

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
По лабораторной работе №4
Дисциплины «Объектно-ориентированное программирование»

Выполнил:

Пустяков Андрей Сергеевич

3 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и
вычислительная техника (профиль)
«Программное обеспечение средств
вычислительной
техники и автоматизированных
систем», очная форма обучения

(подпись)

Руководитель практики:

Воронкин Р. А., доцент департамента
цифровых и робототехнических
систем и электроники института
перспективной инженерии

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: Работа с исключениями в языке Python.

Цель: приобрести навыки по работе с исключениями при написании программ с помощью языка программирования Python версии 3.x.

Ход работы:

Проработка примеров лабораторной работы:

Необходимо для примера лабораторной работы добавить возможность работы с исключениями и логгирование.

Код примера с возможностью работы с исключениями:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from dataclasses import dataclass, field
from datetime import date
import logging
import sys
from typing import List
import xml.etree.ElementTree as ET

# Класс пользовательского исключения в случае, если неверно
# введен номер года.
class IllegalYearError(Exception):
    def __init__(self, year, message="Illegal year number"):
        self.year = year
        self.message = message
        super(IllegalYearError, self).__init__(message)

    def __str__(self):
        return f"{self.year} -> {self.message}"

# Класс пользовательского исключения в случае, если введенная
# команда является недопустимой.
class UnknownCommandError(Exception):

    def __init__(self, command, message="Unknown command"):

        self.command = command
        self.message = message
        super(UnknownCommandError, self).__init__(message)

    def __str__(self):
        return f"{self.command} -> {self.message}"

@dataclass(frozen=True)
class Worker:
    name: str
    post: str
    year: int

@dataclass
class Staff:
```

```

workers: List[Worker] = field(default_factory=lambda: [])

def add(self, name, post, year):
    # Получить текущую дату.
    today = date.today()

    if year < 0 or year > today.year:
        raise IllegalYearError(year)

    self.workers.append(
        Worker(
            name=name,
            post=post,
            year=year
        )
    )

    self.workers.sort(key=lambda worker: worker.name)

def __str__(self):
    # Заголовок таблицы.
    table = []
    line = '+-{}-+-{}-+-{}-+-'.format(
        '-' * 4,
        '-' * 30,
        '-' * 20,
        '-' * 8
    )
    table.append(line)
    table.append(
        '| {:^4} | {:^30} | {:^20} | {:^8} |'.format(
            "№",
            "Ф.И.О.",
            "Должность",
            "Год"
        )
    )
    table.append(line)

    # Вывести данные о всех сотрудниках.
    for idx, worker in enumerate(self.workers, 1):
        table.append(
            '| {:>4} | {:<30} | {:<20} | {:>8} |'.format(
                idx,
                worker.name,
                worker.post,
                worker.year
            )
        )

    table.append(line)

    return '\n'.join(table)

def select(self, period):
    # Получить текущую дату.
    today = date.today()

    result = []
    for worker in self.workers:
        if today.year - worker.year >= period:
            result.append(worker)

    return result

```

```

def load(self, filename):
    with open(filename, 'r', encoding='utf8') as fin:
        xml = fin.read()

    parser = ET.XMLParser(encoding="utf8")
    tree = ET.fromstring(xml, parser=parser)

    self.workers = []
    for worker_element in tree:
        name, post, year = None, None, None

        for element in worker_element:
            if element.tag == 'name':
                name = element.text
            elif element.tag == 'post':
                post = element.text
            elif element.tag == 'year':
                year = int(element.text)

        if name is not None and post is not None and year is not
None:
            self.workers.append(
                Worker(
                    name=name,
                    post=post,
                    year=year
                )
            )

def save(self, filename):
    root = ET.Element('workers')
    for worker in self.workers:
        worker_element = ET.Element('worker')

        name_element = ET.SubElement(worker_element, 'name')
        name_element.text = worker.name

        post_element = ET.SubElement(worker_element, 'post')
        post_element.text = worker.post

        year_element = ET.SubElement(worker_element, 'year')
        year_element.text = str(worker.year)

        root.append(worker_element)

    tree = ET.ElementTree(root)
    with open(filename, 'wb') as fout:
        tree.write(fout, encoding='utf8', xml_declaration=True)

if __name__ == '__main__':
    # Выполнить настройку логгера.
    logging.basicConfig(
        filename='workers.log',
        level=logging.INFO
    )

    # Список работников.
    staff = Staff()

    # Организовать бесконечный цикл запроса команд.
    while True:
        try:

```

```

# Запросить команду из терминала.
command = input(">>> ").lower()

# Выполнить действие в соответствие с командой.
if command == 'exit':
    break

elif command == 'add':
    # Запросить данные о работнике.
    name = input("Фамилия и инициалы? ")
    post = input("Должность? ")
    year = int(input("Год поступления? "))

    # Добавить работника.
    staff.add(name, post, year)
    logging.info(
        f"Добавлен сотрудник: {name}, {post}, "
        f"поступивший в {year} году."
    )

elif command == 'list':
    # Вывести список.
    print(staff)
    logging.info("Отображен список сотрудников.")

elif command.startswith('select '):
    # Разбить команду на части для выделения номера года.
    parts = command.split(maxsplit=1)
    # Запросить работников.
    selected = staff.select(parts[1])

    # Вывести результаты запроса.
    if selected:
        for idx, worker in enumerate(selected, 1):
            print(
                '{:>4}: {}'.format(idx, worker.name)
            )
        logging.info(
            f"Найдено {len(selected)} работников со "
            f"стажем более {parts[1]} лет."
        )
    else:
        print("Работники с заданным стажем не найдены.")
        logging.warning(
            f"Работники со стажем более {parts[1]} лет не
найденны."
        )

elif command.startswith('load '):
    # Разбить команду на части для имени файла.
    parts = command.split(maxsplit=1)
    # Загрузить данные из файла.
    staff.load(parts[1])
    logging.info(f"Загружены данные из файла {parts[1]}.")

elif command.startswith('save '):
    # Разбить команду на части для имени файла.
    parts = command.split(maxsplit=1)
    # Сохранить данные в файл.
    staff.save(parts[1])
    logging.info(f"Сохранены данные в файл {parts[1]}.")

elif command == 'help':

```

```

# Вывести справку о работе с программой.
print("Список команд:\n")
print("add - добавить работника;")
print("list - вывести список работников;")
print("select <стаж> - запросить работников со стажем;")
print("load <имя_файла> - загрузить данные из файла;")
print("save <имя_файла> - сохранить данные в файл;")
print("help - отобразить справку;")
print("exit - завершить работу с программой.")

else:
    raise UnknownCommandError(command)

except Exception as exc:
    logging.error(f"Ошибка: {exc}")
    print(exc, file=sys.stderr)

```

Для проверки результатов работы программы создадим несколько записей с сотрудниками и допустим ошибку в написании команды (рис. 1).

```

C:\Users\Andrey\anaconda3\envs\lab_oop_4\python.exe C:\Users\Andrey\D
>>> add
Фамилия и инициалы? Петров В. П.
Должность? директор
Год поступления? 2000
>>> save
>>> save -> Unknown command

```

Рисунок 1 – Обработка исключения ввода неверной команды

Выполнение индивидуальных заданий:

Задание 8.

Необходимо решить следующую задачу: написать программу, которая запрашивает ввод двух значений. Если хотя бы одно из них не является числом, то должна выполняться конкатенация, т. е. соединение, строк. В остальных случаях введенные числа суммируются.

Код программы решения данной задачи:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class Addition:
    """
    Класс предназначенный для сложения двух чисел или конкатенации строк.
    """

    def __init__(self, first=None, second=None):
        """
        Конструктор класса Addition (Сложения).
        """

```

```

        :param first: Первое число;
        :param second: Второе число.
        """

        self.first = first
        self.second = second

    def input_values(self):
        """
        Метод для ввода значений с клавиатуры.
        """

        self.first = input("Введите первое значение: ")
        self.second = input("Введите второе значение: ")

    def calculate_sum(self):
        """
        Метод для сложения чисел.
        Сначала пробует преобразовать и сложить 2 числа, а если не
получилось,
        выбрасывает исключение с конкатенацией строк.
        """

        try:
            first_num = float(self.first)
            second_num = float(self.second)
            return first_num + second_num
        except ValueError:
            print("Вы ввели строку.")
            return str(self.first) + str(self.second)

if __name__ == "__main__":
    obj = Addition()
    obj.input_values()
    result = obj.calculate_sum()
    print(f"Результат: {result}")

```

Для тестирования данной задачи были написаны unit-тесты с различными вариантами ввода данных, а именно для случая с вводом двух чисел, с вводом числа и строки, строки и числа, двух строк:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest
from src.task_1 import Addition

class TestAddition(unittest.TestCase):

    def test_calculate_sum_with_integers(self):
        """
        Тест когда два целых числа.
        """

        obj = Addition("5", "10")
        result = obj.calculate_sum()
        self.assertEqual(result, 15)

```

```

def test_calculate_sum_with_floats(self):
    """
    Тест когда два вещественных числа.
    """

    obj = Addition("2.5", "3.5")
    result = obj.calculate_sum()
    self.assertEqual(result, 6.0)

def test_calculate_sum_with_invalid_numbers(self):
    """
    Тест когда одно из значений - строка.
    """

    obj = Addition("abc", "3")
    result = obj.calculate_sum()
    self.assertEqual(result, "abc3")

def test_calculate_sum_with_strings(self):
    """
    Тест, когда оба значения - это строки.
    """

    obj = Addition("hello", "world")
    result = obj.calculate_sum()
    self.assertEqual(result, "helloworld")

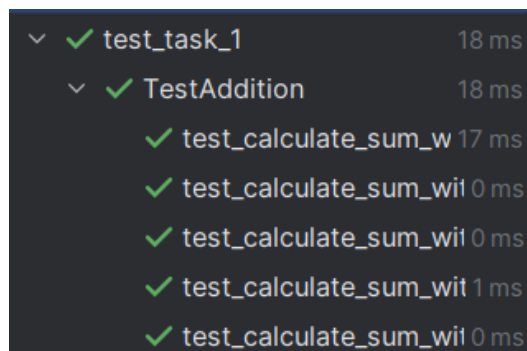
def test_calculate_sum_with_mixed_values(self):
    """
    Тест, когда второе значение - это строка.
    """

    obj = Addition("10", "world")
    result = obj.calculate_sum()
    self.assertEqual(result, "10world")

if __name__ == "__main__":
    unittest.main()

```

Все unit-тесты были завершены успешно, все предусмотренные случаи ввода обработались нормально (рис. 2).



```

v ✓ test_task_1 18 ms
  v ✓ TestAddition 18 ms
    ✓ test_calculate_sum_w 17 ms
    ✓ test_calculate_sum_wil 0 ms
    ✓ test_calculate_sum_wil 0 ms
    ✓ test_calculate_sum_wit 1 ms
    ✓ test_calculate_sum_wil 0 ms

```

Рисунок 2 – Тестирование задачи

Задание 9.

Необходимо решить следующую задачу: написать программу, которая будет генерировать матрицу из случайных целых чисел. Пользователь может указать число строк и столбцов, а также диапазон целых чисел. Произвести обработку ошибок ввода пользователя.

Код программы решения данной задачи:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

class MatrixGenerator:
    """
    Класс для создания и вывода матрицы указанной размерности
    со случайными элементами в указанном диапазоне.
    """

    def __init__(self):
        """
        Конструктор класса MatrixGenerator,
        инициализирует размерность, диапазон и саму матрицу.
        """
        self.num_columns = 0
        self.num_rows = 0
        self.range_start = 0
        self.range_end = 0
        self.matrix = []

    def read_size(self):
        """
        Метод для ввода размеров матрицы от пользователя.
        """
        try:
            self.num_rows = int(input("Введите число строк: "))
            self.num_columns = int(input("Введите число столбцов: "))
            if self.num_rows <= 0 or self.num_columns <= 0:
                raise ValueError("Размеры матрицы должны быть положительными числами!")
        except ValueError as e:
            print(f"Ошибка: {e}")
            raise

    def read_range(self):
        """
        Метод для ввода диапазона значений матрицы.
        """
        try:
            self.range_start = float(input("Введите начало диапазона: "))
            self.range_end = float(input("Введите конец диапазона: "))

            # Проверяем и округляем до целого, если введены дробные значения
            if self.range_start != int(self.range_start) or self.range_end != int(self.range_end):
                print("Матрица состоит из целых чисел, диапазон будет округлен!")
            self.range_start = round(self.range_start)
```

```

        self.range_end = round(self.range_end)

        self.range_start = int(self.range_start)
        self.range_end = int(self.range_end)

        # Меняем местами начало и конец диапазона, если они введены
наоборот
        if self.range_start > self.range_end:
            print("Начало диапазона больше конца. Меняем местами.")
            self.range_start, self.range_end = self.range_end,
self.range_start

        except ValueError:
            raise ValueError("Неверный ввод диапазона! Диапазон должен быть
числом.")
        except Exception:
            raise RuntimeError("Неизвестная ошибка при вводе диапазона.")

    def generate_matrix(self):
        """
        Метод для генерации матрицы заданной размерности и диапазона.
        """
        try:
            # Генерируем строки матрицы
            for i in range(self.num_rows):
                row = [] # Инициализируем новую строку
                for j in range(self.num_columns):
                    # Генерируем случайное число в заданном диапазоне
                    value = random.randint(self.range_start, self.range_end)
                    row.append(value) # Добавляем значение в строку
                self.matrix.append(row) # Добавляем строку в матрицу

        except Exception as e:
            print(f"Ошибка при создании матрицы: {e}")
            raise RuntimeError("Не удалось создать матрицу.")

    def display_matrix(self):
        """
        Метод для отображения матрицы.
        """
        for row in self.matrix:
            print(" ".join(map(str, row)))

if __name__ == "__main__":
    try:
        # Создадим объект матрицы
        generator = MatrixGenerator()

        # Введем значения размерности и диапазона
        generator.read_size()
        generator.read_range()

        # После указания атрибутов создадим матрицу
        generator.generate_matrix()

        # Выведем созданную матрицу
        print("Сформированная матрица:")
        generator.display_matrix()

    except Exception as error:
        # Если матрицу создать не удалось
        print(f"Программа завершилась с ошибкой: {error}")

```

Для данной задачи были созданы unit-тесты для различных вариантов ввода данных в том числе и для проверки выпадения исключений:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest
from unittest.mock import patch

from src.task_2 import MatrixGenerator

class TestMatrixGenerator(unittest.TestCase):
    def setUp(self):
        """Инициализация объекта перед каждым тестом."""
        self.generator = MatrixGenerator()

    def test_read_size_correct_input(self):
        """Тест корректного ввода размеров матрицы."""
        with patch("builtins.input", side_effect=["3", "4"]):
            self.generator.read_size()
            self.assertEqual(self.generator.num_rows, 3)
            self.assertEqual(self.generator.num_columns, 4)

    def test_read_size_invalid_input(self):
        """Тест на некорректный ввод размеров матрицы."""
        with patch("builtins.input", side_effect=["-2", "5"]):
            with self.assertRaises(ValueError) as cm:
                self.generator.read_size()
            self.assertEqual(str(cm.exception), "Размеры матрицы должны быть положительными числами!")

    def test_read_range_correct_input(self):
        """Тест корректного ввода диапазона."""
        with patch("builtins.input", side_effect=["3.5", "7.9"]):
            self.generator.read_range()
            self.assertEqual(self.generator.range_start, 4) # Округление
            self.assertEqual(self.generator.range_end, 8) # Округление вверх

    def test_read_range_reversed_input(self):
        """Тест диапазона, где начало больше конца."""
        with patch("builtins.input", side_effect=["10", "5"]):
            self.generator.read_range()
            self.assertEqual(self.generator.range_start, 5)
            self.assertEqual(self.generator.range_end, 10)

    def test_read_range_invalid_input(self):
        """Тест на некорректный ввод диапазона."""
        with patch("builtins.input", side_effect=["abc", "7"]):
            with self.assertRaises(ValueError) as cm:
                self.generator.read_range()
            self.assertEqual(str(cm.exception), "Неверный ввод диапазона! Диапазон должен быть числом.")

    def test_generate_matrix(self):
        """Тест генерации матрицы."""
        self.generator.num_rows = 3
        self.generator.num_columns = 3
        self.generator.range_start = 1
        self.generator.range_end = 5
        self.generator.generate_matrix()
        self.assertEqual(len(self.generator.matrix), 3)
```

```

        self.assertTrue(all(len(row) == 3 for row in self.generator.matrix))
        self.assertTrue(
            all(
                self.generator.range_start <= value <=
self.generator.range_end
                for row in self.generator.matrix
                for value in row
            )
        )

    def test_display_matrix(self):
        """Тест отображения матрицы."""
        self.generator.matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        with patch("builtins.print") as mock_print:
            self.generator.display_matrix()
            mock_print.assert_any_call("1 2 3")
            mock_print.assert_any_call("4 5 6")
            mock_print.assert_any_call("7 8 9")

if __name__ == "__main__":
    unittest.main()

```

Все unit-тесты завершились успешно, все предусмотренные случаи обработались нормально в том числе и исключения (рис. 3).

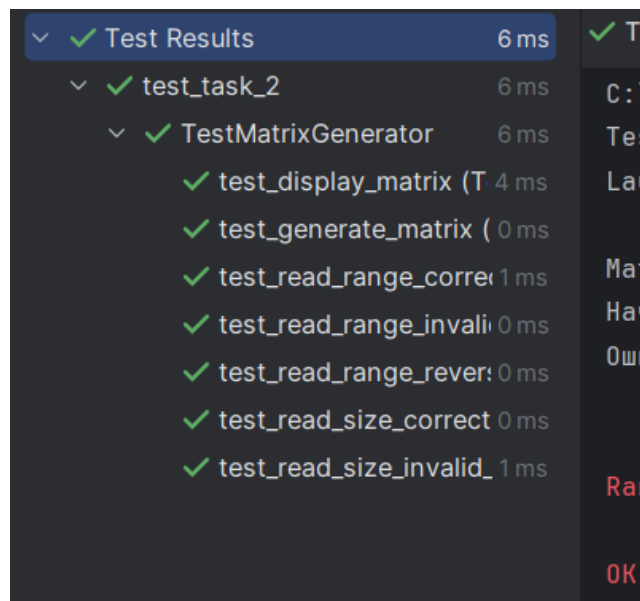


Рисунок 3 – Тестирование задачи с построением матрицы

Задание 1.

Необходимо дополнить задание 1 лабораторной работы 2.19 и добавить для программы с пользовательским интерфейсом возможность работы с исключениями и логгирование.

Лабораторная работа 2.19:

Необходимо использовать словарь, содержащий следующие ключи:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия: ввод с клавиатуры данных в список, состоящий из словарей заданной структуры; записи должны быть упорядочены по времени отправления поезда; вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры; если таких поездов нет, выдать на дисплей соответствующее сообщение (Вариант 26 (7), работа 2.8).

Код программы индивидуального задания 1 с добавлением исключений, логгированием и изменением структуры код под используемый класс «TrainManager»:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Необходимо выполнить индивидуальное задание 1 лабораторной работы 2.19,
# добавив возможность работы с исключениями и логгирование.

# Лабораторная работа 2.19:

# Необходимо использовать словарь, содержащий следующие ключи:
# - название пункта назначения;
# - номер поезда;
# - время отправления.
# Написать программу, выполняющую следующие действия: ввод с клавиатуры
# данных в список, состоящий из словарей заданной структуры; записи должны
# быть упорядочены по времени отправления поезда; вывод на экран информации
# о поездах, направляющихся в пункт, название которого введено с клавиатуры;
# если таких поездов нет, выдать на дисплей соответствующее сообщение.
# (Вариант 26 (7), работа 2.8).

import json
import os
import sys
import logging

class UnknownCommandError(Exception):
    """
    Класс пользовательского исключения в случае,
    если введенная команда является недопустимой.
    Сообщения об этой ошибке записываются в журнал (лог-файл) "trains.log".
    """

    def __init__(self, command, message="Неизвестная команда"):
```

```

    """
    Конструктор класса пользовательского исключения.
    При создании экземпляра исключения возможна запись в лог
    для сохранения информации об ошибочной команде.

    :param command: Неверная команда;
    :param message: Сообщение об ошибке.
    """

    self.command = command
    self.message = message
    super().__init__(message)

def __str__(self):
    """
    Метод вывода сообщения об ошибке (магический метод).

    :return: Неверная команда и сообщение об ошибке.
    """

    return f"{self.command} -> {self.message}"

class TrainManager:
    """
    Класс для управления списком поездов. Предоставляет методы для:
    - добавления поезда (add_train),
    - отображения всех поездов (list_trains),
    - выборки поездов по пункту назначения (select_trains),
    - загрузки поездов из JSON (load_from_json),
    - сохранения поездов в JSON (save_to_json).

    Все основные действия (добавление, загрузка, сохранение) сопровождаются
    записью
    в журнал (лог-файл) "trains.log".
    """

    def __init__(self):
        """
        Инициализировать пустой список поездов.
        """
        self.trains = []

    def add_train(self, departure_point, number_train, time_departure,
destination):
        """
        Добавить информацию о поезде в список.
        Добавленный поезд — это словарь такой структуры:
        {
            "departure_point": <пункт отправления>,
            "number_train": <номер поезда>,
            "time_departure": <время отправления>,
            "destination": <пункт назначения>
        }
        После добавления список поездов упорядочивается по времени
        отправления.

        :param departure_point: Пункт отправления;
        :param number_train: Номер поезда;
        :param time_departure: Время отправления;
        :param destination: Пункт назначения.

        После успешного добавления поезда информация вносится в лог-файл.
        """

```

```

        self.trains.append(
            {
                "departure_point": departure_point,
                "number_train": number_train,
                "time_departure": time_departure,
                "destination": destination
            }
        )
    # Сортировка по времени отправления.
    self.trains.sort(key=lambda train: train["time_departure"])
    logging.info(
        f"Добавлен поезд: пункт отправления={departure_point}, "
        f"№={number_train}, время={time_departure}, "
        f"пункт назначения={destination}"
    )

def list_trains(self):
    """
    Вернуть текущий список поездов.

    :return: Список поездов.
    """

    return self.trains

def select_trains(self, point_user):
    """
    Выбрать поезда, пункт назначения которых совпадает с point_user.
    Результат выборки логируется (указывается, сколько найдено поездов).

    :param point_user: Пункт назначения (введенный пользователем).
    :return: Список таких поездов (может быть пустым).
    """

    point_user = point_user.lower()
    selected = [
        train
        for train in self.trains
        if train["destination"].lower() == point_user
    ]
    logging.info(
        f"Выполнен поиск поездов по пункту назначения='{point_user}'. "
        f"Найдено {len(selected)} поезд(а). "
    )
    return selected

def load_from_json(self, filename):
    """
    Загрузить список поездов из указанного файла в формате JSON.
    Если файл отсутствует список остаётся пустым или прежним.
    При успешной загрузке записывается соответствующее сообщение в лог.
    Если файл не существует, в лог также добавляется предупреждение.

    :param filename: Имя файла для загрузки.
    """

    if os.path.exists(filename):
        with open(filename, "r", encoding="utf-8") as f:
            self.trains = json.load(f)
        logging.info(f"Данные успешно загружены из файла: {filename}.")
    else:
        logging.warning(f"Файл {filename} не найден. Загрузка не
выполнена.")

```

```

def save_to_json(self, filename):
    """
    Сохранить текущий список поездов в указанный файл в формате JSON.
    Если файл отсутствует, создается новый с таким же именем.
    При успешном сохранении выполняется запись в лог-файл.

    :param filename: Имя файла для сохранения.
    """

    if not filename.endswith(".json"):
        filename += ".json"

    with open(filename, "w", encoding="utf-8") as f:
        json.dump(self.trains, f, ensure_ascii=False, indent=4)

    logging.info(f"Данные сохранены в файл: {filename}.")


def print_trains(trains):
    """
    Напечатать таблицу поездов в табличном формате.
    Если список пуст, выводится сообщение о пустом списке.

    :param trains: Список поездов.
    """

    if not trains:
        print("Список поездов пуст или ничего не найдено.")
        return

    # Заголовок таблицы.
    line = '+-{}-+-{}-+-{}-+-{}-+-{}-+'.format(
        '-' * 4,
        '-' * 20,
        '-' * 13,
        '-' * 18,
        '-' * 20,
    )
    print(line)
    print(
        '| {:^4} | {:^20} | {:^13} | {:^18} | {:^20} |'.format(
            "№",
            "Пункт отправления",
            "№ поезда",
            "Время отправления",
            "Пункт назначения"
        )
    )
    print(line)

    # Вывод данных о всех поездах.
    for idx, train in enumerate(trains, 1):
        print(
            '| {:>4} | {:<20} | {:<13} | {:>18} | {:<20} |'.format(
                idx,
                train["departure_point"],
                train["number_train"],
                train["time_departure"],
                train["destination"]
            )
        )
        print(line)

```



```

def main():
    """
    Главная функция, организующая цикл взаимодействия с пользователем.
    Также решается проблема `теневого имен` (одинаковые имена в основном коде
    и методах).
    Все введённые пользователем команды, а также возникшие ошибки,
    регистрируются в лог-файле "trains.log".
    """

    # Настраиваем логгирование:
    logging.basicConfig(
        filename="trains.log",      # Название файла, куда будут писаться логи
        level=logging.INFO,        # Уровень логгирования: INFO
        format="[% (levelname)s] %(message)s" # Формат сообщения в логге
    )

    # Создаём объект класса, список поездов
    manager = TrainManager()

    # Логгируем запуск программы:
    logging.info("Программа запущена.")

    while True:
        try:
            command = input(">>> ").strip().lower()
            logging.info(f"Введена команда: '{command}'")

            if command == "exit":
                logging.info("Программа завершена по команде 'exit'.")
                print("Программа завершена.")
                break

            elif command == "add":
                departure_point = input("Пункт отправления? ")
                number_train = input("Номер поезда? ")
                time_departure = input("Время отправления? ")
                destination = input("Пункт назначения? ")

                manager.add_train(
                    departure_point,
                    number_train,
                    time_departure,
                    destination
                )
                print("Поезд добавлен.")

            elif command == "list":
                trains_list = manager.list_trains()
                print_trains(trains_list)
                logging.info(f"Выведен список из {len(trains_list)}
поезд(ов).")

            elif command.startswith("select "):
                parts = command.split(maxsplit=1)
                point_user = parts[1]
                selected = manager.select_trains(point_user)
                print_trains(selected)

            elif command.startswith("load "):
                parts = command.split(maxsplit=1)
                filename = parts[1]
                manager.load_from_json(filename)
                print(f"Данные загружены из файла {filename}.")

```

```

elif command.startswith("save "):
    parts = command.split(maxsplit=1)
    filename = parts[1]
    manager.save_to_json(filename)
    print(f"Данные сохранены в файл {filename}.")

elif command == "help":
    print("Список доступных команд:")
    print("add - добавить поезд;")
    print("list - вывести список всех поездов;")
    print("select <пункт_назначения> - вывести поезда по пункту
назначения;")

    print("load <имя_файла> - загрузить данные из файла JSON;")
    print("save <имя_файла> - сохранить данные в файл JSON;")
    print("help - показать справку;")
    print("exit - завершить работу.")

else:
    raise UnknownCommandError(command)

except Exception as exc:
    logging.error(f"Произошла ошибка: {exc}")
    print(f"Ошибка: {exc}", file=sys.stderr)

if __name__ == "__main__":
    main()

```

Результаты работы программы: были введены 2 поезда с некоторой информацией и были использованы все доступные команды в том числе и неверные для проверки выпадения исключений (рис. 4).

```

>>> help
Список доступных команд:
add - добавить поезд;
list - вывести список всех поездов;
select <пункт_назначения> - вывести поезда по пункту назначения;
load <имя_файла> - загрузить данные из файла JSON;
save <имя_файла> - сохранить данные в файл JSON;
help - показать справку;
exit - завершить работу.
>>> add
Пункт отправления? Москва
Номер поезда? 001
Время отправления? 10:00
Пункт назначения? Ставрополь
Поезд добавлен.
>>> add
Пункт отправления? Омск
Номер поезда? 002
Время отправления? 11:00
Пункт назначения? Саратов
Поезд добавлен.

```

```

>>> list
+-----+-----+-----+-----+-----+-----+
| № | Пункт отправления | № поезда | Время отправления | Пункт назначения |
+-----+-----+-----+-----+-----+-----+
| 1 | Москва | 001 | 10:00 | Ставрополь |
+-----+-----+-----+-----+-----+-----+
| 2 | Омск | 002 | 11:00 | Саратов |
+-----+-----+-----+-----+-----+-----+

>>> select Саратов
+-----+-----+-----+-----+-----+-----+
| № | Пункт отправления | № поезда | Время отправления | Пункт назначения |
+-----+-----+-----+-----+-----+-----+
| 1 | Омск | 002 | 11:00 | Саратов |
+-----+-----+-----+-----+-----+-----+

>>> save поезда
Данные сохранены в файл поезда.

>>> update
>>> Ошибка: update -> Неизвестная команда

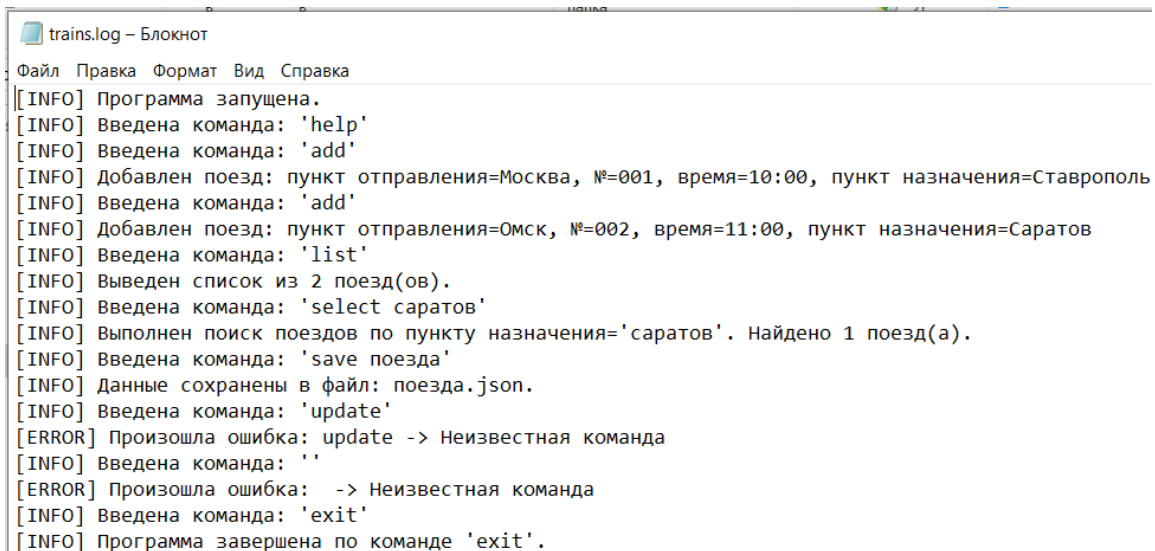
Ошибка: -> Неизвестная команда

>>> exit
Программа завершена.

```

Рисунок 4 – Результаты работы программы с пользовательским интерфейсом

После работы программы было создано 2 файла – с логом и с сохраненным списком поездов в формате JSON (рис. 5, 6).



```

trains.log – Блокнот
Файл Правка Формат Вид Справка
[INFO] Программа запущена.
[INFO] Введена команда: 'help'
[INFO] Введена команда: 'add'
[INFO] Добавлен поезд: пункт отправления=Москва, №=001, время=10:00, пункт назначения=Ставрополь
[INFO] Введена команда: 'add'
[INFO] Добавлен поезд: пункт отправления=Омск, №=002, время=11:00, пункт назначения=Саратов
[INFO] Введена команда: 'list'
[INFO] Выведен список из 2 поезд(ов).
[INFO] Введена команда: 'select саратов'
[INFO] Выполнен поиск поездов по пункту назначения='саратов'. Найдено 1 поезд(а).
[INFO] Введена команда: 'save поезда'
[INFO] Данные сохранены в файл: поезда.json.
[INFO] Введена команда: 'update'
[ERROR] Произошла ошибка: update -> Неизвестная команда
[INFO] Введена команда: ''
[ERROR] Произошла ошибка: -> Неизвестная команда
[INFO] Введена команда: 'exit'
[INFO] Программа завершена по команде 'exit'.

```

Рисунок 5 – Содержимое лог-файла

```

1  [
2      {
3          "departure_point": "Москва",
4          "number_train": "001",
5          "time_departure": "10:00",
6          "destination": "Ставрополь"
7      },
8      {
9          "departure_point": "Омск",
10         "number_train": "002",
11         "time_departure": "11:00",
12         "destination": "Саратов"
13     }
14 ]

```

Рисунок 6 – Содержимое файла формата JSON

Для тестирования некоторых методов и выпадения исключений был создан unit-тест и были протестированы некоторые методы (рис. 7).

```

test_individual_1 2 ms
TestTrainManager 2 ms
test_add_and_list_train 2 ms
test_select_trains (Пр 0 ms
test_unknown_comma 0 ms

```

```

C:\Users\Andrey\AppData\Local\py poetry\Cache\virt
Testing started at 21:11 ...
Launching unittests with arguments python -m unit

```

```

Ran 3 tests in 0.005s
OK
Process finished with exit code 0

```

Рисунок 7 – Успешное тестирование программы

Задание 2.

Необходимо дополнить предыдущее индивидуальное задание 1 используя модуль «logging» и сохранить не только список использованных в ходе работы программы команд и исключений, но и точное время их использования вплоть до миллисекунд. Код дополненной программы индивидуального задания 1:

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import json
import os
import sys
import logging

```

```

class UnknownCommandError(Exception):
    """
    Класс пользовательского исключения в случае,
    если введенная команда является недопустимой.
    Сообщения об этой ошибке записываются в журнал (лог-файл) "trains.log".
    """

    def __init__(self, command, message="Неизвестная команда"):
        """
        Конструктор класса пользовательского исключения.
        При создании экземпляра исключения возможна запись в лог
        для сохранения информации об ошибочной команде.

        :param command: Неверная команда;
        :param message: Сообщение об ошибке.
        """

        self.command = command
        self.message = message
        super().__init__(message)

    def __str__(self):
        """
        Метод вывода сообщения об ошибке (магический метод).

        :return: Неверная команда и сообщение об ошибке.
        """

        return f"{self.command} -> {self.message}"

class TrainManager:
    """
    Класс для управления списком поездов. Предоставляет методы для:
    - добавления поезда (add_train),
    - отображения всех поездов (list_trains),
    - выборки поездов по пункту назначения (select_trains),
    - загрузки поездов из JSON (load_from_json),
    - сохранения поездов в JSON (save_to_json).

    Все основные действия (добавление, загрузка, сохранение) сопровождаются
    записью
    в журнал (лог-файл) "trains.log".
    """

    def __init__(self):
        """
        Инициализировать пустой список поездов.
        """
        self.trains = []

    def add_train(self, departure_point, number_train, time_departure,
destination):
        """
        Добавить информацию о поезде в список.
        Добавленный поезд — это словарь такой структурой:
        {
            "departure_point": <пункт отправления>,
            "number_train": <номер поезда>,
            "time_departure": <время отправления>,
            "destination": <пункт назначения>
        }

```

После добавления список поездов упорядочивается по времени отправления.

```
:param departure_point: Пункт отправления;  
:param number_train: Номер поезда;  
:param time_departure: Время отправления;  
:param destination: Пункт назначения.
```

После успешного добавления поезда информация вносится в лог-файл.
"""

```
self.trains.append(  
    {  
        "departure_point": departure_point,  
        "number_train": number_train,  
        "time_departure": time_departure,  
        "destination": destination  
    }  
)  
# Сортировка по времени отправления.  
self.trains.sort(key=lambda train: train["time_departure"])  
logging.info(  
    f"Добавлен поезд: пункт отправления={departure_point}, "  
    f"N={number_train}, время={time_departure}, "  
    f"пункт назначения={destination}"  
)  
  
def list_trains(self):  
    """  
    Вернуть текущий список поездов.  
  
    :return: Список поездов.  
    """  
  
    return self.trains  
  
def select_trains(self, point_user):  
    """  
    Выбрать поезда, пункт назначения которых совпадает с point_user.  
    Результат выборки логируется (указывается, сколько найдено поездов).  
  
    :param point_user: Пункт назначения (введенный пользователем).  
    :return: Список таких поездов (может быть пустым).  
    """  
  
    point_user = point_user.lower()  
    selected = [  
        train  
        for train in self.trains  
        if train["destination"].lower() == point_user  
    ]  
    logging.info(  
        f"Выполнен поиск поездов по пункту назначения='{point_user}'. "  
        f"Найдено {len(selected)} поезд(а)."  
    )  
    return selected  
  
def load_from_json(self, filename):  
    """  
    Загрузить список поездов из указанного файла в формате JSON.  
    Если файл отсутствует список остаётся пустым или прежним.  
    При успешной загрузке записывается соответствующее сообщение в лог.  
    Если файл не существует, в лог также добавляется предупреждение.
```

```

:param filename: Имя файла для загрузки.
"""

if os.path.exists(filename):
    with open(filename, "r", encoding="utf-8") as f:
        self.trains = json.load(f)
        logging.info(f"Данные успешно загружены из файла: {filename}.")
else:
    logging.warning(f"Файл {filename} не найден. Загрузка не
выполнена.")

def save_to_json(self, filename):
    """
    Сохранить текущий список поездов в указанный файл в формате JSON.
    Если файл отсутствует, создается новый с таким же именем.
    При успешном сохранении выполняется запись в лог-файл.

    :param filename: Имя файла для сохранения.
    """

    if not filename.endswith(".json"):
        filename += ".json"

    with open(filename, "w", encoding="utf-8") as f:
        json.dump(self.trains, f, ensure_ascii=False, indent=4)

    logging.info(f"Данные сохранены в файл: {filename}.")

def print_trains(trains):
    """
    Напечатать таблицу поездов в табличном формате.
    Если список пуст, выводится сообщение о пустом списке.

    :param trains: Список поездов.
    """

    if not trains:
        print("Список поездов пуст или ничего не найдено.")
        return

    # Заголовок таблицы.
    line = '+-{}-+-{}-+-{}-+-{}-+-{}-+'.format(
        '-' * 4,
        '-' * 20,
        '-' * 13,
        '-' * 18,
        '-' * 20,
    )
    print(line)
    print(
        '| {:^4} | {:^20} | {:^13} | {:^18} | {:^20} |'.format(
            "№",
            "Пункт отправления",
            "№ поезда",
            "Время отправления",
            "Пункт назначения"
        )
    )
    print(line)

    # Вывод данных о всех поездах.
    for idx, train in enumerate(trains, 1):
        print(

```

```

        '| {:>4} | {:<20} | {:<13} | {:>18} | {:<20} |'.format(
            idx,
            train["departure_point"],
            train["number_train"],
            train["time_departure"],
            train["destination"]
        )
    )
    print(line)

def main():
    """
    Главная функция, организующая цикл взаимодействия с пользователем.
    Также решается проблема `теневого имен` (одинаковые имена в основном коде
    и методах).
    Все введенные пользователем команды, а также возникшие ошибки,
    регистрируются в лог-файле "trains.log".
    """

    # Настраиваем логгирование:
    logging.basicConfig(
        filename="trains.log", # Название файла, куда будут писаться логи
        level=logging.INFO, # Уровень логгирования: INFO (записывает и
        # предупреждения, и ошибки)
        format="%(asctime)s [%(levelname)s] %(message)s" # Формат сообщения
        в логге
    )

    # Создаём объект класса, список поездов
    manager = TrainManager()

    # Логируем запуск программы:
    logging.info("Программа запущена.")

    while True:
        try:
            # Считать команду.
            command = input(">>> ").strip().lower()
            logging.info(f"Введена команда: '{command}'")

            if command == "exit":
                logging.info("Программа завершена по команде 'exit'.")
                print("Программа завершена.")
                break

            elif command == "add":
                departure_point = input("Пункт отправления? ")
                number_train = input("Номер поезда? ")
                time_departure = input("Время отправления? ")
                destination = input("Пункт назначения? ")

                manager.add_train(
                    departure_point,
                    number_train,
                    time_departure,
                    destination
                )
                print("Поезд добавлен.")

            elif command == "list":
                trains_list = manager.list_trains()
                print_trains(trains_list)
                logging.info(f"Выведен список из {len(trains_list)}")

```



```

поезд(ов).")

    elif command.startswith("select "):
        parts = command.split(maxsplit=1)
        point_user = parts[1]
        selected = manager.select_trains(point_user)
        print_trains(selected)

    elif command.startswith("load "):
        parts = command.split(maxsplit=1)
        filename = parts[1]
        manager.load_from_json(filename)
        print(f"Данные загружены из файла {filename}.")

    elif command.startswith("save "):
        parts = command.split(maxsplit=1)
        filename = parts[1]
        manager.save_to_json(filename)
        print(f"Данные сохранены в файл {filename}.")

    elif command == "help":
        print("Список доступных команд:")
        print("add - добавить поезд;")
        print("list - вывести список всех поездов;")
        print("select <пункт_назначения> - вывести поезда по пункту
назначения;")
        print("load <имя_файла> - загрузить данные из файла JSON;")
        print("save <имя_файла> - сохранить данные в файл JSON;")
        print("help - показать справку;")
        print("exit - завершить работу.")

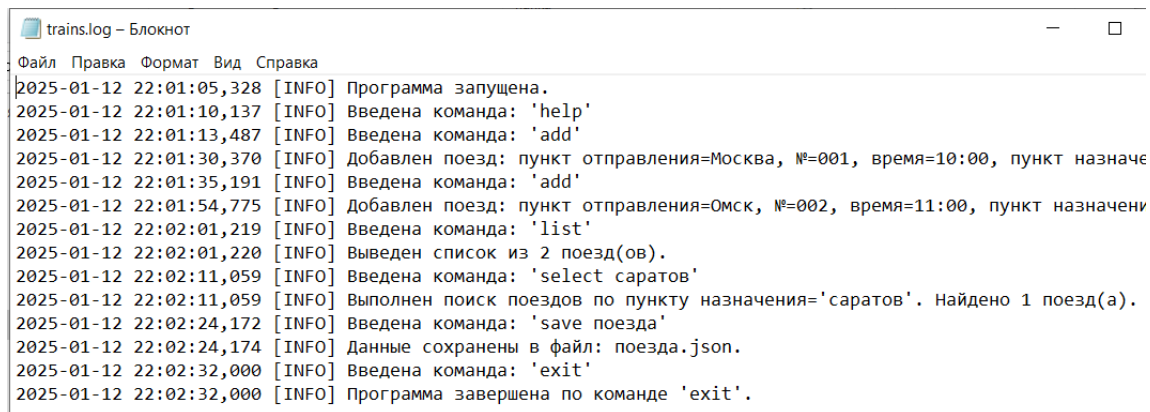
    else:
        raise UnknownCommandError(command)

except Exception as exc:
    # Логгируем ошибку.
    logging.error(f"Произошла ошибка: {exc}")
    # Выводим сообщение об ошибке в консоль.
    print(f"Ошибка: {exc}", file=sys.stderr)

if __name__ == "__main__":
    main()

```

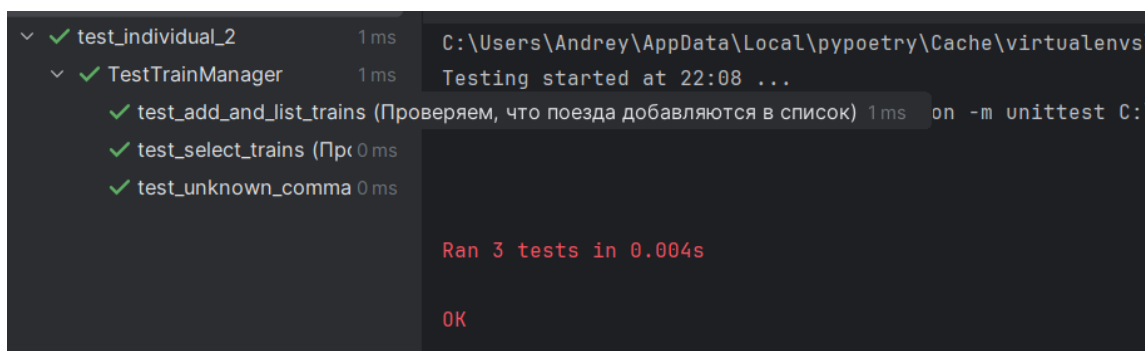
После завершения работы программы файл лога стал выглядеть следующим образом (рис. 8).



```
trains.log – Блокнот
Файл Правка Формат Вид Справка
2025-01-12 22:01:05,328 [INFO] Программа запущена.
2025-01-12 22:01:10,137 [INFO] Введена команда: 'help'
2025-01-12 22:01:13,487 [INFO] Введена команда: 'add'
2025-01-12 22:01:30,370 [INFO] Добавлен поезд: пункт отправления=Москва, №=001, время=10:00, пункт назначе
2025-01-12 22:01:35,191 [INFO] Введена команда: 'add'
2025-01-12 22:01:54,775 [INFO] Добавлен поезд: пункт отправления=Омск, №=002, время=11:00, пункт назначени
2025-01-12 22:02:01,219 [INFO] Введена команда: 'list'
2025-01-12 22:02:01,220 [INFO] Выведен список из 2 поезд(ов).
2025-01-12 22:02:11,059 [INFO] Введена команда: 'select саратов'
2025-01-12 22:02:11,059 [INFO] Выполнен поиск поездов по пункту назначения='саратов'. Найдено 1 поезд(а).
2025-01-12 22:02:24,172 [INFO] Введена команда: 'save поезда'
2025-01-12 22:02:24,174 [INFO] Данные сохранены в файл: поезда.json.
2025-01-12 22:02:32,000 [INFO] Введена команда: 'exit'
2025-01-12 22:02:32,000 [INFO] Программа завершена по команде 'exit'.
```

Рисунок 8 – Содержимое лог файла с указанием точного времени команд

Для тестирования некоторых методов программы индивидуального задания 2 и выпадения исключений был написан unit-тест. Результаты проверки программы индивидуального задания 2 (рис. 9).



```
✓ test_individual_2 1 ms C:\Users\Andrey\AppData\Local\py poetry\Cache\virtualenvs
  ✓ TestTrainManager 1 ms Testing started at 22:08 ...
    ✓ test_add_and_list_trains (Проверяем, что поезда добавляются в список) 1 ms on -m unittest C:
    ✓ test_select_trains (Проверяем, что поезда добавляются в список) 0 ms
    ✓ test_unknown_comma 0 ms

Ran 3 tests in 0.004s

OK
```

Рисунок 9 – Результаты тестирования программы индивидуального задания 2

Ссылка на репозиторий данной лабораторной работы:

https://github.com/AndreyPust/Object-Oriented_Programming_laboratory_work_4.git

Ответы на контрольные вопросы:

1. Какие существуют виды ошибок в языке программирования Python?

Синтаксические ошибки возникают при нарушении правил синтаксиса Python.

Исключения - ошибки, возникающие во время выполнения программы.

Некоторые распространенные виды исключений:

- ValueError – неверное значение;
- TypeError – недопустимый тип данных;
- IndexError – выход за пределы списка;

- `KeyError` – отсутствующий ключ в словаре;
- `ZeroDivisionError` – деление на ноль;
- `ImportError` – ошибка импорта модуля;
- `FileNotFoundError` – файл не найден.

2. Как осуществляется обработка исключений в языке программирования Python?

Обработка исключений осуществляется с помощью конструкции «try-except».

3. Для чего нужны блоки «finally» и «else» при обработке исключений?

«Finally» выполняется всегда, независимо от того, возникло исключение или нет. Обычно используется для освобождения ресурсов (например, закрытия файлов или соединений). «else» выполняется, если исключение не возникло.

4. Как осуществляется генерация исключений в языке Python?

Исключения можно генерировать вручную с помощью ключевого слова «raise».

5. Как создаются классы пользовательских исключений в языке Python?

Для создания пользовательских исключений создается класс, наследующийся от встроенного класса «Exception».

```
class MyCustomError(Exception):  
    def init(self, message):  
        super().init(message)
```

6. Каково назначение модуля «logging»?

Модуль «logging» используется для регистрации событий (логгирования) во время выполнения программы. Логи могут быть использованы для:

- отладки;
- мониторинга работы программы;
- анализа ошибок.

7. Какие уровни логгирования поддерживаются модулем «logging»? Приведите примеры, в которых могут быть использованы сообщения с этим уровнем журналирования.

Модуль «logging» поддерживает следующие уровни логгирования:

- DEBUG: Детальная информация для отладки, `logging.debug("Переменная x равна 10");`
- INFO: Общая информация о выполнении программы, `logging.info("Запущен процесс обработки данных");`
- WARNING: Уведомления о возможных проблемах, `logging.warning("Место на диске заканчивается");`
- ERROR: Ошибки, препятствующие нормальной работе программы, `logging.error("Не удалось открыть файл");`
- CRITICAL: Критические ошибки, требующие немедленного вмешательства, `logging.critical("Сбой в системе: немедленно обратитесь к администратору");`

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с исключениями при написании программ с помощью языка программирования Python версии 3.x, а также было изучено понятие логгирования и модуль «logging».