

## Лабораторная работа №5

**Тема:** «Анализ ошибок и обработка исключений. Обработка и генерация исключений. Модульное тестирование».

### Обработка и генерация исключений

При программировании на Python мы можем столкнуться с двумя типами ошибок. Первый тип представляют синтаксические ошибки (*syntax error*). Они появляются в результате нарушения синтаксиса языка программирования при написании исходного кода. При наличии таких ошибок программа не может быть скомпилирована. При работе в какой-либо среде разработки, например, в *PyCharm*, IDE сама может отслеживать синтаксические ошибки и каким-либо образом их выделять.

Второй тип ошибок представляют ошибки выполнения (*runtime error*). Они появляются в уже скомпилированной программе в процессе ее выполнения. Подобные ошибки еще называются *исключениями*. Например, в прошлых темах мы рассматривали преобразование числа в строку:

```
string = "5"
number = int(string)
print(number)
```

Данный скрипт успешно выполнится, так как строка "5" вполне может быть конвертирована в число. Однако возьмем другой пример:

```
string = "hello"
number = int(string)
print(number)
```

При выполнении этого скрипта будет выброшено исключение *ValueError*, так как строку *"hello"* нельзя преобразовать в число. С одной стороны, здесь очевидно, что строка не представляет число, но мы можем иметь дело с вводом пользователя, который также может ввести не совсем то, что мы ожидаем:

```
string = input("Введите число: ")
number = int(string)
print(number)
```

При возникновении исключения работа программы прерывается, и чтобы избежать подобного поведения и обрабатывать исключения в Python есть конструкция *try/except*, которая имеет следующее формальное определение:

```
try:
    инструкции
except [Тип_исключения]:
    инструкции
```

Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова *try*. Если в этом коде генерируется исключение, то работа кода в блоке *try* прерывается, и выполнение переходит в блок *except*.

После ключевого слова *except* опционально можно указать, какое исключение будет обрабатываться (например, *ValueError* или *KeyError*). После слова *except* на следующей строке идут инструкции блока *except*, выполняемые при возникновении исключения.

Рассмотрим обработку исключения на примере преобразовании строки в число:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except:
    print("Преобразование прошло неудачно")
print("Завершение программы")
```

Вводим строку:

```
Введите число: hello
Преобразование прошло неудачно
Завершение программы
```

Как видно из консольного вывода, при вводе строки вывод числа на консоль не происходит, а выполнение программы переходит к блоку *except*.

Вводим правильное число:

```
Введите число: 22
Введенное число: 22
Завершение программы
```

Теперь все выполняется нормально, исключение не возникает, и, соответственно, блок *except* не выполняется.

В примере выше обрабатывались сразу все исключения, которые могут возникнуть в коде. Однако мы можем конкретизировать тип обрабатываемого исключения, указав его после слова *except*:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except ValueError:
    print("Преобразование прошло неудачно")
print("Завершение программы")
```

Если ситуация такова, что в программе могут быть сгенерированы различные типы исключений, то мы можем их обработать по отдельности, используя дополнительные выражения *except*:

```
try:
    number1 = int(input("Введите первое число: "))
    number2 = int(input("Введите второе число: "))
    print("Результат деления:", number1/number2)
except ValueError:
    print("Преобразование прошло неудачно")
except ZeroDivisionError:
    print("Попытка деления числа на ноль")
except Exception:
    print("Общее исключение")
print("Завершение программы")
```

Если возникнет исключение в результате преобразования строки в число, то оно будет обработано блоком *except ValueError*. Если же второе число будет равно нулю, то есть будет деление на ноль, тогда возникнет исключение *ZeroDivisionError*, и оно будет обработано блоком *except ZeroDivisionError*.

Тип *Exception* представляет общее исключение, под которое попадают все исключительные ситуации. Поэтому в данном случае любое исключение, которое не представляет тип *ValueError* или *ZeroDivisionError*, будет обработано в блоке *except Exception*.

## Блок `finally`

При обработке исключений также можно использовать необязательный блок *finally*. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except ValueError:
    print("Не удалось преобразовать число")
finally:
    print("Блок try завершил выполнение")
print("Завершение программы")
```

Как правило, блок `finally` применяется для освобождения используемых ресурсов, например, для закрытия файлов.

## Получение информации об исключении

С помощью оператора *as* мы можем передать всю информацию об исключении в переменную, которую затем можно использовать в блоке *except*:

```
try:
    number = int(input("Введите число: "))
    print("Введенное число:", number)
except ValueError as e:
    print("Сведения об исключении", e)
print("Завершение программы")
```

Пример некорректного ввода:

```
Введите число: fdsf
Сведения об исключении invalid literal for int() with base 10: 'fdsf'
Завершение программы
```

## Генерация исключений

Иногда возникает необходимость вручную сгенерировать то или иное исключение.

Для этого применяется оператор *raise*.

```
try:
    number1 = int(input("Введите первое число: "))
    number2 = int(input("Введите второе число: "))
    if number2 == 0:
        raise Exception("Второе число не должно быть равно 0")
    print("Результат деления двух чисел:", number1/number2)
except ValueError:
    print("Введены некорректные данные")
except Exception as e:
    print(e)
print("Завершение программы")
```

При вызове исключения мы можем ему передать сообщение, которое затем можно вывести пользователю:

```
Введите первое число: 1
Введите второе число: 0
Второе число не должно быть равно 0
Завершение программы
```

## Отладка с помощью инструкции assert

Инструкция *assert* позволяет производить проверки истинности утверждений, что может быть использовано в отладочных целях.

Если проверка не прошла, выбрасывается исключение *AssertionError*.

Рекомендуется использовать инструкцию только для проверки внутреннего состояния программы – ситуаций, которые не должны происходить вовсе, которые нельзя обработать или это не имеет смысла (обычно это является указанием на то, что код программы содержит ошибку). Инструкция также может использоваться для документирования ожиданий (например, входных параметров или результата). В остальных случаях следует определять свои типы исключений.

```

passed = False
assert passed, 'Not passed'  # Поднимается исключение.
# assert passed
# Можно и не указывать текст описания, но рекомендуется.

# Запись выше эквивалентна следующей конструкции:
if __debug__:
    if not passed:
        raise AssertionError('Not passed')

# Пример правильного переноса строки описания:
assert a, ('Long exception chunked.')

# А теперь пример неправильного (передача кортежа),
# при котором проверка всегда проходит без ошибок:
assert (a, 'Long exception chunked.')

```

В текущей реализации `__debug__` – встроенная константа, по умолчанию имеющая значение **True**. Если интерпретатор запущен в режиме оптимизации (с флагом командной строки **-O**), значение константы становится **False**, а генератор кода перестаёт производить байткод для рассматриваемой инструкции. Таким образом, отключив проверки, но не убирая их из кода, можно снизить неизбежные для них накладные расходы.

### Стандартные исключения

№	Имя	Описание
1	Exception	Базовый класс для всех исключений
2	StopIteration	Возникает, когда метод next() цикла не указывает на какой-либо объект
3	SystemExit	Вызываемая функция sys.exit()
4	StandardError	Базовый класс для всех встроенных исключений, кроме StopIteration и SystemExit
5	ArithmeticError	Базовый класс для всех ошибок, которые возникают для числовых расчетов
6	OverflowError	Возникает, когда значение превышает максимальный предел для числового типа
7	FloatingPointError	Возникает, когда расчет с плавающей точкой терпит неудачу

№	Имя	Описание
8	ZeroDivisonError	Возникает при делении на ноль или делении по модулю нуля, имеет место для всех числовых типов
9	AssertionError	Вызывается в случае выхода из строя заявления Assert
10	AttributeError	Вызывается в случае выхода из строя ссылки на атрибут или назначения
11	EOFError	Возникает, когда нет входного сигнала либо когда из функций raw_input () или input() достигнут конец файла
12	ImportError	Возникает, когда оператор import терпит неудачу
13	KeyboardInterrupt	Возникает, когда пользователь прерывает выполнение программы, как правило, нажав Ctrl + C
14	LookupError	Базовый класс для всех ошибок поиска
15	IndexError	Возникает, когда индекс не найден в последовательности
16	KeyError	Возникает, когда указанный ключ не найден в словаре
17	NameError	Возникает, когда идентификатор не найден в локальном или глобальном пространстве имен
18	UnboundLocalError	Возникает при попытке доступа к локальной переменной в функции или методе, но значение не было присвоено
19	EnvironmentError	Базовый класс для всех исключений, которые происходят за пределами среды Python
20	IOError	Возникает, когда операция ввода/вывода не работает, например, заявление для печати или функции Open() при попытке открыть файл, который не существует
21	OSError	Вызывает ошибку, связанную с операционной системой
22	SyntaxError	Возникает, когда есть ошибка в синтаксисе Python
23	IndentationError	Возникает, когда неправильно указаны отступы
24	SystemError	Возникает, когда интерпретатор находит внутреннюю проблему, но при возникновении этой ошибки интерпретатор Python не выходит
25	SystemExit	Вызывается, когда интерпретатор Python выходит с помощью функции sys.exit(). Если код не обработан, транслятор завершает работу

№	Имя	Описание
26	TypeError	Возникает при попытке операции или функции, недопустимой для указанного типа данных
27	ValueError	Возникает, когда встроенная функция для типа данных имеет допустимый тип аргументов, но аргументы имеют недопустимые значения
28	RuntimeError	Возникает, когда генерируется ошибка, не попадающая ни в какую категорию
29	NotImplementedError	Возникает, когда абстрактный метод, который должен быть реализован у унаследованного класса, фактически не реализуется

### Пользовательские исключения (User-defined Exceptions)

В Python можно создавать собственные исключения. Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.

Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений.

```
class NegValException(Exception):
    pass

# Пример использования пользовательского исключения NegValException
try:
    val = int(input("input positive number: "))
    if val < 0:
        raise NegValException("Neg val: " + str(val))
    print(val + 10)
except NegValException as e:
    print(e)
```



## Модульное тестирование

**Автономный тест** – это автоматизированная часть кода, которая вызывает тестируемую единицу кода и затем проверяет некоторые предположения о единственном конечном результате этой единицы. В качестве тестируемой единицы может выступать как отдельная функция (или метод), так и совокупность классов (или функций). Идея автономной единицы в том, что она представляет собой некоторую логически законченную сущность программы. Автономное тестирование ещё называют модульным или unit тестированием (unit-testing).

**unittest** – это фреймворк для тестирования, входящий в стандартную библиотеку языка Python. Данный фреймворк поддерживает автоматизацию тестов, использование общего кода для настройки и завершения тестов, объединение тестов в группы, а также позволяет отделять тесты от фреймворка для вывода информации.

**Test fixture** – обеспечивает подготовку окружения для выполнения тестов, а также организацию мероприятий по их корректному завершению (например очистка ресурсов). Подготовка окружения может включать в себя создание баз данных, запуск необходимых серверов и т. п.

**Test case** – это элементарная единица тестирования, в рамках которой проверяется работа компонента тестируемой программы (метод, класс, поведение и т. п.). Для реализации этой сущности используется класс TestCase.

**Test suite** – это коллекция тестов, которая может в себя включать как отдельные «тест кейсы», так и целые коллекции (т. е. можно создавать коллекции коллекций). Коллекции используются с целью объединения тестов для совместного запуска.

**Test runner** – это компонент, который оркестрирует (координирует взаимодействие) запуск тестов и предоставляет пользователю результат их выполнения. Test runner может иметь графический интерфейс, текстовый интерфейс или возвращать какое-то заранее заданное значение, которое будет описывать результат прохождения тестов.

Вся работа по написанию тестов заключается в том, что мы разрабатываем отдельные тесты в рамках тест кейсов, собираем их в модули и запускаем. Если нужно объединить несколько тест кейсов для их совместного запуска, они помещаются в тестовые наборы (test suite), которые помимо тест-кейсов могут содержать другие тестовые наборы.

```
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        # Тест: 'foo'.upper() должно вернуть 'FOO'
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        # 'FOO'.isupper() должно вернуть True
        self.assertTrue('FOO'.isupper())

        # 'Foo'.isupper() должно вернуть False
        self.assertFalse('Foo'.isupper())

if __name__ == '__main__':
    unittest.main()
```

Запуск тестов можно сделать как из командной строки, так и с помощью графического интерфейса пользователя (GUI). Интерфейс командной строки (CLI), позволяет запускать тесты из целого модуля, класса или даже обращаться к конкретному тесту.

Запуск всех тестов в модуле `test_string.py`:

```
>> python -m unittest test_string.py
```

Запуск тестов из класса `TestStringMethods`:

```
>> python -m unittest test_string.TestStringMethods
```

Запуск теста `test_upper()`:

```
>> python -m unittest test_string.TestStringMethods.test_upper
```

Когда некоторые тесты имеют лишь незначительные отличия, например некоторые параметры, `unittest` позволяет различать их внутри одного тестового метода, используя менеджер контекста `subTest()`.

```
import unittest

class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Тест, что все числа между 0 и 5 должны быть чётными
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

## Требования к выполнению лабораторной работы №5

1. Изучите теоретическую часть к пятой лабораторной работе:
  - a. Теоретическая часть к пятой лабораторной работе.
  - b. Лекция №5.
2. Создайте новый проект.
3. Запустите примеры из лабораторной работы.
4. Выполните задание:
  - a. Каждое задание представляет собой отдельный скрипт формата:  
`lab_{номер_ЛР}_{номер_задания}_{номер_варианта}.py`, пример:  
`lab_5_1_2.py`
  - b. Отправьте выполненное задание в ОРИОКС (*раздел Домашние задания*).

### Формат защиты лабораторных работ:

1. Продемонстрируйте выполненные задания.
2. Ответьте на вопросы по вашему коду.
3. При необходимости выполните дополнительное (*дополнительные*) задание от преподавателя.
4. Ответьте (*устно*) преподавателю на контрольные вопросы.

### Список вопросов

1. Генерация исключений.
2. `try / except / else`
3. Сцепление исключений.
4. Оператор `assert`.
5. Диспетчер контекстов.
6. Что такое автономный тест?
7. `Test fixture`.
8. `Test case`.
9. `Tetst suite`.
10. `Test runner`.

## Задания

### Общее задание

1. Необходимо проверять корректность вводимых данных и выводить соответствующие сообщения об ошибках.
2. В качестве базовой программы используйте наработки из предыдущей лабораторной работы.
3. Добавьте пользовательский ввод данных с клавиатуры.
4. Добавьте возможность сохранять и восстанавливать данные из файла (*бинарная запись*).
5. Добавьте обработку исключений при чтении и записи файла.
6. Добавьте обработку исключений на случай некорректных данных.
7. Добавьте пользовательское исключение.
8. Для контроля качества выполнения лабораторной работы напишите модульные тесты.