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 тестируемый 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).

Pytest – это фреймворк для тестирования кода на Python.

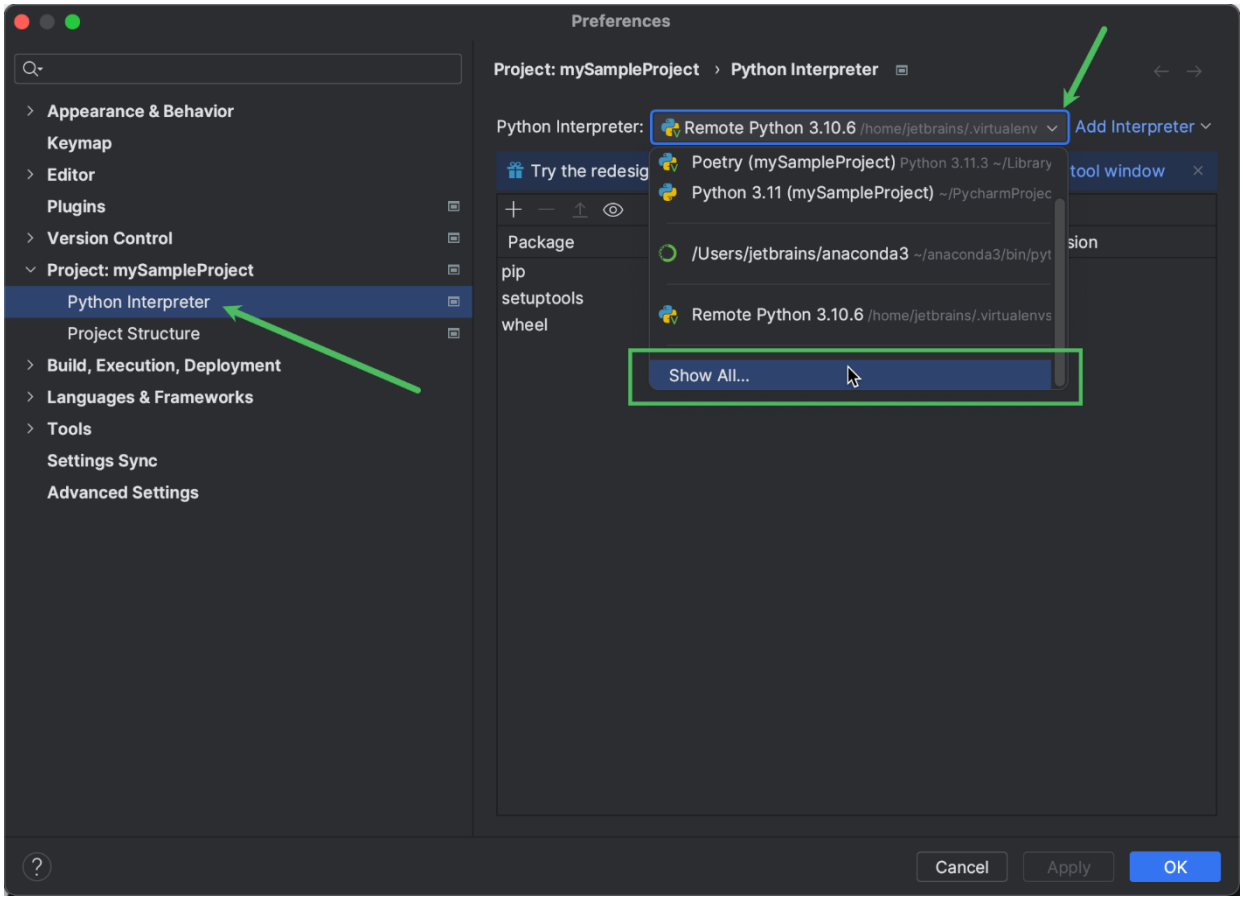
1. Установка Pytest через терминал:

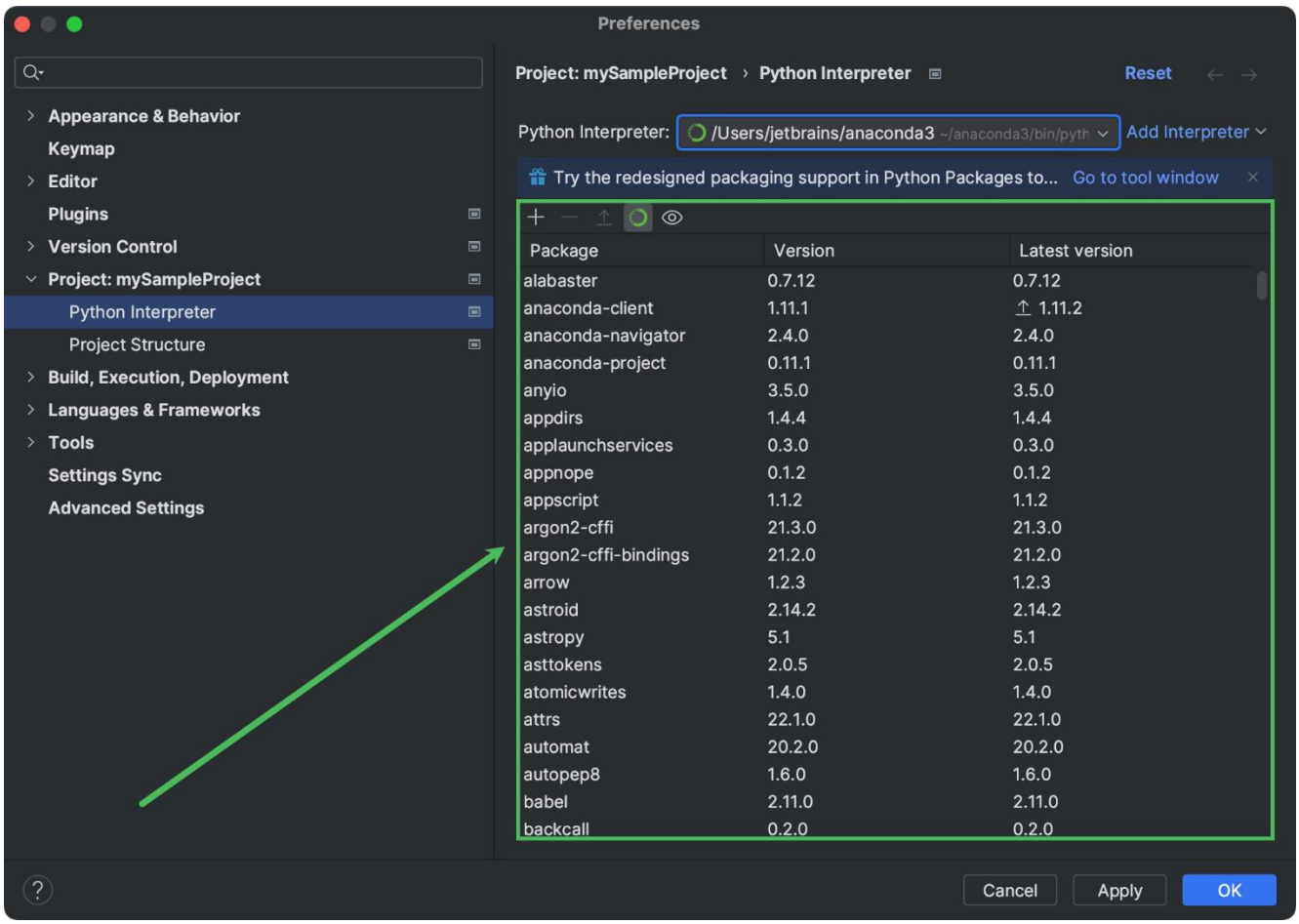
```
pip install -U pytest
```

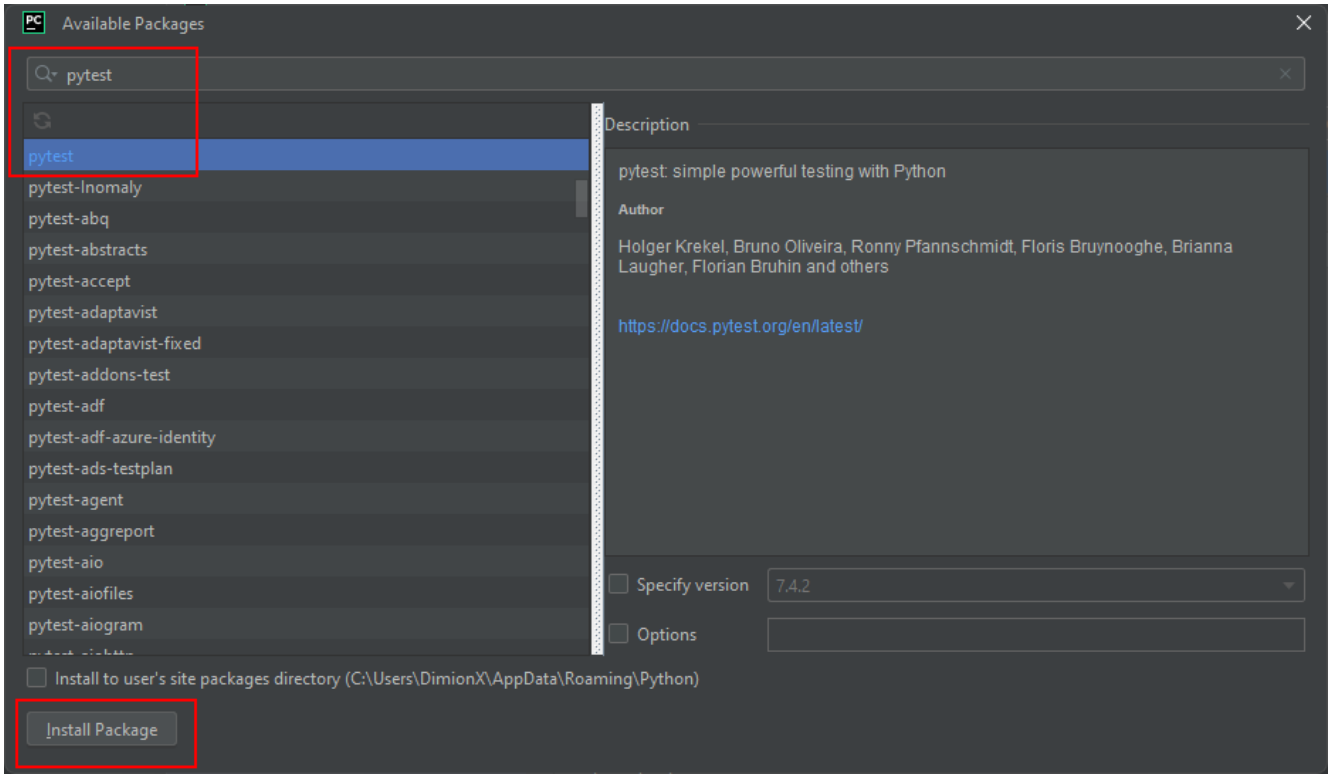
```
pytest --version
```

2. Установка Pytest через IDE PyCharm

```
Settings => Project => Python Interpreter
```




Preferences
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Project: mySampleProject
Python Interpreter
Project Structure
Build, Execution, Deployment
Languages & Frameworks
Tools
Settings Sync
Advanced Settings
Project: mySampleProject › Python Interpreter
Reset
Python Interpreter: /Users/jetbrains/anaconda3 ~/anaconda3/bin/pyth
Add Interpreter
Try the redesigned packaging support in Python Packages to...
Go to tool window
Package
Version
Latest version
alabaster 0.7.12 0.7.12
anaconda-client 1.11.1 1.11.2
anaconda-navigator 2.4.0 2.4.0
anaconda-project 0.11.1 0.11.1
anyio 3.5.0 3.5.0
appdirs 1.4.4 1.4.4
applaunchservices 0.3.0 0.3.0
appnope 0.1.2 0.1.2
appscript 1.1.2 1.1.2
argon2-cffi 21.3.0 21.3.0
argon2-cffi-bindings 21.2.0 21.2.0
arrow 1.2.3 1.2.3
astroid 2.14.2 2.14.2
astropy 5.1 5.1
asttokens 2.0.5 2.0.5
atomicwrites 1.4.0 1.4.0
attrs 22.1.0 22.1.0
automat 20.2.0 20.2.0
autopep8 1.6.0 1.6.0
babel 2.11.0 2.11.0
backcall 0.2.0 0.2.0
Cancel
Apply
OK


Available Packages
pytest
pytest
pytest-Inomaly
pytest-abq
pytest-abstracts
pytest-accept
pytest-adaptavist
pytest-adaptavist-fixed
pytest-addons-test
pytest-adf
pytest-adf-azure-identity
pytest-ads-testplan
pytest-agent
pytest-aggreport
pytest-aio
pytest-aiofiles
pytest-aiogram
Description
pytest: simple powerful testing with Python
Author
Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, Florian Bruhin and others
https://docs.pytest.org/en/latest/
Specify version
7.4.2
Options
Install to user's site packages directory (C:\Users\DimionX\AppData\Roaming\Python)
Install Package

Тестовые функции – функции, названия которых называется с test_.

```
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 4
```

Pytest запускает все файлы вида `test_*.py` или `*_test.py` в текущей директории и поддиректориях.

Pytest позволяет создать класс, содержащий более одного теста:

```
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

Запуск тестов

```
pytest
```

или

```
python -m pytest
```

Параметризация тестовых функций

Встроенный декоратор `pytest.mark.parametrize` позволяет параметризовать аргументы тестовых функций.

```
import pytest


@pytest.mark.parametrize("test_value,expected", [(1, 1), (10, 100), (3, 9)])
def test_eval(test_value, expected):
    assert test_value ** 2 == expected
```

Test fixture – обеспечивает подготовку окружения для выполнения тестов, а также необходимые действия для их корректного завершения (например очистка ресурсов).

Подготовка окружения может включать в себя создание баз данных, запуск необходимых серверов и т. п.

Шаблон «фабрика-фикстура» может помочь в ситуациях, когда результат, возвращаемый фикстурой, используется много раз в отдельном тесте. Вместо того, чтобы напрямую возвращать данные, фикстура возвращает функцию, которая генерирует данные. Затем эта функция может быть неоднократно вызвана в тесте.

Для того, чтобы зарегистрировать функцию как фикстуру, нужно использовать декоратор `@pytest.fixture`.

Фикстуры позволяют тестовым функциям легко получать предварительно инициализированные объекты и работать с ними, не заботясь об импорте/установке/очистке.

Teardown/Cleanup

При запуске тестов необходимо убедиться, что они не влияют на другие тесты (а также, что они не оставляют после себя огромное количество тестовых данных, которые раздувают систему). Фикстуры в pytest позволяют реализовать метод `teardown`, который позволит определить необходимые шаги завершения.

Подобный метод можно реализовать двумя способами. В данной лабораторной работе рассматриваются только Yield-фикстуры (рекомендуемый способ).

Pytest поддерживает выполнение фикстурами специфического завершающего кода при выходе из области действия. Если вы используете оператор `yield` вместо `return`, то весь код после `yield` выполняет роль «уборщика»:

```
import smtplib
import pytest


@pytest.fixture(scope="module")
def smtp_connection():
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp_connection  # возвращает значение фикстуры
    print("teardown smtp")
    smtp_connection.close()
```

Операторы `print` и `smtp.close()` будут выполнены после завершения последнего теста модуля независимо от того, было ли вызвано исключение или нет.

Область видимости фикстур

Фикстуры создаются при первом запросе тестом и уничтожаются в зависимости от их области действия `scope`. Аргумент `scope` может принимать следующие параметры:

- `function`: область действия по умолчанию, фикстура уничтожается в конце каждого теста, где она используется;
- `class`: фикстура уничтожается во время завершения последнего теста в классе;
- `module`: фикстура уничтожается во время завершения последнего теста в модуле;
- `package`: фикстура уничтожается во время завершения последнего теста в пакете;
- `session`: фикстура уничтожается в конце тестового сеанса.

Пример: `@pytest.fixture(scope="module")`

Autouse fixtures

Пример, когда фикстуры вызываются автоматически, без явного указания их в качестве аргумента:

```
import pytest


@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order(first_entry):
    return []


@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]


def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

Тестирование исключений

Чтобы написать тест, который проверяет возникающие исключения, вы можете использовать `pytest.raises()` следующим образом:

```
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

Маркировка тестов

Благодаря `pytest.mark` можно передавать метаданные в ваши тесты.

```
import pytest


@pytest.mark.my_custom_marker
def test_foo():
    pass


def test_bar():
    pass
```

Запуск маркированного теста:

```
pytest -v -m my_custom_marker
```

или

```
python -m pytest -v -m my_custom_marker
```

Инверсия: `pytest -v -m "not my_custom_marker "`

Маркеры skip и xfail

Тестовые функции, которые не могут быть запущены на определенных платформах или от которых вы ожидаете сбоя, можно пометить так, чтобы pytest работал с ними соответствующим образом и представлял сводку сеанса тестирования, считая набор тестов пройденным и помечая его зеленым.

`skip` используется в случае, когда вы ожидаете, что ваш тест пройдет только при соблюдении некоторых условий, в противном случае pytest должен полностью пропустить выполнение теста. Распространенными примерами являются пропуск тестов только для Windows на платформах, отличных от Windows, или пропуск тестов, зависящих от внешнего ресурса, который в данный момент недоступен (например, базы данных).

Пример:

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    ...
```

`xfail` применяется, когда вы ожидаете, что тест по каким-то причинам должен провалиться. Обычный пример – это тест на еще не реализованную функцию или еще не исправленную ошибку. Когда тест, помеченный `pytest.mark.xfail`, проходит, несмотря на ожидаемое падение, в сводке результатов он будет помечен как `xpass`.

Pytest подходит как для тестирования кода на python, так и для тестирования сторонних сервисов/сайтов/API.

Требования к выполнению лабораторной работы №5

1. Изучите теоретическую часть к пятой лабораторной работе.
2. Создайте новый проект.
3. Запустите примеры из лабораторной работы.
4. Выполните задания.
5. 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. Параметризация тестовых функций.
8. Test fixture.
9. Teardown/Cleanup
10. Autouse
11. Маркировка тестов
12. Маркеры skip и xfail

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.