареы инструменты sys.exc\_type и sys.exc\_value, но они могут использ. только применительно к единственному исключению, глоабльному для всего процесса. Ф-ция sys.exc\_info(), доступна и в 2.6 и в 3.0, теперь она более предпочтительна чем старые инструменты.

**Советы по применению и типичные проблемы исключений.** стр. 980

Настоящее искусство использования исключений заключается в принятии решения, насколько универсальными должны быть предложения except и какой объем кода должен быть обернут инструкциями try. Рассмотрим эти вопросы.

**Что должно быть обернуто.**

Конечно можно обернуть каждую инструкцию в инструкцию try, но это было бы слишком уж (тогда нужно и try обернуть в try!). Эта проблема, с обертыванием, никак не связано с конкретным ЯП и более очевидна на практике. Поэтому приводим несколько практических правил:

- В инструкции try следует заворачивать операции, для которых неудача не явл-ся чем-то необычным. Напр. операции, взаимодействующие с системой (открытие файлов, работа с сокетами и т.д.), явл-ся первыми кандидатами для заключения их в try.

- При этом из первого правила есть исключение - в простых сценариях бывает нужно, чтобы подобные неудачи приводили к завершению работы программы. Это особенно верно, когда неудачи ожидаемы. Неудачи в языке Питон приводя тк выводу полезных сообщений (только не в случае краха программы, конечно же), и они часто представляют собой лучший результат, на который можно только надеяться.

- Завершающие операции должны заключаться в инструкции try/finally, чтобы гаранитровать их выполнение. Эта форма инструкции позволяет выполнять программный код независимо, возникло исключение или нет.

- Иногда более удобно завернуть вызов крупной ф-ции в одну инструкцию try, чем засорять эту ф-цию несколькими инструкциями try. При таком подходе все исключения, возникшие в ф-ции, будут перехвачены инструкцией try, окружающей вызов, за счет чего можно уменьшить объем кода внутри самой ф-ции.

Количество обработчиков нередко зависит от типа программы. Напр. серверные проги должны работать постоянно, и поэтому в них инструкции try наверняка будут необходимы, чтобы перехватывать исключения и выполнять восстановительные операции после них. В тестировании также нужно вып-ть обработку исключений. Однако в простых сценариях часто можно вообще игнорировать исключения, т.к. неудача на любом этапе вып-ия требует прекращения работы.

**Не перехватывайте слишком много: избегайте пустых предложений except.**

Иногда нужно быть осторожнее, чтобы не перехватывать слишком много. Например как в случае с пустым except, который перехватывает все исключения, которые могут быть в try. Иногда это полезно, но может привести к тому, что будет перехвачена ошибка, обработка которой предусмотрена в инструкции try на более высоком уровневлож. структуры.

Пример, обработчик перехватывает и деактивируетвсе исключения, которые достигнут его, независимо, ожидает ли какие-то сиключения обработчик уровнем выше:

def func():

try:

... # Здесь возбуждается исключение IndexError

except:

... # Но все исключения попадают сюда!

try:

func()

except IndexError: # Исключение должно обрабатываться здесь

...

Что еще хуже, такой код может перехватыват ьисключения, которые вообще не имеют отношения к программе. Напр. ошибки работы с памятью, настоящие ошибки в коде, прекращение итераций, прерывание с клавиатуры и выход из программы, возбуждают исключения. Обчно такие исключения не должны перехватываться.

Напр., сценарии обычно завершают работу, когда поток управления достигает конца файла. Однако в Питон есть ф-ция sys.exit(statuscode), с пом. которой можн озавершить работу программы. Чтобы заверишть прогу, эта ф-ция возбуждает исключение SystemExit, благодаря чему есть возможность реализовать выполнение завершающих операций в инструкции try/finally, а в специализированных прогах - перехватить это событие. (Примечание: похожая ф-ция os.\_exit также завершает работу проги, но делает это непосредственно, - она пропускает этап вып-ия завершающих действий и не может быть перехвачена с пом try/except и try/finally.)

По этой причине инструкция try с пустым except может нечаянно перехватить такое важное исключение, пример файла exiter.py:

import sys

def bye():

sys.exit(40) # Серьезная ошибка: завершить работу немедленно!

try:

bye()

except:

print('got it') # Ой! Мы проигнорировали команду на завершение

print('continuing...')

C:\Python34>exiter.py

got it

continuing...

Мы просто не можем предугадать все исключения, которые могут произойти (особенно в крупном модульном проекте с несколькими разработчиками, или просто большой программе). Решить проблему в этом конкретном случае можно с пом. встр. классов исключений, благодаря тому, что суперкласс Exception не наследуется классом SystemError:

try:

bye()

except Exception: # Не будет препятствовать завершению программы,

... # но БУДЕТ, перехватывать массу других исключений!

Однако в др. случаях такой подход не лучше пустого except - т.к. Exception явл-ся суперклассом всех встр. исключений, кроме исключений завершения программы, при его использ. будут перехватываться все исключения, обрабатывать которые, возможно, предполагается где-то в другом месте в программе.

Хуже всего то, что пустое except может перехватить настоящие ошибки в коде, которым желательн обыло бы позволить пройти дальше. Фактич. пустые except могут отключать механизм интер-ра, предназначенный для вывода сообщений об шибках, скрывая возможные ошибки в коде. Например:

mydictionary = {...}

...

try:

x = myditctionary['spam'] # Ой, опечатка в имени словаря!

except:

x = None # А мы предполагаем, что получили KeyError

... продолжение работы с х ...

Здесь программист предполагает, что в данной ситуации возможен единственный тип ошибки - это ошибка отсутствующего ключа. Но поскольку в имени словаря mydictionary сделана ошибка (myditctionary), интер-р возбуждает исключение NameError, встретив ссылку на неопределенное имя, которое будет перехвачено и проигнорировано обработчиком. Обработчик неправильно запишет в переменную значение по умолч., замаскировав ошибку в проге. И даже использ-ие Exception в предложении except даст тот же эффект, что и пустое except. В зависимости от запутанности и величины сценария отладка такой ошибки может быть очень трудоемка.

Возьмите за правило специализировать свои обработчики насколько это возможно - пустые except удобны в использовании, но они потенциально опасны. В послднем примере лучше было бы использ. предложение except KeyError, для избежания перехвата посторонних событий. В простых сценариях проблемы могут быть не такими критичными, но в общем случае универсальные обработчики обычно доставляют массу неприятностей.

**Не перехватывайте слишком мало: используйте категории.**

С др. нежелательно делать обработчики слишком узкоспециализированными. Когда в инструкции try перечисляются конкретные исключения, перехватываться будут только те исключения, которые были перечислены. Это не обязательно плохо, но если в процессе развития программы появится новое исключение, нам нужно будет вернуться и добавить его в список обрабатываемых в своем коде.

Мы изучали эту проблему ы предыдущей главе. Пример, обработчик интерпретирует исключения MyExcept1 и MyExcept2 ка кнормальную ситуацию, а все остальные - как ошибку. Если в будущем добавим MyExcept3, оно будет обрабатываться как ошибка, если его не добавить в список исключений:

try:

...

except (MyExcept1, MyExcept2): # Работает неправильно при добавлении MyExcept3

... # Нет ошибки

else:

... # Рассматривается как ошибка

При осторожном использовании исключений на классах, можно избавиться от этой ловушки. Как мы знаем, если перехватывать общий суперкласс, в будущем можно будет добавлять и возбуждать более конкретные подклассы исключений без необходимости изменять список исключений в except - суперкласс становится легко расширяемой категорией исключений:

try:

...

except SuccessCategoryNme: # Работает правильно при добавлении MyExcept3

... # Нет ошибки

else:

... # Рассматривается как ошибка

Др. словами порой придется пройти длинный путь, чтобы найти оптимальное решение. Мораль такая, что нам слудет с особым тщанием подходить к выбору степени детализации, чтобы обработчики исключений не были ни слишком универсальными, ни слишком детализированными. Политика должна быть составной частью общего дизайна, особенно в крупных программах.

**Заключение по основам языка.** стр. 984

Итак, на этом мы завершаем изучение основ языка программирования Python. Нам осталось подвести некоторые итоги и можно присутпать к изучению практического программирования.

**Набор инструментальных средств Питона.**

Иерархия инструментальных средств в Питоне следующая:

- **Встроенные.** Встроенные типы, такие как, например, строки, списки и словари, помогают быстро создавать несложные программы.

- **Расширения на языке Питон.** Для решения более сложных задач вы можете расширить возможности Питона своими собственными ф-циями, модулями и классами.

- **Компилируемые расширения.** Возможности Питон можно расширять с пом. модулей, написанных на др. языках программирования, таких как С или С++ (или Java).

Приведем примеры некоторых встр. и функциональных возможностей, доступных в Питон:

*Типы объектов* - Списки, словари, файлы, строки ...

*Функции* - len, range, open ...

*Исключения* - IndexError, KeyError ...

*Модули* - os, tkinter, pickle, re ...

*Атрибуты* - \_\_dict\_\_, \_\_name\_\_, \_\_class\_\_ ...

*Внешние инструменты* - NumPy, SWIG, Jython, IronPython, Django и др.

**Инструменты разработки крупных проектов.**

Приведем краткое описание некоторых наиболее часто используемых инструментов:

- *PyDoc и строки документирования.* Ф-ция help и HTML-интерфейсы PyDoc были изучены в гл.15. Модуль PyDoc реализует документацию для модулей и объектов и интегрирован со строками документации. Это стандарт. часть системы Питона, читайте справку.

- *PyCheker.* Питон - это динамический ЯП, поэтому ошибки бывает сложно найти, пока не запустишь прогу, напрмие рсинтаксические. Конечно нужны тесты перед запуском, их нужно обязательн описать и выполнять. При использ-ии Питона этап компиляции замещается этапом начального тестирования. Кроме того есть вывод сообщений об ошибках, что помогает в их устранении. Системы PyCheker и PyLint помогают выявлять ошибки еще до запуска сценариев. Это сторонние пакеты, распространяемые с открытыми исходными текстами.

- *PyUnit (он же unittest).* В гл. 24 мы добавляли код самопроверки, который проверял \_\_name\_\_ == '\_\_main\_\_'. Доп, для нужд тестирования, в Питон есть два инструмента. PyUnit (в руководстве наз-ся unittest), обеспечивает комплект классов, для настройки тестов. Напоминает библиотеку JUnit в Java.

- *doctest.* Модуль doctest реализует более простой подход к регрессивному тестированию. Основан на использ. строк документирования. Чтобы воспольз-ся этим модулем нужно скопировать рез-ты тестирования в интерактив. сеансе в строки документирования в файле с исходным текстом. После этого doctest извлечет эти строки документирования,Ю вычленит из них описание тестов с ожидаемыми рез-ми и повторно вып-ит тесты, чтобы сравнить полученные рез-ты с ожидаемыми.

- *Интегрированные среды разработки.*Такие IDE как Eclipse, Komodo, NetBeans, Wing, а также наша знакомая IDLE, еще список смотрите в гл. 3 или на сайтах в Интернет.

- *Профилировщики.* Конечно мы хотим знать производительность своих программ. Намнужно добавить логику, выполняющую временные замеры с пом. инструментов в модулях time и timeit, или с пом. своего кода по управлением модуля profile. Профилирование - обычно самый первый шаг, который вып-ся на этапе оптимизации, выявляющий узкие места программы. Модуль profile - это стандарт. модуль, выдает отчет с нифой о производительности переданной ему строки (напр. импорт файла или вызов ф-ции). Этот модуль может запускаться как самостоятельный сценарий, импортироваться и доп. настраиваться. Пример работы, импортируйте модуль profile (интерактив. сеанс напр.) и вызовите ф-цию profile.run('code'), передав ей строку с кодом, который нужно оценить. Из системной оболочик нужно вызывать команду python -m profile main.py args...

- *Отладчики.* Стандарт. модуль отладки pdb, подобно профилировщику отладчик запускается и в интерактив. сеансе и в системной оболочке, импортироваться и вызываться, как ф-ция из модуля pdb (pdb.run("main()")), после чего можно вводить команды отладчика. Из систем. строки запускается так же как и профилировщик. Используется редко, т.к. в современных IDE (даже в IDLE) есть свои отладчики.

- *Варианты распространения.* Инструменты, такие как py2exe, PyInstaller, freeze могут упаковывать байт-код и вирутальную машину с интер-ром Питон в "фиксированные двоичные файлы", способные вып-ся как самостоятельные программы. Также можно распространять файлами .py или в виде байт-кода .pyc, также реализуются самораспаковывающиеся архивы и шифрование байт-кода. Также есть стандарт. модуль упаковывающий сценарии - distutils. Также появилась (на момент написания книги) система "eggs" - альтернативная система подготовки дистрибутивов.

- *Способы оптимизации.* Система Psyco, оптимизирует по скорости выполнения. Shedskin - транслятор исходных текстов с Питона на С++. Файлы .pyo с оптимизированным байт-кодом. Наконец, можно отдельные части программ переносить на компилирующий ЯП, напр. на С.

**Упражнения к седьмой части.** стр. 991