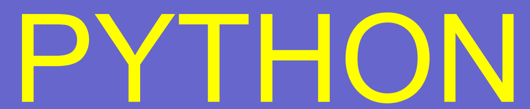

PYTHON

Виключення

План

1. Виключення.
2. Виникнення виключень.
3. Обробка виключень.
4. Синтаксичні помилки.
5. Користувацькі виключення.
6. Попередження.
7. LBYL і EAFP.

Основні поняття механізму виключень

Обробка виняткових ситуацій або **обробка виключень** (exception handling)- механізм мов програмування, призначений для опису реакції програми на помилки часу виконання і інші можливі проблеми (виключення), які можуть виникнути при виконанні програми і призводять до неможливості (безглуздості) подальшого відпрацювання програмою її базового алгоритму.

Основні поняття механізму виключень

Деякі класичні приклади виняткових ситуацій:

- ділення на нуль;
- помилка при спробі зчитати зовнішні дані;
- вичерпання доступної пам'яті.

Виникнення виключень.

- **Python** автоматично генерує виключення при виникненні помилки часу виконання.
- Код на **Python** може згенерувати виключення за допомогою ключового слова **raise**. Після нього вказується об'єкт виключення. Також, можна вказати клас виключення, в такому випадку буде автоматично викликаний конструктор без параметрів. **raise** може викидати в якості винятків тільки екземпляри класу **BaseException** і його спадкоємців, а також (в **Python 2**) екземпляри класів старого типу.

Приклад 1

```
raise Exception('some error occurred')
```

```
line 1, in <module>
raise Exception('some error occurred')
Exception: some error occurred
>>>
```

Виникнення виключень.

- Пам'ятайте, що класи старого типу в **Python 2** існують тільки для зворотної сумісності і використовувати їх не слід.
- Всі стандартні класи виключень в **Python** є класами нового типу і успадковуються від **Exception** або безпосередньо від **BaseException**. Всі призначені для користувача винятку повинні бути спадкоємцями **Exception**.

Обробка виняткових ситуацій в Python

```
try:                     # область дії обробника
    …
except Exception1:       # обробник виключення Exception1
    …
except Exception2:       # обробник виключення Exception2
    …
except:                  # стандартний обробник виключення
    …
else:                    # код, що виконується, якщо ніяке
                         # виключення не виникло
    …
finally:                 # код, що виконується в будь-якому  випадку
    …
```

Блок try

- Блок **try** задає область дії обробника винятків. Якщо при виконанні операторів в даному блоці було викинуто виключення, їх виконання переривається і управління переходить до одного з обробників. Якщо не виникло жодного винятку, блоки **except** пропускаються.

Приклад 2

```
try:
    x = 2 / 0
except ZeroDivisionError:
    print('Division by zero detected')
```

Division by zero detected

>>>

Блок except

```
try:
    pass
except Exception1:
    pass
except (Exception2, Exception3):
    pass
except Exception4 as exception:
    pass
except Exception4, exception:
    pass
except:
    pass
```

Приклад 3

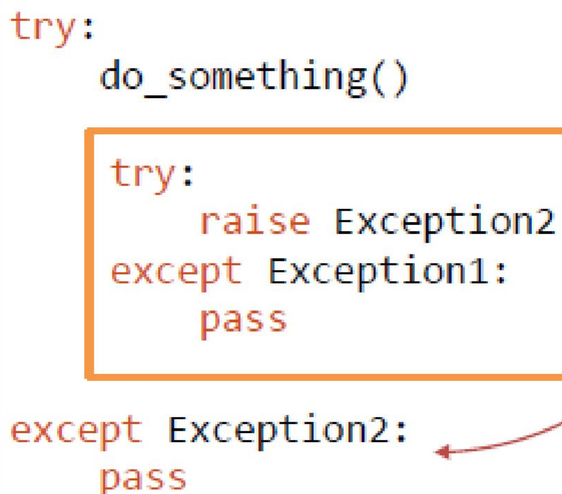
```
def divide_numbers():
    loop = True
    while loop:
        try:
            first_number = float(input('First number: '))
            second_number = float(input('Second number: '))
            print('Result:', first_number / second_number)
            loop = False
        except (ValueError, ZeroDivisionError) as error:
            print('Error:', error)
            print('Please try again')
            print()
            loop = True

if __name__ == '__main__':
    divide_numbers()
```

```
First number: 2.5
Second number: 0
Error: float division by zero
Please try again

First number: 2.5
Second number: 5
Result: 0.5
```

Блок except

Блоки **except** обробляються зверху вниз і управління передається не більше, ніж одному оброблювачу. Тому при необхідності по-різному обробляти виключення, що знаходяться в ієрархії спадкування, спочатку потрібно вказувати обробники менш загальних винятків, а потім - більш загальних. Також саме тому стандартний блок **except** може бути тільки останнім.

Необроблені виключення



- Якщо жоден із заданих блоків **except**t **НЕ** перехоплює виключення, то воно буде перехоплено найближчим зовнішнім блоків **try / except**, в якому є відповідний обробник. Якщо ж програма **НЕ** перехоплює виняток взагалі, то інтерпретатор завершує виконання програми і виводить інформацію про виключення в стандартний потік помилок **sys.stderr**. З цього правила є два винятки:

Приклад 3

```
try:
    try:
        raise ValueError('incorrect value')
        except ZeroDivisionError:
            print('division by zero')
        except Exception as e:
            print(e)
```

RESTART: D:\Оля_2016\Python\Python …… *.py
incorrect value

Необроблені виключення

- Якщо виключення виникло в деструкторі об'єкта, виконання програми не завершується, а в стандартний потік помилок виводиться попередження "**Exception ignored**" з інформацією про виключення.
- При виникненні виключення **SystemExit** відбувається тільки завершення програми без виведення інформації про виключення на екран (не стосується попереднього пункту, в деструкторі поведінку даного виключення буде таким же, як і інших).

Приклад 4

```
class MyClass(object):
    def __del__(self):
        raise ZeroDivisionError


print('Creating object')
obj = MyClass()

print('Deleting object')
del obj

print('Done')
```

Приклад 4

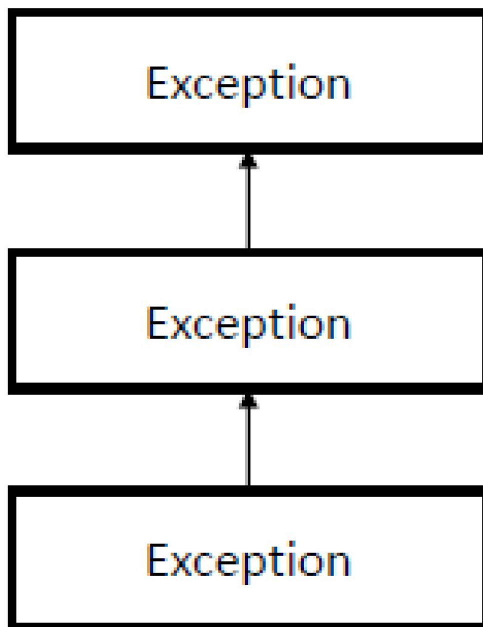
```
RESTART: D:\... *.py
Creating object
Deleting object

Exception ignored in: <bound method MyClass.__del__ of <__main__.MyClass object at 0x02FAF290>>
Traceback (most recent call last):
  File "D:\Оля_2016\Python\Python …*.py", line 3, in __del__
    raise ZeroDivisionError
ZeroDivisionError:

Done
```

Передача виключення на один рівень вище

```
try:
    do_some_actions()
except Exception as exception:
    handle_exception(exception)
    raise
```

Для того, щоб в обробнику виключення виконати певні дії, а потім передати виключення далі, на один рівень обробника вище (тобто, викинути те ж саме виключення ще раз), використовується ключове слово **raise** без параметрів.

Винятки в блоці except. Зчеплення винятків



- В **Python 3** при виникненні виключення в блоці **except** старе виключення зберігається в атрибуті даних **__context__.**
- Для зв'язування винятків використовується конструкція

raise нове_виключення **from** старе_ виключення

або

raise нове_виключення **from** None

Винятки в блоці except. Зчеплення винятків

- У першому випадку вказане виключення зберігається в атрибуті **__cause__** і атрибут **__suppress_context__** (який припиняє вивід виключення з **__context__**) встановлюється в **True**. Тоді, якщо нове виключення оброблено, буде виведена інформація про те, що старе виключення є причиною нового.
- У другому випадку **__suppress_context__** встановлюється в **True** і **__cause__** в **None**. Тоді при виведенні виключення воно, фактично, буде замінено новим (хоча старе виключення все ще зберігається в **__context__**).

Приклад 5

```
a = 5
b = 0

try:
    c = a / b
except ZeroDivisionError as error:
    raise ValueError('variable b is incorrect') from error
```

Приклад 5

```
RESTART: D:\... *.py
Traceback (most recent call last):
  File "D:\Оля_2016\Python\Python …*.py", line 5, in <module>
    c = a / b
ZeroDivisionError: division by zero

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

Traceback (most recent call last):
  File "D:\... *.py", line 7, in <module>
    raise ValueError('variable b is incorrect') from error
ValueError: variable b is incorrect
```

Приклад 6

```
a = 5
b = 0

try:
    c = a / b
except ZeroDivisionError as error:
    raise ValueError('variable b is incorrect') from None
```

Приклад 6

```
RESTART: D:\... *.py
Traceback (most recent call last):
  File "D:\... *.py", line 7, in <module>
    raise ValueError('variable b is incorrect') from None

ValueError: variable b is incorrect
```

Блок else

- Необов'язковий блок.
- Оператори всередині нього виконуються, якщо ніяке виключення не виникло.
- Призначений для того, щоб відокремити код, який може викликати виключення, яке повинно бути оброблено в даному блоці **try / except**, від коду, який може викликати виключення того ж класу, яке повинно бути перехоплено на рівні вище, і звести до мінімуму кількість операторів в блоці **try**.

Приклад 7

```
def divide_numbers():
    while True:
        try:
            first_number = float(input('First number: '))
            second_number = float(input('Second number: '))
            result = first_number / second_number
        except (ValueError, ZeroDivisionError) as error:
            print('Error:', error)
            print('Please try again')
            print()
        else:
            print('Result:', result)
            break


if __name__ == '__main__':
    divide_numbers()
```

```
First number: 2.4
Second number: 0
Error: float division by zero
Please try again

First number: 2.1
Second number: 2.1
Result: 1.0
```

Блок finally

- Оператори всередині блоку **finally** виконуються незалежно від того, чи виникло якесь виключення.
- Призначений для виконання так званих **cleanup actions**, тобто дій з очищення: закриття файлів, видалення тимчасових об'єктів і т.д.
- Якщо виключення виключення не було перехоплено жодним з блоків **except**, то воно заново викидається інтерпретатором після виконання дій в блоці **finally**.
- Блок **finally** виконується перед виходом з оператора **try / except** завжди, навіть якщо одна з його гілок містить оператор **return** (коли оператор **try / except** знаходиться всередині функції), **break** або **continue** (коли оператор **try / except** знаходиться всередині циклу) або виникло інше необроблене виключення при обробці даного виключення.

Приклад 8

```
def function_that_may_fail():
    response = None
    while response != 'y' and response != 'n':
        response = input('Raise an exception? (y/n) ')
    if response == 'y':
        raise Exception


try:
    function_that_may_fail()
except:
    print('Exception handler')
finally:
    print('Finally block')
```

```
Raise an exception? (y/n) y
Exception handler
Finally block

Raise an exception? (y/n) n
Finally block
```

Базові стандартні класи виключень

Клас	Опис
BaseException	Базовий клас для всіх винятків
Exception	Базовий клас для всіх стандартних винятків, які не вказують на обов'язкове завершення програми, і всіх призначених для користувача винятків
ArithmeticError	Базовий клас для всіх винятків, пов'язаних з арифметичними операціями
BufferError	Базовий клас для винятків, пов'язаних з операціями над буфером

Базові стандартні класи виключень

Клас	Опис
LookupError	Базовий клас для винятків, пов'язаних з неправильним ключем і індексом колекції
StandardError (Python 2)	Базовий клас для всіх вбудованих винятків, крім StopIteration, GeneratorExit, KeyboardInterrupt і SystemExit
EnvironmentError (Python 2)	Базовий клас для винятків, пов'язаних з помилками, які відбуваються поза інтерпретатора Python.

Синтаксичні помилки

- Помилка синтаксису виникає, коли синтаксичний аналізатор **Python** стикається з ділянкою коду, який не відповідає специфікації мови і не може бути інтерпретований.
- У головному модулі виникає до початку виконання програми і не може бути перехоплена.
- Ситуації, в яких синтаксична помилка як виняток **SyntaxError** може бути перехоплена і оброблена:
  - синтаксична помилка в імпортованому модулі;
  - синтаксична помилка в коді, який представляється рядком і передається функції **eval** або **exec**.

Попередження

- **Попередження** зазвичай виводяться на екран в ситуаціях, коли не гарантована помилкова поведінка і програма, як правило, може продовжувати роботу, однак користувача слід повідомити про що-небудь.
- Базовим класом для попереджень є **Warning**, який успадковується від **Exception**.
- Базовим класом-спадкоємцем **Warning** для призначених для користувача попереджень є **UserWarning**.
- У модулі **warning** зібрані функції для роботи з попередженнями. Основною є функція **warn**, яка приймає один обов'язковий параметр **message**, який може бути або рядком-повідомленням, або екземпляром класу або підкласу **Warning** (в такому випадку параметр **category** встановлюється автоматично) і два опціональних параметра: **category** (за замовчуванням - **UserWarning**) - клас попередження і **stacklevel** (за замовчуванням - 1) - рівень вкладеності функцій, починаючи з якого необхідно виводити вміст стеку викликів.

LBYL і EAFP

- У статично типізованих мовах компілятор контролює, чи реалізує клас, екземпляром якого є даний об'єкт, певний інтерфейс. При динамічній качиній типізації відповідальність за це лежить на програмістові. Є два протилежні підходи до реалізації таких перевірок.
- **LBYL** (**Look Before You Leap** - «сім разів відміряй, один раз відріж») - стиль, який характеризується наявністю безлічі перевірок і умовних операторів. У контексті качиної типізації може означати перевірку наявності необхідних атрибутів за допомогою функції **hasattr**.
- **EAFP** (**Easier to Ask for Forgiveness than Permission** - «простіше попросити вибачення, ніж дозволу») - стиль, який характеризується наявністю блоків **try / except**. У контексті качиної типізації - написання коду виходячи з припущення, що даний об'єкт реалізує необхідний інтерфейс, і обробка виключення AttributeError в протилежному випадку.

LBYL і EAFP

- **LBYL** і EAFP - це досить загальні стилі написання коду на динамічних мовами, які стосуються не тільки качиної типізації. Наприклад: перевірка існування ключа в словнику (**LBYL**) або обробка виключення **KeyError** (**EAFP**), перевірка існування файлу (**LBYL**) або обробка виключення **IOError** (**EAFP**).
- Переваги стилю **EAFP**:
  - код простіше читається завдяки відсутності зайвих перевірок;
  - виключення в **Python** працюють досить швидко;
  - позбавлений ризику виникнення стану змагальності в багатопотоковому оточенні, що іноді відбувається при використанні підходу **LBYL**.