

Python Essential

Виключення та їх обробка

Python Essential

Після уроку обов'язково



Повторіть цей урок у форматі відео на [ITVDN.com](http://itvdn.com)

Доступ можна отримати через керівництво вашого навчального центру



Перевірте, як Ви засвоїли цей матеріал на [TestProvider.com](http://testprovider.com)

Виключення та їх обробка

Виключення та їх обробка

Концепція механізму винятків

Обробка виняткових ситуацій, або **Обробка винятків** (англ. exception handling) – механізм мов програмування, який призначений для опису реакції програми на помилки часу виконання та інші можливі проблеми (винятки), які можуть виникнути під час виконання програми та призводять до неможливості (безглуздості) подальшого відпрацювання програмою її базового алгоритму.

Деякі класичні приклади виняткових ситуацій:

- ділення на нуль;
- помилка під час спроби порахувати зовнішні дані;
- вичерпання доступної пам'яті.



Виключення та їх обробка

Викид винятків

- Python автоматично генерує винятки у разі виникнення помилки часу виконання.
- Код на Python може згенерувати виняток за допомогою ключового слова **raise**. Після нього зазначається об'єкт винятку. Також можна вказати клас винятку. У такому разі буде автоматично викликаний конструктор без параметрів. **raise** може викидати як виняток лише екземпляри класу **BaseException** та його спадкоємців, а також (у Python 2) екземпляри класів старого типу.
- Пам'ятайте, що класи старого типу Python 2 існують тільки для зворотної сумісності, тому не варто їх використовувати.
- Усі стандартні класи винятків у Python є класами нового типу й успадковуються від **Exception** або безпосередньо від **BaseException**. Усі користувацькі винятки мають бути спадкоємцями **Exception**.



Виключення та їх обробка

Обробка виняткових ситуацій у Python

```
try:                                # зона дії оброблювача
    ...
except Exception1:                  # обробник винятку Exception1
    ...
except Exception2:                  # обробник винятку Exception2
    ...
except:                             # стандартний обробник винятків
    ...
else:                               # код, який виконується, якщо не
    ...                             # виник ніякий виняток
finally:                            # код, який виконується у будь-якому
    ...                             випадку
    ...
```



Виключення та їх обробка

Блок try

Блок **try** задає зону дії оброблювача винятків. Якщо під час виконання операторів у цьому блоці було викинуто виняток, їхнє виконання переривається, а управління переходить до одного з оброблювачів. Якщо не виникло жодного винятку, блоки **except** пропускаються.

```
try:
    if six.PY2:
        register_hstore(connection)
    else:
        register_hstore(connection)
except ProgrammingError:
    # Hstore is not available on this version.
    #
    # If someone tries to create an index on a table
    # This is necessary as someone may have
    # installed but be using other feed
    #
    # ... needed in order to ...
```

Виключення та їх обробка

Блок except

```
try:
    pass
except Exception1:           # обробник винятку Exception1
    pass
except (Exception2, Exception3): # обробник винятку Exception2 та Exception3
    pass
except Exception4 as exception: # обробник винятку Exception4
    pass                     # екземпляр винятку доступний під ім'ям exception
except Exception4, exception:  # застарілий синтаксис, не підтримується в Python 3
    pass
except:                       # стандартний обробник, перехоплює всі винятки
    pass
```



Блоки except обробляються зверху вниз, і керування передається не більш як одному обробнику. Тому за необхідності по-різному обробляти винятки, які розташовані в ієрархії успадкування. Спочатку потрібно вказувати обробники менш загальних винятків, а потім – загальніших. Також саме тому стандартний блок except може бути лише останнім.

Виключення та їх обробка


Необроблені винятки

Якщо жоден із заданих блоків `except` не перехоплює виняток, то він буде перехоплений найближчим зовнішнім блоком `try/except`, в якому є відповідний обробник. Якщо ж програма не перехоплює виняток взагалі, то інтерпретатор завершує виконання програми та виводить інформацію про виняток у стандартний потік помилок `sys.stderr`. У цьому правилі є два винятки:

```
try:
    do_something()

    try:
        raise Exception2
    except Exception1:
        pass

except Exception2:
    pass
```



- Якщо виняток виник у деструкторі об'єкта, виконання програми не завершується, а в стандартний потік помилок виводиться попередження «Exception ignored» з інформацією про виняток.
- Під час виникнення винятку `SystemExit` відбувається лише завершення програми без виведення інформації про виняток на екран (не стосується попереднього пункту, у деструкторі поведінка цього винятку буде такою самою, як і інших).

Виключення та їх обробка

Передання винятку на один рівень вищий

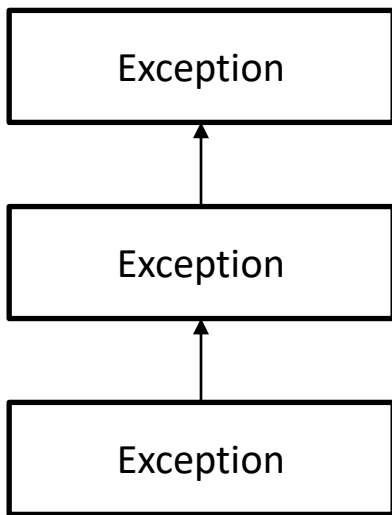
```
try:
    do_some_actions()           # дії, які можуть спричинити виняток
except Exception as exception: # обробник винятку
    handle_exception(exception) # певні дії з цим винятком
    raise                       # згенерувати той самий виняток ще раз
```



Щоб в обробнику винятку виконати певні дії, а потім передати виняток далі, на один рівень обробників вище (тобто викинути той самий виняток ще раз), використовується ключове слово `raise` без параметрів.

Виключення та їх обробка

Винятки у блоці `except`. Зчеплення винятків



- У Python 3 під час викиду винятку в блоці `except` старий виняток зберігається в атрибуті даних `__context__`.
- Для зв'язування винятків використовується конструкція
`raise новий_виняток from старий_виняток`
або
`raise новий_виняток from None`
- У першому випадку зазначений виняток зберігається в атрибуті `__cause__` й атрибут `__suppress_context__` (який пригнічує виведення винятку з `__context__`) встановлюється в `True`. Тоді, якщо новий виняток не опрацьовано, буде виведено інформацію про те, що старий виняток є причиною нового.
- У другому випадку `__suppress_context__` встановлюється в `True` та `__cause__` у `None`. Тоді під час виведення винятку він, фактично, буде замінений на новий (хоча старий виняток все ще зберігається в `__context__`).



**Python 2 не має зв'язування винятків.
Будь-який виняток, викинутий у блоці `except`, замінює старий.**

Виключення та їх обробка

Порівняння raise та assert

```
def validate_email(address):  
    if not "@" in address:  
        raise ValueError("Email Addresses must contain @ sign")  
  
try:  
    validate_email("test.com")  
except ValueError as ex:  
    print("We can do some special invalid input handling here")  
finally:  
    print("Finally always runs whether we succeed or not")
```

```
def validate_email(address):  
    assert "@" in address, "Email Addresses must contain @ sign"  
  
try:  
    validate_email("test.com")  
except Exception as e:  
    print(f'{e.__class__}: {e}')  
finally:  
    print("Finally always runs whether we succeed or not")
```



Виключення та їх обробка

Блок else

- Необов'язковий блок.
- Оператори всередині нього виконуються, якщо жодного винятку не виникло.
- Призначений для того, щоб відокремити код, який може викликати виняток, який має бути оброблений в цьому блоці try/except від коду, який може викликати виняток того ж класу, який має бути перехоплений на рівні вище, і звести до мінімуму кількість операторів блок try.

```
try:
    import PyShell
except ImportError:
    raise
else:
    import os
    idledir = os.path.dirname(os.path.abspath(PyShell.__file__))
    if idledir != os.getcwd():
        # We're not in the IDLE directory, help the user find it
        pypath = os.environ.get('PYTHONPATH', '')
        if pypath:
            os.environ['PYTHONPATH'] = pypath + ':' + idledir
        else:
            os.environ['PYTHONPATH'] = idledir
    PyShell.main()
```

Виключення та їх обробка

Блок finally

- Оператори всередині блоку **finally** виконуються незалежно від того, чи виник якийсь виняток.
- Призначений для виконання так званих cleanup actions, тобто дій з очищення: закриття файлів, видалення тимчасових об'єктів тощо.
- Якщо виняток не було перехоплено жодним з блоків except, він заново викидається інтерпретатором після виконання дій у 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** розташований всередині функції), **break** або **continue** (коли оператор **try/except** розташований всередині циклу) або виник інший необроблений виняток під час обробленні цього винятку.

finally.

Виключення та їх обробка

Базові стандартні класи винятків

Клас	Опис
BaseException	Базовий клас для всіх винятків.
Exception	Базовий клас для всіх стандартних винятків, які не вказують на обов'язкове завершення програми, та всіх користувацьких винятків.
ArithmeticError	Базовий клас для всіх винятків, які пов'язані з арифметичними операціями.
BufferError	Базовий клас для винятків, які пов'язані з операціями над буфером.
LookupError	Базовий клас для винятків, які пов'язані з неправильним ключем або індексом колекції.
StandardError (Python 2)	Базовий клас для всіх вбудованих винятків, окрім StopIteration, GeneratorExit, KeyboardInterrupt та SystemExit.
EnvironmentError (Python 2)	Базовий клас для винятків, які пов'язані з помилками, що виникають поза інтерпретатором Python.

Виключення та їх обробка

Синтаксичні помилки

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



Виключення та їх обробка

Попередження

- **Попередження** зазвичай виводяться на екран у ситуаціях, коли помилкова поведінка не гарантується. Програма, як правило, може продовжувати роботу, однак користувача варто повідомити про що-небудь.
- Базовим класом для попереджень є **Warning**, який успадковується від **Exception**.
- Базовим класом-спадкоємцем **Warning** для користувацьких попереджень є **UserWarning**.
- У модулі **warning** зібрані функції для роботи із попередженнями. Основною є функція **warn**, яка приймає один обов'язковий параметр `message`, який може бути або рядком-повідомленням, або екземпляром класу чи підкласу **Warning** (у такому випадку параметр `category` встановлюється автоматично) та два опціональні параметри: `category` (за замовчуванням – **UserWarning**) – клас попередження та `stacklevel` (за замовчування – 1) – рівень вкладеності функцій, починаючи з якого необхідно виводити вміст стека викликів.



Виключення та їх обробка

EAFP vs LBYL

- У статично типізованих мовах компілятор контролює, реалізує клас, екземпляром якого є даний об'єкт чи певний інтерфейс. За динамічної качиної типізації відповідальність за це лежить на програмісті. 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 та 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, що характеризується наявністю блоків `try/except`. У контексті качиної типізації: написання коду з огляду на припущення, що цей об'єкт реалізує необхідний інтерфейс, та обробка винятку `AttributeError` в іншому випадку. Механізм винятків розглядається на уроці №7.



Переважає стилем у Python є EAFP

Виключення та їх обробка

EAFP vs LBYL

LBYL і EAFP – це досить загальні стилі написання коду динамічними мовами, які стосуються не тільки качиної типізації. Наприклад: перевірка наявності ключа у словнику (LBYL) або обробка винятку `KeyError` (EAFP), перевірка наявності файлу (LBYL) або обробка винятку `IOError` (EAFP).

Переваги стилю EAFP:

- код простіше читається завдяки відсутності зайвих перевірок;
- винятки в Python працюють досить швидко;
- позбавлений ризику виникнення стану гонитви в багатопотоковому оточенні, що іноді відбувається за використання підходу LBYL.



Дивіться наші уроки у відео форматі

ITVDN.com



ITVDN

IT VIDEO DEVELOPERS NETWORK

Перегляньте цей урок у відеоформаті на освітньому порталі [ITVDN.com](http://itvdn.com) для закріплення пройденого матеріалу.

Усі курси записані сертифікованими тренерами, які працюють у навчальному центрі CyberBionic Systematics.

Перевірка знань

TestProvider.com

TestProvider – це online-сервіс перевірки знань з інформаційних технологій. Завдяки ньому Ви можете оцінити Ваш рівень і виявити слабкі місця. Він буде корисним як у процесі вивчення технології, так і загальної оцінки знань IT-фахівця.

Після кожного уроку проходите тестування для перевірки знань на TestProvider.com

Успішне проходження фінального тестування дасть Вам змогу отримати відповідний Сертифікат.



TestProvider

Python Essential

Q&A

Інформаційний відеосервіс для розробників програмного забезпечення

