Исключения и менеджеры контекста

Исключения

Зачем нужны исключения?

- Исключения нужны для *исключительных* ситуаций, например:
  - не удалось выделить память для объекта,
    ```
    >>> [0] * int(1e16)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    MemoryError
    ```
  - импортируемый модуль не был найден,
    ```
    >>> import foobar
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'foobar'
    ```
  - программист написал код, складывающий список и число
    ```
    >>> [1, 2, 3] + 4
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: can only concatenate list to list
    ```
- Исключения — это ошибки, которые можно обрабатывать. В этом их прелесть.

Обработка исключений: `try...except`

- Для обработки исключений в Python используются операторы **`try`** и **`except`**:

```
>>> try:
...     something_dangerous()
... except (ValueError, ArithmeticError):
...     pass
... except TypeError as e:
...     pass
```

- Ветка **`except`** принимает два аргумента:
  1. выражение, возвращающее тип или кортеж типов,
  2. опциональное имя для перехваченного исключения.
- Исключение `e` обрабатывается веткой **`except`**, если её первый аргумент `expr` *можно сопоставить* с исключением: `isinstance(e, expr)`
- При наличии нескольких веток **`except`** интерпретатор сверху вниз ищет подходящую.

Обработка исключений: подробнее о `try...except`

- На месте выражения в ветке **`except`** может стоять любое выражение, например, вызов функции или обращение к переменной:

```
>>> try:
...     something_dangerous()
... except Exception as e:
...     try:
...         something_else()
...     except type(e):  # Какое исключение мы
...         pass         # перехватим?
```

- Время жизни переменной `e` ограничивается веткой **`except`**:

```
>>> try:
...     1 + "42"
... except TypeError as e:
...     pass  # Что делать, если нам нужно e?
...
>>> e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'e' is not defined
```

- **BaseException** — базовый класс для встроенных исключений в Python.

```
>>> BaseException.__subclasses__()
[<class 'Exception'>, <class 'GeneratorExit'>,
 <class 'KeyboardInterrupt'>, <class 'SystemExit'>]
```

- Напрямую от класса **BaseException** наследуются только системные исключения и исключения, приводящие к завершению работы интерпретатора.
- Все остальные встроенные исключения, а также исключения, объявленные пользователем, должны наследоваться от класса **Exception**.
- Отсюда следует, что, чтобы обработать любое исключение, достаточно написать:

```
>>> try:
...     something_dangerous()
... except Exception:  # Почему не BaseException?
...     pass
```

Встроенные исключения: AssertionError

- Исключение **AssertionError** поднимается, когда условие оператора **assert** не выполняется:

```
>>> assert 2 + 2 == 5, ("Math", "still", "works")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: ('Math', 'still', 'works')
```

- Оператор **assert** используется для ошибок, которые могут возникнуть только в результате ошибки программиста, поэтому перехватывать **AssertionError** считается дурным тоном.

Встроенные исключения: ImportError и NameError

- Если оператор **import** не смог найти модуль с указанным именем, поднимается исключение **ImportError**:

```
>>> import foobar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'foobar'
```

- **NameError** поднимается, если не была найдена локальная или глобальная переменная:

```
>>> foobar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foobar' is not defined
```

Встроенные исключения: AttributeError и LookupError

- Исключение **AttributeError** поднимается при попытке прочитать или (в случае `__slots__`) записать значение в несуществующий атрибут:

```
>>> object().foobar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'foobar'
```

- Исключения **KeyError** и **IndexError** наследуются от базового класса **LookupError** и поднимаются, если в контейнере нет элемента по указанному ключу или индексу:

```
>>> {}["foobar"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foobar'
>>> [][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Встроенные исключения: ValueError и TypeError

- Исключение **ValueError** используется в случаях, когда другие более информативные исключения, например, **KeyError**, не применимы:

```
>>> "foobar".split("")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: empty separator
```

- Исключение **TypeError** поднимается, когда оператор, функция или метод вызываются с аргументом несоответствующего типа:

```
>>> b"foo" + "bar"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
```

- Полный список исключений можно найти в документации языка[1].

[1] https://docs.python.org/3/library/exceptions.html

Исключения, объявленные пользователем

- Для объявления нового типа исключения достаточно объявить класс, наследующийся от базового класса `Exception`.
- Хорошая практика при написании библиотек на Python — объявлять свой базовый класс исключений, например:

```
>>> class CSCException(Exception):
...     pass
...
>>> class TestFailure(CSCException):
...     def __str__(self):
...         return "lecture test failed"
```

- Наличие базового класса позволяет пользователю обработать любое исключение, специфичное для библиотеки в одной ветке `except`:

```
>>> try:
...     do_something()
... except CSCException:
...     # ...
```

Интерфейс исключений в Python довольно нехитрый:

- атрибут `args` хранит кортеж аргументов, переданных конструктору исключения,
- атрибут `__traceback__` содержит информацию о стеке вызовов на момент возникновения исключения.

```
>>> try:
...     1 + "42"
... except Exception as e:
...     caught = e
...
>>> caught.args
("unsupported operand type(s) for +: 'int' and 'str'",)
>>> caught.__traceback__
<traceback object at 0x10208d148>
>>> import traceback
>>> traceback.print_tb(caught.__traceback__)
  File "<stdin>", line 2, in <module>
```

Оператор raise

- Поднять исключение можно с помощью оператора **raise**:

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

- Аргумент оператора **raise** должен наследоваться от базового класса **BaseException**:

```
>>> raise 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must derive from BaseException
```

- Если вызвать оператор **raise** без аргумента, то он поднимет последнее пойманное исключение или, если такого исключения нет, **RuntimeError**.

```
>>> raise
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: No active exception to reraise
```

Оператор `raise from`

```
>>> try:
...     {}["foobar"]
... except KeyError as e:
...     raise RuntimeError("Ooops!") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'foobar'

The [...] exception was the [...] cause of the following [...]:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Ooops!
```

Обработка исключений: `try...finally`

- Иногда требуется выполнить какое-то действие вне зависимости от того, произошло исключение или нет, например, закрыть файл:

```
>>> try:
...     handle = open("example.txt", "wt")
...     try:
...         do_something(handle)
...     finally:
...         handle.close()
... except IOError as e:
...     print(e, file=sys.stderr)
```

- Аналогичным образом нужно работать с любыми другими ресурсами: сетевыми соединениями, примитивами синхронизации.

Обработка исключений: `try...else`

- С помощью ветки **`else`** можно выполнить какое-то действие в ситуации, когда внутри **`try`** блока не возникло исключения:

```
>>> try:
...     handle = open("example.txt", "wt")
... else:
...     report_success(handle)
... except IOError as e:
...     print(e, file=sys.stderr)
```

- Чем использование **`else`** лучше следующего варианта?

```
>>> try:
...     handle = open("example.txt", "wt")
...     report_success(handle)
... except IOError as e:
...     print(e, file=sys.stderr)
```

Цепочки исключений: `except` и оператор `raise`

```
>>> try:
...     {}["foobar"]
... except KeyError:
...     "foobar".split("")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'foobar'

During handling of [...] exception, [...] exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: empty separator
```

Цепочки исключений: `finally` и оператор `raise`

```
>>> try:
...     {}["foobar"]
... finally:
...     "foobar".split("")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'foobar'

During handling of [...] exception, [...] exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: empty separator
```

- Механизм обработки исключений в Python похож на аналогичные конструкции в C++ и Java, но Python расширяет привычную пару **`try`**`...`**`except`** веткой **`else`**.
- Поднять исключение можно с помощью оператора **`raise`**, его семантика эквивалентна `throw` в C++ и Java.
- В Python много встроенных типов исключений, которые можно и нужно использовать при написании функций и методов.
- Для объявления нового типа исключения достаточно унаследоваться от базового класса **`Exception`**.
- Два важных правила при работе с исключениями:
  - минимизируйте размер ветки **`try`**,
  - всегда старайтесь использовать наиболее специфичный тип исключения в ветке **`except`**.

Менеджеры контекста

Зачем нужны менеджеры контекста?

- Менеджеры контекста позволяют компактно выразить уже знакомый нам паттерн управления ресурсами:

```
>>> r = acquire_resource()
... try:
...     do_something(r)
... finally:
...     release_resource(r)
```

- С помощью менеджера контекста пример выше можно записать так:

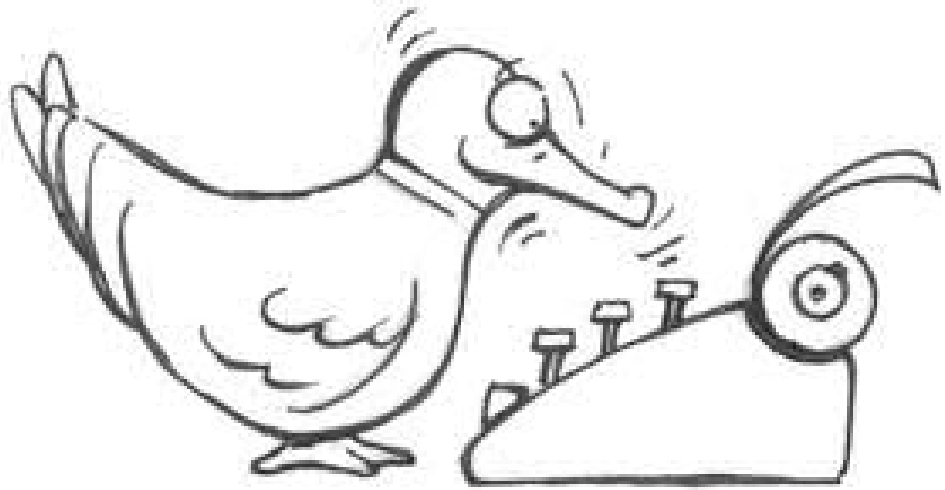
```
>>> with acquire_resource() as r:
...     do_something(r)
```

  Действие `release_resource` будет выполнено автоматически, вызывать его явно не нужно.

- Протокол менеджеров контекста состоит из двух методов.
  - Метод `__enter__` инициализирует контекст, например, открывает файл или захватывает мьютекс. Значение, возвращаемое методом `__enter__`, записывается по имени, указанному после оператора **`as`**.
  - Метод `__exit__` вызывается после выполнения тела оператора **`with`**. Метод принимает три аргумента:
    1. тип исключения,
    2. само исключение и
    3. объект типа `traceback`.

    Если в процессе исполнения тела оператора **`with`** было поднятно исключение, метод `__exit__` может подавить его, вернув **`True`**.
- Экземпляр **любого** класса, реализующего эти два метода, является менеджером контекста.

Duck Typing[2]



[2] http://theregister.co.uk/Print/2007/05/06/fables

- Напоминание:

```
>>> with acquire_resource() as r:
...     do_something(r)
```

- Процесс исполнения оператора **with** можно концептуально записать так:

```
>>> manager = acquire_resource()
>>> r = manager.__enter__()
>>> try:
...     do_something(r)
... finally:
...     exc_type, exc_value, tb = sys.exc_info()
...     suppress = manager.__exit__(exc_type,
...                                 exc_value, tb)
...     if exc_value is not None and not suppress:
...         raise exc_value
```

Расширенные возможности оператора with

- Оператор **with** позволяет работать с несколькими контекстными менеджерами одновременно:

```
>>> with acquire_resource() as r, \
...       acquire_other_resource() as other:
...     do_something(r, other)
```

Такая запись эквивалентна двум вложенным менеджерам контекста:

```
>>> with acquire_resource() as r:
...     with acquire_other_resource() as other:
...         do_something(r, other)
```

- Можно также использовать оператор **with** без указания имени переменной:

```
>>> with acquire_resource():
...     do_something()
```

```
>>> from functools import partial
>>> class opened:
...     def __init__(self, path, *args, **kwargs):
...         self.opener = partial(open, path,
...                               *args, **kwargs)
...
...     def __enter__(self):
...         self.handle = self.opener()
...         return self.handle
...
...     def __exit__(self, *exc_info):
...         self.handle.close()  # Почему можно обойтись
...         del self.handle      # без return?
...
>>> with opened("./example.txt", mode="rt") as handle:
...     pass
```

Капитан сообщает

opened интересен только в качестве примера, потому что файлы в Python уже поддерживают протокол менеджеров контекста.

Примеры менеджеров контекста: модуль `tempfile`

- Модуль `tempfile` реализует классы для работы с временными файлами.
- Все классы реализуют протокол менеджеров контекста, которые работают так же, как и для обычных файлов.
- Интересный пример — класс `TemporaryFile`, который автоматически удаляет временный файл при выходе из менеджера контекста:

```
>>> import tempfile
>>> with tempfile.TemporaryFile() as handle:
...     path = handle.name
...     print(path)
...
/var/folders/nj/T/tmptpy6nn5y
>>> open(path)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory
```

Примеры менеджеров контекста: syncronized

```
>>> class synchronized:
...     def __init__(self):
...         self.lock = threading.Lock()
...
...     def __enter__(self):
...         self.lock.acquire()
...
...     def __exit__(self, *exc_info):
...         self.lock.release()
...
>>> with synchronized():
...     do_something()
```

Капитан сообщает

Большая часть примитивов синхронизации в Python, включая класс `Lock`, реализует протокол менеджера контекста. Использовать менеджер `synchronized` не нужно — он интересен только в качестве примера.

Примеры контекстных менеджеров: cd

```
>>> import os
>>> class cd:
...     def __init__(self, path):
...         self.path = path
...
...     def __enter__(self):
...         self.saved_cwd = os.getcwd()
...         os.chdir(self.path)
...
...     def __exit__(self, *exc_info):
...         os.chdir(self.saved_cwd)
...
>>> print(os.getcwd())
./csc/python
>>> with cd("/tmp"):
...     print(os.getcwd())
...
/tmp
```

- Менеджеры контекста — удобный способ управлять жизненным циклом ресурсов в Python.
- Для работы с менеджером контекста используется оператор `with`.
- Менеджером контекста является экземпляр любого класса, реализующего методы `__enter__` и `__exit__`.
- Некоторые встроенные типы, например, файлы и примитивы синхронизации уже поддерживают протокол менеджеров контекста — этим можно и нужно пользоваться при написании кода.

Модуль contextlib

Модуль contextlib: closing

- Менеджер контекста `closing` обобщает логику уже известного нам `opened` на экземпляр любого класса, реализующего метод `close`.
- Реализовать `closing` самому несложно, но приятно, когда в стандартной библиотеке языка есть и такие мелочи.
- С помощью `closing` можно, например, безопасно работать с HTTP ресурсами:

```
>>> from contextlib import closing
>>> from urllib.request import urlopen
>>> url = "http://compscicenter.ru"
>>> with closing(urlopen(url)) as page:
...     do_something(page)
```

Модуль contextlib: redirect_stdout

- Менеджер контекста `redirect_stdout` позволяет локально перехватывать вывод в стандартный поток.
- Пример использования:

```
>>> from contextlib import redirect_stdout
>>> import io
>>> handle = io.StringIO()
>>> with redirect_stdout(handle):
...     print("Hello, World!")
...
>>> handle.getvalue()
'Hello, World!\n'
```

Вопрос

Как можно было бы реализовать `redirect_stdout`?

Модуль contextlib: suppress

- С помощью менеджера контекста suppress можно локального подавить исключения указанных типов:

```
>>> from contextlib import suppress
>>> with suppress(FileNotFoundError):
...     os.remove("example.txt")
```

- Реализация менеджера не хитра:

```
>>> class supress:
...     def __init__(self, *suppressed):
...         self.suppressed = suppressed
...
...     def __enter__(self):
...         pass
...
...     def __exit__(self, exc_type, exc_value, tb):
...         return (exc_type is not None and
...                 issubclass(exc_type, suppressed))
```

- При использовании suppress, как и в целом при работе с исключениями, стоит указывать наиболее специфичный тип исключения.

- Базовый класс `ContextDecorator` позволяет объявлять менеджеры контекста, которые можно использовать как декораторы.
- Зачем это нужно?

```
def f():
    with context():
        # ...
```

```
@context()
def f():
    # ...
```

- Переход к синтаксису декораторов:
  - подчеркивает, что менеджер контекста применяется ко всему телу функции,
  - позволяет сэкономить 4 пробела :)

Вопрос

Как должен быть реализован менеджер контекста, чтобы его можно было использовать в качестве декоратора?

Модуль contextlib: менеджеры контекста и декораторы

- Для того, чтобы менеджер контекста можно было использовать как декоратор, достаточно унаследовать его от `ContextDecorator`.
- Модифицируем менеджер `suppress` из модуля `contextlib`, чтобы с помощью него можно было подавлять исключения во всей функции:

```
>>> from contextlib import (suppress as _suppress,
...                         ContextDecorator)
>>> class suppressed(_suppress, ContextDecorator):
...     pass
...
>>> @suppressed(IOError)
... def do_something():
...     pass
```

Модуль contextlib: ExitStack

- Что делать, если количество ресурсов может быть произвольным? Например:

```
>>> def merge_logs(output_path, *logs):
...     handles = open_files(logs)
...     with open(output_path, "wt") as output:
...         merge(output, handles)
...     close_files(logs)
```

- Правильный ответ: ExitStack. Менеджер ExitStack позволяет управлять произвольным количеством менеджеров контекста:

```
>>> from contextlib import ExitStack
>>> def merge_logs(output_path, *logs):
...     with ExitStack() as stack:
...         handles = [stack.enter_context(open(log))
...                    for log in logs]
...         output = open(output_path, "wt")
...         stack.enter_context(output)
...         merge(output, handles)
```

“Семантика” менеджера ExitStack

- Менеджер `ExitStack` поддерживает стек вложенных менеджеров контекста:

```
>>> with ExitStack() as stack:
...     stack.enter_context(some_resource)
...     stack.enter_context(other_resource)
...     do_something(some_resource, other_resource)
```

- При выходе из контекста, `ExitStack` обходит список вложенных менеджеров контекста в обратном порядке и вызывает у каждого менеджера метод `__exit__`.
- Менеджер `ExitStack` корректно обрабатывает ситуации,
  - когда метод `__exit__` подавил исключение
  - или, когда в процессе работы метода `__exit__` возникло новое исключение.

Модуль `contextlib`: резюме

- Модуль `contextlib` содержит функции и классы, украшающие жизнь любителя менеджеров контекста.
- Мы поговорили про:
  - `closing`,
  - `redirect_stdout`,
  - `suppress`,
  - `ContextDecorator`,
  - `ExitStack`.