Лекция 9. ¶

Исключения и менеджеры контекста

https://repl.it/data/classrooms/share/a2f0457abdbf0f72cd99ba6954ddf0a7
(https://repl.it/data/classrooms/share/a2f0457abdbf0f72cd99ba6954ddf0a7)

Исключения

- Зачем нужны исключения
- Иерархия исключений в python
- Обработка исключения
- Интерфейс исключений
- Пользовательские исключения
- Оператор raise, цепочки исключения

Зачем нужны исключения?

Исключения нужны для исключительных ситуаций, это ошибки, которые можно обрабатывать.
Например...

```
In [8]: [0] * int(1e16)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-8-ecc9860f6c0b> in <module>
----> 1 [0] * int(1e16)

MemoryError:
```

```
In [9]: import foobar
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-9-cc8dfa1d0002> in <module>
----> 1 import foobar

ModuleNotFoundError: No module named 'foobar'
```

```
In [10]: [1, 2, 3] + 4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-5842ff442cc5> in <module>
----> 1 [1, 2, 3] + 4

TypeError: can only concatenate list (not "int") to list
```

Встроенные исключения

https://docs.python.org/3/library/exceptions.html#exception-hierarchy (https://docs.python.org/3/library/exceptions.html#exception-hierarchy)

BaseException

Базовый класс для встроенных исключений в Python. 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*.

```
In [11]: BaseException.__subclasses__()
```

```
Out[11]: [Exception, GeneratorExit, SystemExit, KeyboardInterrupt]
```

AssertionError

Возникает, когда условие оператора *assert* не выполняется.
Оператор *assert* используется для ошибок, которые могут возникнуть только в результате ошибки программиста, поэтому перехватывать **AssertionError** считается дурным тоном.

```
In [12]: assert 2 + 2 == 5, "Math still works!!!"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-12-d083d52b60d3> in <module>
----> 1 assert 2 + 2 == 5, "Math still works!!!"

AssertionError: Math still works!!!
```

ImportError

```
In [13]: import foobar
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-13-cc8dfa1d0002> in <module>
----> 1 import foobar

ModuleNotFoundError: No module named 'foobar'
```

NameError

```
In [14]: foobar
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-988881adc9fc> in <module>
----> 1 foobar

NameError: name 'foobar' is not defined
```

AttributeError

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

```
In [15]: object().foobar
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-f8fac2a2a97d> in <module>
----> 1 object().foobar

AttributeError: 'object' object has no attribute 'foobar'
```

LookupError

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

```
In [16]: {}['foobar']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-16-74d6dc1e238e> in <module>
----> 1 {}['foobar']

KeyError: 'foobar'
```

```
In [17]: [][0]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-212bc78bf802> in <module>
----> 1 [][0]

IndexError: list index out of range
```

ValueError

Используется в случаях, когда другие более информативные исключения, например, *KeyError*, неприменимы.

```
In [18]: 'foobar'.split('')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-18-3a266ee364dc> in <module>
----> 1 'foobar'.split('')

ValueError: empty separator
```

TypeError

Возникает, если оператор, функция или метод вызываются с аргументом несоответствующего типа.

```
In [19]: b'foo' + 'bar'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-62b5b4737107> in <module>
----> 1 b'foo' + 'bar'

TypeError: can't concat str to bytes
```

Обработка исключений

try...except

Для обработки исключений в Python используются операторы **try** и **except**. 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.

In [ ]:
```
try:
    1/0
except (ValueError, ArithmeticError):
    pass
except TypeError as e:
    pass
except Exception as e:
    pass
```

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

In [ ]:
```
try:
    something_dangerous()
except Exception as e:
    try:
        something_else()
    except type(e): # Какое исключение мы перехватим?
        pass
```

Время жизни переменной *e* ограничивается веткой **except**.

In [20]:
```
try:
    1 + "42"
except TypeError as e:
    pass # Что делать, если нам нужно e?

e
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-5622d1de4a92> in <module>
      4     pass # Что делать, если нам нужно e?
      5
----> 6 e

NameError: name 'e' is not defined
```

try...finally

Иногда требуется выполнить какое-то действие вне зависимости от того, произошло исключение или нет, например, закрыть файл, сетевое соединение, примитив синхронизации, etc...

In [21]:
```
handle = open("example.txt", "wt")
try:
    print(f'{handle.name} closed? {handle.closed}')
    raise
finally:
    handle.close()
```

```
example.txt closed? False

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-21-510047c0bae2> in <module>
      2 try:
      3     print(f'{handle.name} closed? {handle.closed}')
----> 4     raise
      5 finally:
      6     handle.close()

RuntimeError: No active exception to reraise
```

In [22]:
```
print(f'{handle.name} closed? {handle.closed}')
```

```
example.txt closed? True
```

try...else

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

In [23]:
```
try:
    handle = open('example.txt', 'wt')
except IOError as e:
    print(e, file=sys.stderr)
else:
    print('Success open')
    handle.close()
```

```
Success open
```

In [ ]:
```
# Чему лучше такого варианта?
try:
    handle = open('example.txt', 'wt')
    print('Success open')
    handle.close()
except IOError as e:
    print(e, file=sys.stderr)
```

Полная форма

In [ ]:
```
try:
    1/0
except ImportError:
    print('ImportError occured')
except ZeroDivisionError:
    print('It\'s Zero Division!')
except ValueError:
    print('Maybe ValueError??')
else:
    print('No exception at all!!!')
finally:
    print('Bye')
```

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.

In [24]:
```
class BaseLibraryException(Exception):
    pass
```

In [25]:
```
class SpecificException(BaseLibraryException):
    def __str__(self):
        return 'My custom exception'
```

In [26]:
```
try:
    raise SpecificException
except BaseLibraryException as e:
    print(f'Caught `{e}`')
```

```
Caught `My custom exception`
```

Интерфейс исключений

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

In [27]:
```
try:
    1 + "42"
except Exception as e:
    caught = e
```

In [28]:
```
caught.args
```

Out[28]:
```
("unsupported operand type(s) for +: 'int' and 'str'",)
```

In [29]:
```
caught.__traceback__
```

Out[29]:
```
<traceback at 0x247f468b048>
```

In [30]:
```
import traceback
traceback.print_tb(caught.__traceback__)
```

```
  File "<ipython-input-27-00129d968d0d>", line 2, in <module>
    1 + "42"
```

Оператор raise

In [31]:
```
raise TypeError('type mismatch')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-e5c84622989a> in <module>
----> 1 raise TypeError('type mismatch')

TypeError: type mismatch
```

Если вызвать оператор *raise* без аргумента, то он поднимет последнее пойманное исключение.

In [32]:
```
try:
    1 / 0
except Exception:
    print('Caught something!')
    raise
```

```
Caught something!

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-32-3f0c901ead69> in <module>
      1 try:
----> 2     1 / 0
      3 except Exception:
      4     print('Caught something!')
      5     raise

ZeroDivisionError: division by zero
```

Если такого исключения нет, то **RuntimeError**

In [33]:
```
raise
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-33-9c9a2cba73bf> in <module>
----> 1 raise

RuntimeError: No active exception to reraise
```

Оператор raise from

In [34]:
```
try:
    {}["foobar"]
except KeyError as e:
    raise RuntimeError("Ooops!") from e
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-34-6fb98b014d1e> in <module>
      1 try:
----> 2     {}["foobar"]
      3 except KeyError as e:

KeyError: 'foobar'

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

RuntimeError                              Traceback (most recent call last)
<ipython-input-34-6fb98b014d1e> in <module>
      2     {}["foobar"]
      3 except KeyError as e:
----> 4     raise RuntimeError("Ooops!") from e

RuntimeError: Ooops!
```

Цепочки исключений

In [35]:
```
try:
    {}['foobar']
except KeyError:
    'foobar'.split('')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-35-347fa0ac2db9> in <module>
      1 try:
----> 2     {}['foobar']
      3 except KeyError:

KeyError: 'foobar'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-35-347fa0ac2db9> in <module>
      2     {}['foobar']
      3 except KeyError:
----> 4     'foobar'.split('')

ValueError: empty separator
```

Исключения - резюме

- Механизм обработки исключений в Python похожна 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: https://docs.python.org/3/library/exceptions.html (https://docs.python.org/3/library/exceptions.html)
- Для объявления нового типа исключения достаточно унаследоваться от базового класса **Exception**.
- Два важных правила при работе с исключениями:
  1. минимизируйте размер ветки try,
  2. всегда старайтесь использовать наиболее специфичный тип исключения в ветке except.

Задача 1

Даны два значения: a, b. Необходимо выполнить деление a/b и вывести ответ.
Все взаимодействие происходит через **input** пользователя.
Пользователь вводит сначала число пар которые он будет вводить. Затем вводит все пары через пробел.
Пример взаимодействия:

```
 3
1 0
Error code: division by zero
9 $
Error code: invalid literal for int() with base 10: '$'
6 2
3
```

Менеджеры контекста

- Зачем нужны менеджеры контекста
- Протокол менеджеров контекста
- Модуль *contextlib*

Зачем нужны менеджеры контекста?

In [6]:
```
# Без контекстных менеджеров
out = open('output.txt', 'w')

for i in range(10000):
    if i == 5000:
        raise
    print(f'Line number {i}', file=out)

out.close()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-6-a15f0e15a18c> in <module>
      4 for i in range(10000):
      5     if i == 5000:
----> 6         raise
      7     print(f'Line number {i}', file=out)
      8

RuntimeError: No active exception to reraise
```

In [7]:
```
!tail -n 3 ./output.txt
```

```
Line number 4617
Line number 4618
Line number 4619
```

In [36]:
```
# С контекстными менеджерами
with open('output1.txt', 'w') as out:
    for i in range(10000):
        if i == 5000:
            raise
        print(f'Line number {i}', file=out)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-36-00f2d23aa434> in <module>
      3     for i in range(10000):
      4         if i == 5000:
----> 5             raise
      6         print(f'Line number {i}', file=out)

RuntimeError: No active exception to reraise
```

In [37]:
```
!tail -n 3 output1.txt
```

```
Line number 4997
Line number 4998
Line number 4999
```

In [ ]:
```
!rm output.txt output1.txt
```

Протокол менеджеров контекста

Протокол менеджеров контекста состоит из двух методов.

1. Метод **__enter__** инициализирует контекст, например, открывает файл или захватывает мьютекс. Значение, возвращаемое этим методом, записывается по имени, указанному после оператора **as**.
2. Метод **__exit__** вызывается после выполнения тела оператора **with**. Метод принимает три аргумента:

    - тип исключения,
    - само исключение и
    - 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.

In [38]:
```
from functools import partial

class CustomOpen:
    def __init__(self, path, *args, **kwargs):
        self.open_func = partial(open, path, *args, **kwargs)
        print(f'Init CM for {path}')

    def __enter__(self):
        self.handler = self.open_func()
        print(f'Opened {self.handler.name}')
        return self.handler

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.handler.close()
        print(f'Closed {self.handler.name}')
        del self.handler
```

In [39]:
```
with CustomOpen('./example.txt', 'w') as f:
    f.write('Hello World!!!')

print(f'{f.name} closed? {f.closed}')
```

```
Init CM for ./example.txt
Opened ./example.txt
Closed ./example.txt
./example.txt closed? True
```

In [40]:
```
!cat example.txt
```

```
Hello World!!!
```

Процесс исполнения оператора with

In [ ]:
```
with CustomOpen('./example.txt', 'w') as f
    f.write('Hello World!!!')
```

In [ ]:
```
manager = CustomOpen('./example.txt', 'w')
f = manager.__enter__()

try:
    f.write('Hello World!!!')
finally:
    import sys
    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
```

Вложенные менеджеры контекста

In [41]:
```
with CustomOpen('./example1.txt', 'w') as f1, \
    CustomOpen('./example2.txt', 'w') as f2:
    print('Inside managers')
```

```
Init CM for ./example1.txt
Opened ./example1.txt
Init CM for ./example2.txt
Opened ./example2.txt
Inside managers
Closed ./example2.txt
Closed ./example1.txt
```

In [42]:
```
with CustomOpen('./example3.txt', 'w'):
    with CustomOpen('./example4.txt', 'w'):
        print('Inside managers')
```

```
Init CM for ./example3.txt
Opened ./example3.txt
Init CM for ./example4.txt
Opened ./example4.txt
Inside managers
Closed ./example4.txt
Closed ./example3.txt
```

Примеры

- open
- tempfile
- threading.Lock

Задача 2

Написать контекстный менеджер **cd**, который меняет текущую директорию на заданную.
При входе в контекст нужно запомнить прежнюю директорию и при выходе восстановить ее.

Модуль contextlib

https://docs.python.org/3/library/contextlib.html (https://docs.python.org/3/library/contextlib.html)

contextlib.contextmanager

In [43]:
```
import contextlib

@contextlib.contextmanager
def CustomOpenCL(path, *args, **kwargs):
    f = open(path, *args, **kwargs)

    try:
        print('Enter context')
        yield f
    finally:
        print('Exit context')
        f.close()


with CustomOpenCL('./more_example.txt', 'w'):
    print('Inside context')
```

```
Enter context
Inside context
Exit context
```

contextlib.ContextDecorator

In [44]:
```
class MyContextDecorator(contextlib.ContextDecorator):
    def __init__(self, no):
        self.no = no

    def who_am_i(self):
        return f'I\'m number {self.no}'

    def __enter__(self):
        print(f'Before #{self.no}')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f'After #{self.no}')
```

In [45]:
```
@MyContextDecorator(54)
def managed_func():
    print('Managed code')
```

In [ ]:
```
managed_func()
```

Встроенные contextlib менеджеры

In [46]:
```
with contextlib.redirect_stdout(open('out.txt', 'w')):
    help(pow)
```

In [47]:
```
!cat out.txt
```

```
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three argument
s)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

In [48]:
```
from urllib.request import urlopen

with contextlib.closing(urlopen('https://www.python.org')) as page:
    print(page.info())
```

```
Connection: close
Content-Length: 48921
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur
Via: 1.1 varnish
Accept-Ranges: bytes
Date: Sun, 14 Jun 2020 17:12:01 GMT
Via: 1.1 varnish
Age: 2347
X-Served-By: cache-bwi5138-BWI, cache-ams21043-AMS
X-Cache: HIT, HIT
X-Cache-Hits: 3, 5
X-Timer: S1592154721.413468,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

In [49]:
```
import os

with contextlib.suppress(FileNotFoundError):
    os.remove('somefile.tmp')
```

Менеджеры контекста - резюме

- Менеджеры контекста - важный и удобный паттерн при работе с ресурсами, которые нужно правильно освобождать
- Менеджером контекста является любой объект, который реализует 2 метода: **__enter__** и **__exit__**
- В стандартной библиотеке Python а также в сторонних библиотеках много менеджеров контекста
- Модуль **contextlib** стандартной библиотеки делает создание своих менеджеров простым и приятным, а также содержит много готовых менеджеров

Домашнее задание

Задание 1

Реализовать контекстный менеджер - аналог **tempdir**.

1. При входе в контекст создается директория с уникальным именем.
2. Вся дальнейшая работа ведется в этой директории (она становится текущей).
3. При выходе из контекста директория удаляется вместе со всеми файлами в ней.
4. Рабочей директорией становиться та, что была до входа в контекст.

Использовать протокол менеджеров контекста (реализовать методы `__enter__` и `__exit__`).
Продемонстрировать работу своего менеджера: пока находимся в его контексте, пишем что-нибудь на диск, после выхода - проверяем, что все подчистилось без каких-то дополнительных команд.

Задание 2

Реализовать контекстный менеджер, выводящий в файл следующую информацию:

- дата
- время выполнения кода
- информация о возникшей ошибке (в коде, обернутом контекстным менеджером).

Файл указать при конструировании менеджера.
Файл открывается в режиме `append`, чтобы при вызове менеджера с одним и тем же файлом информация дописывалась (такой самописный лог).
Выше ошибка прокидывается (происходит reraise).
Используйте ContextDecorator для решения.

Задание 3*

Пересмотрите задания, которые вы выполняли ранее в курсе, свои пет-проекты, курсовые и т.п.
Найдите фрагменты, которые вы бы теперь переписали с использованием менеджеров контекста (если раньше их не использовали)?
Переписывать код не нужно, достаточно объяснить преподавателю, почему контекстные менеджеры здесь - то, что нужно.
Если вы и раньше использовали их в своем коде, тоже покажите!