

Python Essential

Исключения

Python Essential

После урока обязательно



Повторите этот урок в видео формате на [ITVDN.com](http://itvdn.com)



Проверьте как Вы усвоили данный материал
на TestProvider.com

Python Essential

Тема

Исключения

Исключения

Понятие механизма исключений

Обработка исключительных ситуаций или **обработка исключений** (англ. exception handling) – механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Некоторые классические примеры исключительных ситуаций:

- деление на ноль;
- ошибка при попытке считать внешние данные;
- исчерпание доступной памяти.



Исключения

Выброс исключений

- Python автоматически генерирует исключения при возникновении ошибки времени выполнения.
- Код на Python может сгенерировать исключение при помощи ключевого слова **raise**. После него указывается объект исключения. Также, можно указать класс исключения, в таком случае будет автоматически вызван конструктор без параметров. **raise** может выбрасывать в качестве исключений только экземпляры класса **BaseException** и его наследников, а также (в Python 2) экземпляры классов старого типа.
- Помните, что классы старого типа в Python 2 существуют только для обратной совместимости и использовать их не следует.
- Все стандартные классы исключений в Python являются классами нового типа и наследуются от **Exception** либо напрямую от **BaseException**. Все пользовательские исключения должны быть наследниками **Exception**.



Исключения

Обработка исключительных ситуаций в Python

```
try:                                # область действия обработчика
    ...
except Exception1:                  # обработчик исключения Exception1
    ...
except Exception2:                  # обработчик исключения Exception2
    ...
except:                             # стандартный обработчик исключений
    ...
else:                               # код, который выполняется, если никакое
    ...                             # исключение не возникло
finally:                            # код, который выполняется в любом случае
    ...
```



Исключения

Блок try

Блок **try** задаёт область действия обработчика исключений. Если при выполнении операторов в данном блоке было выброшено исключение, их выполнение прерывается и управление переходит к одному из обработчиков. Если не возникло никакого исключения, блоки **except** пропускаются.

```
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
    # This is necessary as someone may
    # installed but be using other feed
    #
    # ... needed in order to c...
```

Исключения

Блок except

```
try:
    pass
except Exception1:                # обработчик исключения Exception1
    pass
except (Exception2, Exception3):  # обработчик исключений Exception2 и Exception3
    pass
except Exception4 as exception:   # обработчик исключения Exception4
    pass                         # экземпляр исключения доступен под именем exception
except Exception4, exception:    # устаревший синтаксис, не поддерживается в Python 3
    pass
except:                           # стандартный обработчик, перехватывает все исключения
    pass
```

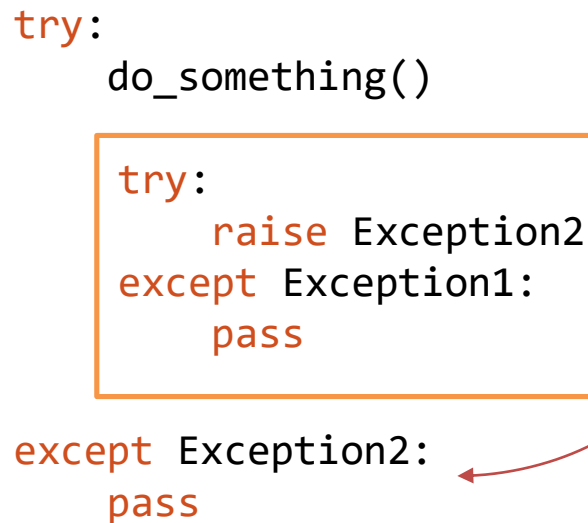


Блоки except обрабатываются сверху вниз и управление передаётся не больше, чем одному обработчику. Поэтому при необходимости по-разному обрабатывать исключения, находящиеся в иерархии наследования, сначала нужно указывать обработчики менее общих исключений, а затем – более общих. Также именно поэтому стандартный блок except может быть только последним.

Исключения

Необработанные исключения

```
try:
    do_something()
    try:
        raise Exception2
    except Exception1:
        pass
except Exception2:
    pass
```



Если ни один из заданных блоков except не перехватывает возникнувшее исключение, то оно будет перехвачено ближайшим внешним блоком try/except, в котором есть соответствующий обработчик. Если же программа не перехватывает исключение вообще, то интерпретатор завершает выполнение программы и выводит информацию об исключении в стандартный поток ошибок sys.stderr. Из этого правила есть два исключения:

- Если исключение возникло в деструкторе объекта, выполнение программы не завершается, а в стандартный поток ошибок выводится предупреждение “Exception ignored” с информацией об исключении.
- При возникновении исключения SystemExit происходит только завершение программы без вывода информации об исключении на экран (не касается предыдущего пункта, в деструкторе поведение данного исключения будет таким же, как и остальных).

Исключения

Передача исключения на один уровень выше

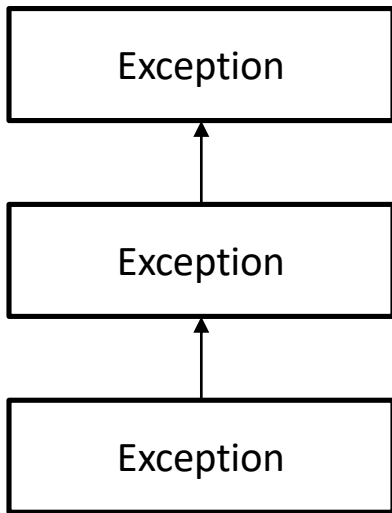
```
try:
    do_some_actions()           # действия, которые могут вызвать исключение
except Exception as exception: # обработчик исключения
    handle_exception(exception) # определённые действия с данным исключением
    raise                       # сгенерировать то же исключение ещё раз
```



Для того, чтобы в обработке исключения выполнить определённые действия, а затем передать исключение дальше, на один уровень обработчиков выше (то есть, выбросить то же самое исключение ещё раз), используется ключевое слово `raise` без параметров.

Исключения

Исключения в блоке except. Сцепление исключений



- В Python 3 при выбросе исключения в блоке **except** старое исключение сохраняется в атрибуте данных `__context__`.
- Для связывания исключений используется конструкция
raise новое_исключение **from** старое_исключение
либо
raise новое_исключение **from** None
- В первом случае указанное исключение сохраняется в атрибуте `__cause__` и атрибут `__suppress_context__` (который подавляет вывод исключения из `__context__`) устанавливается в **True**. Тогда, если новое исключение не обработано, будет выведена информация о том, что старое исключение является причиной нового.
- Во втором случае `__suppress_context__` устанавливается в **True** и `__cause__` в None. Тогда при выводе исключения оно, фактически, будет заменено новым (хотя старое исключение всё ещё хранится в `__context__`).



В Python 2 нет связывания исключений.

Любое исключение, выброшенное в блоке except, заменяет старое.

Исключения

Блок else

- Необязательный блок.
- Операторы внутри него выполняются, если никакое исключение не возникло.
- Предназначен для того, чтобы отделить код, который может вызвать исключение, которое должно быть обработано в данном блоке try/except, от кода, который может вызвать исключение того же класса, которое должно быть перехвачено на уровне выше, и свести к минимуму количество операторов в блоке 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()
```

Исключения

Блок finally

- Операторы внутри блока **finally** выполняются независимо от того, возникло ли какое-либо исключение.
- Предназначен для выполнения так называемых cleanup actions, то есть действий по очистке: закрытие файлов, удаление временных объектов и т.д.
- Если исключение не было перехвачено ни одним из блоков except, то оно заново выбрасывается интерпретатором после выполнения действий в блоке **finally**.
- Блок **finally** выполняется перед выходом из оператора **try/except** всегда, даже если одна из его веток содержит оператор **return** (когда оператор **try/except** находится внутри функции), **break** или **continue** (когда оператор **try/except** находится внутри цикла) или возникло другое необработанное исключение при обработке данного исключения.

finally.

Исключения

Базовые стандартные классы исключений

Класс	Описание
BaseException	Базовый класс для всех исключений.
Exception	Базовый класс для всех стандартных исключений, которые не указывают на обязательное завершение программы, и всех пользовательских исключений.
ArithmeticError	Базовый класс для всех исключений, связанных с арифметическими операциями.
BufferError	Базовый класс для исключений, связанных с операциями над буфером.
LookupError	Базовый класс для исключений, связанных с неверным ключом или индексом коллекции.
StandardError (Python 2)	Базовый класс для всех встроенных исключений, кроме StopIteration, GeneratorExit, KeyboardInterrupt и SystemExit.
EnvironmentError (Python 2)	Базовый класс для исключений, связанных с ошибками, которые происходят вне интерпретатора Python.

Исключения

Синтаксические ошибки

- Ошибка синтаксиса возникает, когда синтаксический анализатор Python сталкивается с участком кода, который не соответствует спецификации языка и не может быть интерпретирован.
- В главном модуле возникает до начала выполнения программы и не может быть перехвачена.
- Ситуации, в которых синтаксическая ошибка в виде исключения `SyntaxError` может быть перехвачена и обработана:
 - ошибка синтаксиса в импортируемом модуле;
 - ошибка синтаксиса в коде, который представляется строкой и передаётся функции `eval` или `exec`.



Исключения

Предупреждения

- **Предупреждения** обычно выводятся на экран в ситуациях, когда не гарантируется ошибочное поведение и программа, как правило, может продолжать работу, однако пользователя следует уведомить о чём-либо.
- Базовым классом для предупреждений является **Warning**, который наследуется от **Exception**.
- Базовым классом-наследником **Warning** для пользовательских предупреждений является **UserWarning**.
- В модуле **warning** собраны функции для работы с предупреждениями. Основной является функция **warn**, которая принимает один обязательный параметр *message*, который может быть либо строкой-сообщением, либо экземпляром класса или подкласса **Warning** (в таком случае параметр *category* устанавливается автоматически) и два опциональных параметра: *category* (по умолчанию – **UserWarning**) – класс предупреждения и *stacklevel* (по умолчанию – 1) – уровень вложенности функций, начиная с которого необходимо выводить содержимое стека вызовов.



Исключения

EAFP vs LBYL

- В статически типизированных языках компилятор контролирует, реализует ли класс, экземпляром которого является данный объект, определённый интерфейс. При динамической утиной типизации ответственность за это лежит на программисте. Есть два противоположных подхода к реализации таких проверок.
- **LBYL** (*Look Before You Leap* – «семь раз отмерь, один раз отрежь») – стиль, который характеризуется наличием множества проверок и условных операторов. В контексте утиной типизации может означать проверку наличия необходимых атрибутов при помощи функции `hasattr`.
- **EAFP** (*Easier to Ask for Forgiveness than Permission* – «проще попросить прощения, чем разрешения») – стиль, который характеризуется наличием блоков `try/except`. В контексте утиной типизации – написание кода исходя из предположения, что данный объект реализует необходимый интерфейс, и обработка исключения `AttributeError` в противном случае. Механизм исключений рассматривается в уроке №7.



Предпочтительным стилем в Python является EAFP

Исключения

EAFP vs LBYL

LBYL и EAFP – это довольно общие стили написания кода на динамических языках, которые касаются не только утиной типизации. Например: проверка существования ключа в словаре (LBYL) или обработка исключения `KeyError` (EAFP), проверка существования файла (LBYL) или обработка исключения `IOError` (EAFP).

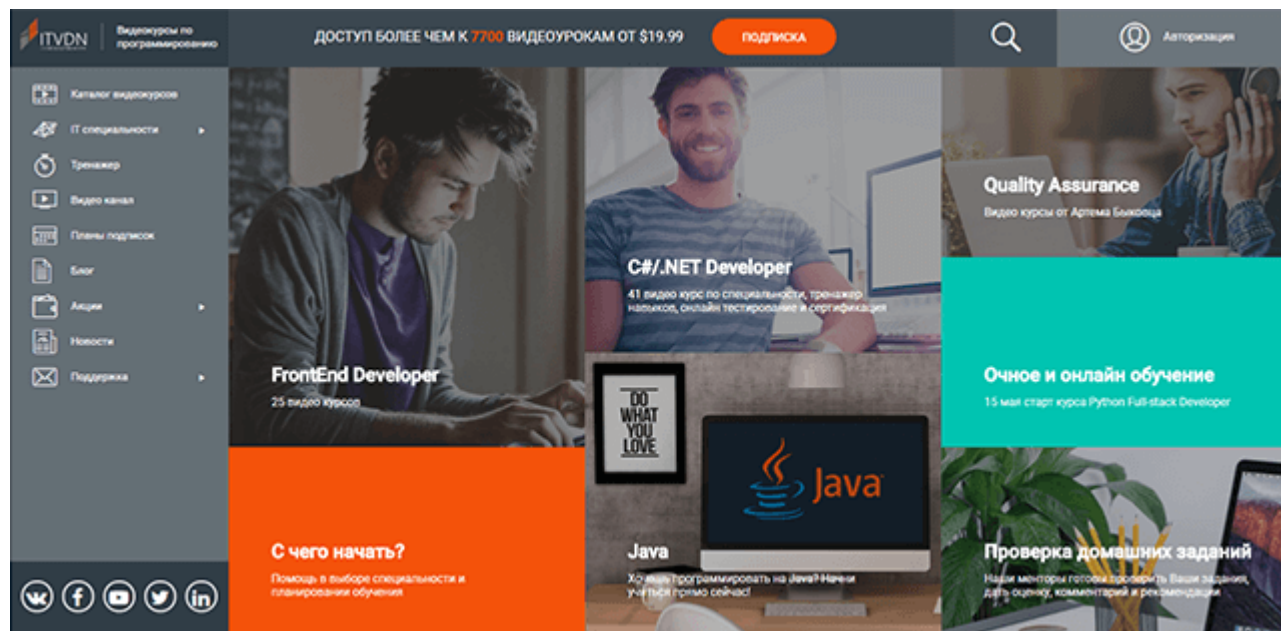
Преимущества стиля EAFP:

- код проще читается благодаря отсутствию лишних проверок;
- исключения в Python работают довольно быстро;
- лишён риска возникновения состояния гонки в многопоточном окружении, что иногда происходит при использовании подхода LBYL.



Смотрите наши уроки в видео формате

ITVDN.com



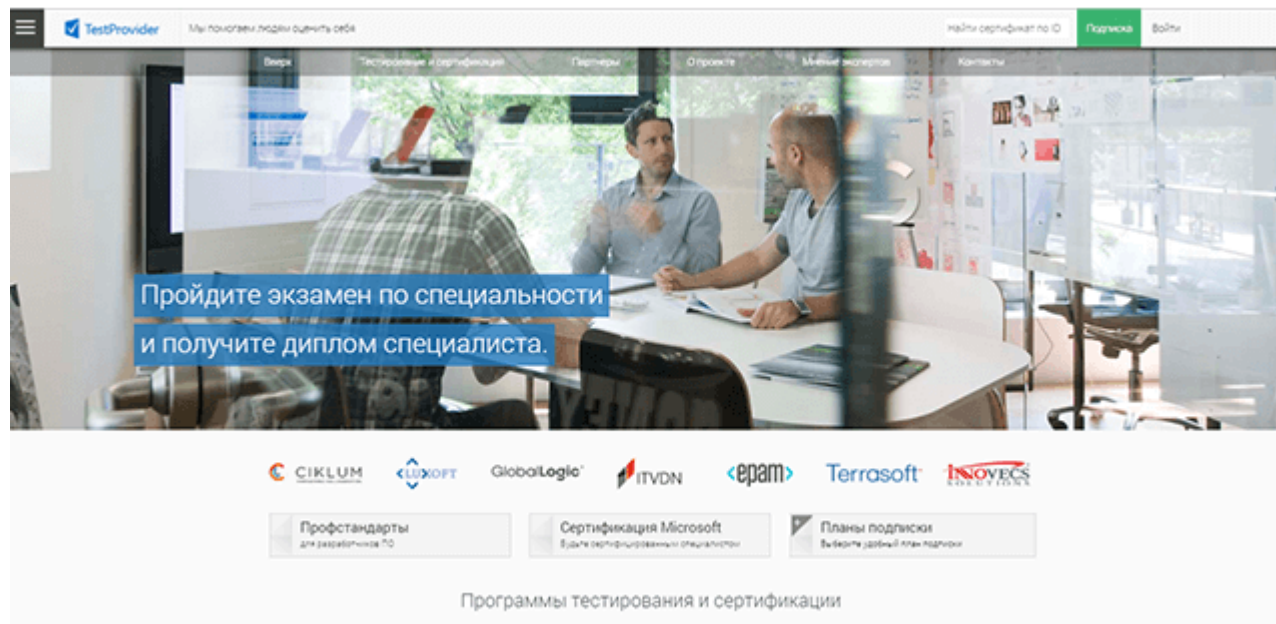
Посмотрите этот урок в видео формате на образовательном портале ITVDN.com для закрепления пройденного материала.

Курсы записаны сертифицированными тренерами, которые работают в учебном центре CyberBionic Systematics и другими высококвалифицированными разработчиками.



Проверка знаний

TestProvider.com



TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

После каждого урока проходите тестирование для проверки знаний на [TestProvider.com](https://testprovider.com)

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.



Python Essential

Q&A

Информационный видеосервис для разработчиков программного обеспечения

