Занятие 6: Обработка исключений. Менеджеры контекстов. Тестирование.

Практикум на ЭВМ 2021/2022

Афанасьев Глеб Ильич

МГУ имени М. В. Ломоносова, факультет ВМК, кафедра ММП

4 октября 2021 г.

Введение

В больших проектах без тестирования жить сложно:

+ Дополнительные проверки корректности кода
+ Быстрая проверка корректности при изменениях в коде
– Требуется время на написание
– Код для тестов может быть длиннее кода программы

Исключения

При написании программы часто возникают такие ситуации, когда успешное ее выполнение зависит от внешних условий, которые программист гаранитировать не может. Для обработки таких ситуаций был создан механизм исключений.
Формальное определение:
Исключения (exceptions) — проблемы, возникающие в ходе выполнения программы, приводящие к невозможности дальнейшей отработки программой её базового алгоритма.

Обработка исключений — механизм, предназначенный для описания реакции программы на исключения

Исключения

В Python исключения(как и все остальное) являются объектами. В C++ в качестве исключения мог выступать любой тип, в python - нет. Все исключения - объекты класса, отнаследованного от класса `Exception`.
Через них можно получить информацию об ошибке:

```
>>> 1 / 0
ZeroDivisionError: division by zero
>>> ZeroDivisionError.__doc__ # документация к исключению
'Second argument to a division or modulo operation was zero.'
>>> ZeroDivisionError.__mro__
(ZeroDivisionError, ArithmeticError, Exception,
BaseException, object)
```

Примеры исключений

- `BaseException` — базовый класс для встроенных исключений в Python (наследуются системные исключения и исключения, приводящие к завершению работы интерпретатора)
- `Exception` — более общий класс, наследуются все остальные исключения
- `AssertionError` — поднимается, если не выполнено условие оператора `assert`

```
>>> assert 2 * 2 == 5, "Wrong equation"
AssertionError: Wrong equation
```

- `ImportError` — поднимается, если оператор `import` не смог найти модуль с указанным именем

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

Связка try ... except

Связка try ... except позволяет перехватывать исключения, возбужденные интерпретатором или программным кодом, и выполнять восстановительные операции:

```
>>> def one(x):
...     try:
...         print(x / x)
...     except ZeroDivisionError: # перехватывается конкретное
...         print('We did it!')          # исключение
...     except:                          # перехватываются все
...         print('Something is wrong') # исключения
...     # остальная часть программы
...
>>> one(5)
1
>>> one(0)
'We did it!'
>>> one('not a number')
'Something is wrong'
```

Связка try ... except

```
>>> def one(x):
...     try:
...         print(x / x)
...     except ZeroDivisionError:
...         print('We did it!')
...     except Exception as e: # е живёт только внутри блока
...         print('Something is wrong')
...         return e
...     # остальная часть программы
...
>>> my_exception = one('not a number')
'Something is wrong'
>>> my_exception.args
("unsupported operand type(s) for /: 'str' and 'str'",)
```

Связка try ... else

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

```
>>> import sys
...
>>> try:
...     handle = open("example.txt", "r")
>>> except IOError as e:
...     print(e, file=sys.stderr)
>>> else:
...     print('File was opened')
```

Ключевое слово finally

С помощью ветки finally можно выполнить какое-то действие вне зависимости от того, произошло исключение или нет.

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

Ключевое слово raise

`raise` позволяет возбудить исключение программно:

```
>>> def one(x):
...     if type(x) != int:
...         raise TypeError('My message') # любое сообщение
...     return x / x
...
>>> one('string')
TypeError: My message
```

Можно создавать пользовательские исключения:

```
>>> class MyError(Exception):
...     pass
```

Еще про исключения

except ловит лишь исключения, поднятые в своем блоке или блоках, вложенных в него.

```
>>> try:
...     g = 4
...     raise Exception("MyMessage1") #поднимаем исключение
                                      #Exeption
>>> except TypeError as e: #пытаемся поймать TypeError,
                           #не ловим
...     print("Something is wrong!1")
>>> finally:
...     print("Finally1")

>>> try:
...     #код
>>> except Exception as e:
...     print("Something is wrong!2")
>>> finally:
...     print("Finally2")
```

Еще про исключения

except ловит лишь исключения, поднятые в своем блоке или блоках, вложенных в него.

Здесь исключение будет поймано.

```
>>> try:
...     #код
...     try:
...         g = 4
...         raise Exception("MyMessage1")
...     except TypeError as e:
...         print("Something is wrong!1")
...     finally:
...         print("Finally1")
>>> except Exception as e:
...     print("Something is wrong!2")
>>> finally:
...     print("Finally2")
```

Советы по использованию исключений

- В Python много встроенных типов исключений, которые можно и нужно использовать при написании функций и методов
- При объявлении нового типа исключения нужно наследоваться от базового класса Exception
- При создании большого количества пользовательских исключений, наследуйте их от одного базового пользовательского исключения

Что такое менеджер контекста?

Менеджеры контекста позволяют компактно выразит паттерн управления ресурсами:

```
>>> r = get_resource()
>>> try:
...     do_something(r)
>>> finally:
...     free_resourse(r)
```

С помощью менеджера контекста это можно записать так:

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

Протокол менеджеров контекста

Протокол менеджеров контекста состоит из двух методов:

- метод `__enter__`
  Инициализация контекста, например, открытие файла.
  Результат записывается в переменную после `as`.
- метод `__exit__`
  Метод вызывается после выполнения тела оператора `with`.
  Имеет три аргмента:
  - тип исключения
  - само исключение
  - объект типа `traceback`

  Если в процессе исполнения тела оператора `with` было поднятно исключение, метод `__exit__` может подавить его, вернув `True`.

Что происходит в операторе with?

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

```
>>> manager = get_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
```

Примеры менеджеров контекста: open

```
from functools import partial
>>> class open:
...     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()
...         del self.handle
...
>>> with opened("./example.txt", mode="r") as handle:
...     pass
```

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
```

Когда использовать менеджеры контекста?

Менеджеры контекста используются для следующих ситуаций:

1. Открытие/закрытие
2. Занятие/освобождение
3. Изменение/откат изменений
4. Начало/конец

Библиотека `contextlib`

- `closing` — безопасное закрытие для любого класса с методом `close`
- `redirect_stdout` — перехват `stdout` в переменную
- `suppress` — подавление исключений указанного вида
- `ExitStack` — открытие недетерминированного числа менеджеров контекста

Юнит-тестирование

Общие принципы:

- Код программы разбит на независимые части
- Каждая часть тестируется отдельно и независимо

Признаки хорошего теста:

- Корректность
- Понятность
- Конкретность (проверяет что-то одно)
- Полезность
- Быстрота выполнения

Функция для тестирования

Функция, которую мы будем тестировать:

```
>>> def factorial(x):
...     if x <= 0:
...         return 0
...     elif x == 1:
...         return 1
...     else:
...         return x * factorial(x - 1)
```

Самый простой вариант тестирования:

```
>>> print(factorial(3))
6
```

Тестирование вручную

Оператор `assert` возбуждает исключение, при невыполнении определенного условия:

```
>>> assert 1 == 2, 'Error' # второй аргумент - любое сообщение
AssertionError: Error
```

Напишем несколько тестов:

```
>>> def test_factorial(): # плохой
...     assert factorial(5) == (lambda n: [1, 0][n > 1] or
...                             (n - 1) * n)(5)
>>> def test_factorial(): # плохой
...     assert factorial(4) == 24, 'Positive numbers'
...     assert factorial(0) == 1, 'Zero'
...     try:
...         factorial(1000)
...     except RecursionError:
...         assert False, 'Recursion problem'
```

Тестирование вручную

```
>>> def test_factorial(): # плохой
...     assert factorial(5) != factorial(12)
```

```
>>> def test_factorial(): # плохой
...     assert factorial('string') != factorial(12)
```

```
>>> def test_factorial(): # хороший
...     assert factorial(0) == 1, 'Zero'
```

Тестирование вручную:

+ Быстро и легко писать
– Надо запускать вручную

Модуль unittest

unittest — фреймворк для автоматизации тестов.
Тест — метод экземпляра наследника unittest.TestCase, начинающийся на test_.

```
>>> # модуль test_factorial.py
>>> import unittest
>>> from my_module import factorial
...
>>> class Test_factorial(unittest.TestCase):
...     def test_factorial_positive(self):
...         self.assertEqual(factorial(5), 120)
...
...     def test_factorial_zero(self):
...         self.assertEqual(factorial(0), 5) # специально!
...
>>> if __name__ == '__main__':
...     unittest.main() # запускает все тесты модуля
```

Результат работы unittest

```
python -m unittest test_factorial.py

.F
======================================================================
FAIL: test_factorial_zero (test_factorial.Test_factorial)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_factorial.py", line 10, in test_factorial_zero
    self.assertEqual(factorial(0), 5)
AssertionError: 1 != 5

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Контекст тестов в unittest

В unittest можно запускать тесты с контекстом:

```
>>> # модуль test_factorial.py
>>> import unittest
>>> from my_module import factorial
...
>>> class Test_factorial(unittest.TestCase):
...     def setUp(self): # вызывается перед каждым тестом
...         self.base = open('file_with_examples')
...
...     def test_factorial_positive(self):
...         self.assertEqual(factorial(5), 120)
...
...     def tearDown(self): # вызывается после каждого теста
...         self.base.close()
...
>>> if __name__ == '__main__':
...     unittest.main()
```

Функции для проверки

Модуль unittest предоставляет множество функций для самых различных проверок, например:

- `assertIs(a, b)` — `a is b`
- `assertRaises(exc, func, *args, **kwds)` — `func(*args, **kwds)` порождает исключение exc

Дополнительные виды проверок можно получить из модуля `numpy.testing`:

- `assert_array_almost_equal(x, y, decimal)` — `x` и `y` совпадают с некоторой точностью

Кратко о `unittest`

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ Много возможностей (см. документацию)
+ Есть в стандартной библиотеке
– Имеет нестандартный, непривычный интерфейс

Модуль py.test

py.test — альтернатива unittest для написания тестов.

У py.test минимальные требования к интерфейсу, тест это:

- функция с именем, начинающимся с test_
- то же, что и у unittest

```
>>> # модуль test_factorial.py
>>> import py.test
>>> from my_module import factorial
...
>>> def test_positive():
...     assert factorial(4) == 24
...
>>> def test_zero():
...     assert factorial(0) == 5 # специально!
```

Результат работы py.test

```
python3 -m pytest test_factorial.py
============================= test session starts ==============
platform linux -- Python 3.5.2, pytest-3.2.1, py-1.4.34
rootdir: /home/user/Programs, inifile:
collected 2 items

test_factorial.py .F

=================================== FAILURES ===================
_______________________________ test_zero ______________________

    def test_zero():
>       assert factorial(0) == 5
E       assert 1 == 5
E        +  where 1 = factorial(0)

test_factorial.py:19: AssertionError
======================== 1 failed, 1 passed in 0.02 seconds =====
```

Контекст тестов в py.test

Для задания контекста необходимо задать функции с специальными названиями setup_ и teardown_ (контекст может задаваться перед запуском модуля или функции):

```
def setup_module(module):
    print ("module setup")

def teardown_module(module):
    print ("module teardown")

def setup_function(function):
    print ("function setup")

def teardown_function(function):
    print ("function teardown")
```

Параметризация тестов в py.test

ОСТОРОЖНО: декоратор!

```
>>> @pytest.mark.parametrize("i, right_answer", [
...     (1, 1),
...     (2, 2),
...     (3, 6)
... ])
>>> def test_factorial(i, right_answer):
...     assert factorial(i) == right_answer
```

Кратко о `py.test`

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ Много возможностей (см. документацию)
+ Простой интерфейс
– Более магический чем `unittest` (но не в наших примерах)

doc.test

doctest позволяет писать тесты внутри документации.

doctest проводит сравнение (как строк!) записанного ответа и вывода интерпретатора на инструкцию:

```
import doctest
def factorial(x):
    """
    Documentation

    >>> factorial(5)
    120

    >>> factorial(0) # специально
    5
    """
    # код функции

doctest.testmod()
```

Результат работы doc.test

```
**********************************************************************
File "__main__", line 8, in __main__.factorial
Failed example:
    factorial(0) # специально
Expected:
    5
Got:
    1
**********************************************************************
1 items had failures:
   1 of   2 in __main__.factorial
***Test Failed*** 1 failures.
```

Кратко о `doc.test`

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ В документации всегда актуальные примеры кода
- Результаты сравниваются только как строки
- Если в середине теста произошла ошибка, оставшиеся тесты не выполнятся

Заключение

В Python большое количество способов тестировать свой код:

- Ручное тестирование с помощью `assert`
- Прописывание простых тестов внутри документации с помощью `doc.test`
- Продвинутые библиотеки `unittest` и `py.test`

Писать тесты к своему коду полезно!