Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

институт

Кафедра вычислительной техники

кафедра

ПРАКТИЧЕСКАЯ 0

Преподаватель _____ Д. С. Романова
подпись, дата инициалы, фамилия

Студент КИ20-07Б, 031939247 _____ И. В. Тимченко
номер группы, зачетной книжкой подпись, дата инициалы, фамилия

Красноярск 2025

СОДЕРЖАНИЕ

Задача 1: Учет урожая

Напишите программу, которая позволяет фермеру вести учет урожая различных культур. Используя структуру, программа должна хранить информацию о названии культуры, площади посева, урожайности и общем объеме урожая. Используя циклы, фермер сможет вводить данные по нескольким культурам, а программа будет рассчитывать и выводить общий объем урожая за сезон.

Основная программа на Python (с использованием исключений):

```
class Crop:
    """Класс для представления культуры."""

    def __init__(self, name: str, area: float, yield_per_hectare: float):
        """
        Инициализация культуры.

        :param name: Название культуры
        :param area: Площадь посева в гектарах
        :param yield_per_hectare: Урожайность в тоннах на гектар
        """
        self.name = name
        self.area = area  # Площадь посева в гектарах
        self.yield_per_hectare = yield_per_hectare  # Урожайность в тоннах на гектар

    def total_yield(self) -> float:
        """Расчет общего объема урожая для данной культуры."""
        return self.area * self.yield_per_hectare


class Farm:
    """Класс для представления фермы."""

    def __init__(self):
        """Инициализация фермы с пустым списком культур."""
        self.crops = []

    def add_crop(self, crop: Crop):
        """Добавление культуры на ферму.

        :param crop: Экземпляр класса Crop
        """
        self.crops.append(crop)
```

```
    def total_yield(self) -> float:
        """Расчет общего объема урожая всех культур на ферме."""
        return sum(crop.total_yield() for crop in self.crops)


def main():
    """Основная функция для взаимодействия с пользователем."""
    farm = Farm()

    while True:
        try:
            name = input("Введите название культуры (или 'exit' для выхода): ")
            if name.lower() == 'exit':
                break

            area = float(input("Введите площадь посева (в гектарах): "))
            yield_per_hectare = float(input("Введите урожайность (в тоннах на
гектар): "))

            if area < 0 or yield_per_hectare < 0:
                raise ValueError("Площадь и урожайность должны быть
неотрицательными числами.")

            crop = Crop(name, area, yield_per_hectare)
            farm.add_crop(crop)

        except ValueError as e:
            print(f"Ошибка ввода: {e}. Пожалуйста, попробуйте снова.")

    print(f"Общий объем урожая за сезон: {farm.total_yield()} тонн")


if __name__ == "__main__":
    main()
```

Описание кода

1. Класс Crop:

   Хранит информацию о культуре: название, площадь и урожайность.

   Метод total_yield вычисляет общий объем урожая для данной культуры.

2. Класс Farm:

Хранит список культур.

Метод add_crop добавляет новую культуру в список.

Метод total_yield вычисляет общий объем урожая всех культур на ферме.

3. Функция main:

Обеспечивает пользовательский интерфейс для ввода данных о культурах.

Обрабатывает исключения, чтобы предотвратить ошибки ввода (например, если пользователь вводит отрицательные значения или некорректные типы данных).

Позволяет пользователю вводить данные до тех пор, пока он не введет 'exit'.

Юнит-тесты

Теперь добавим юнит-тесты для проверки функциональности программы. Будем использовать библиотеку unittest.

```
import unittest

class TestCrop(unittest.TestCase):
    def test_total_yield(self):
        """Тестирование расчета общего объема урожая для культуры."""
        crop = Crop("Пшеница", 10, 3)
        self.assertEqual(crop.total_yield(), 30)

class TestFarm(unittest.TestCase):
    def test_total_yield_multiple_crops(self):
        """Тестирование расчета общего объема урожая для нескольких культур."""
        farm = Farm()
        farm.add_crop(Crop("Пшеница", 10, 3))
        farm.add_crop(Crop("Кукуруза", 5, 4))
        self.assertEqual(farm.total_yield(), 50)

    def test_negative_area(self):
        """Тестирование обработки отрицательной площади."""
        with self.assertRaises(ValueError):
            Crop("Пшеница", -10, 3)
 def test_negative_yield_per_hectare(self):
        """Тестирование обработки отрицательной урожайности."""
        with self.assertRaises(ValueError):
            Crop("Пшеница", 10, -3)
```

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

Описание юнит-тестов

1. Тесты для класса Crop:

Проверяем правильность расчета общего объема урожая с помощью метода test_total_yield.

2. Тесты для класса Farm:

Проверяем правильность расчета общего объема урожая для нескольких культур с помощью метода test_total_yield_multiple_crops.

Проверяем обработку ошибок при вводе отрицательных значений площади и урожайности.

Задача 2: Контроль поголовья скота

Создайте программу для учета поголовья скота на ферме. Используя структуру, программа должна хранить информацию о виде животного, количестве голов, среднесуточном привесе и общем приросте за период. Используя циклы, фермер сможет вводить данные по различным видам животных, а программа будет рассчитывать и выводить общий прирост поголовья за определенный период.

Основная программа на Python (с использованием исключений):

```
class Animal:
    def __init__(self, species: str, head_count: int, daily_gain: float):
        self.species = species
        self.head_count = head_count
        self.daily_gain = daily_gain
        self.total_gain = 0

    def calculate_total_gain(self, days: int):
        if days < 0:
            raise ValueError("Количество дней не может быть отрицательным.")
        self.total_gain = self.head_count * self.daily_gain * days


def main():
    animals = []

    while True:
        try:
            species = input("Введите вид животного (или 'exit' для выхода): ")
            if species.lower() == 'exit':
                break

            head_count = int(input("Введите количество голов: "))
            daily_gain = float(input("Введите среднесуточный привес (в кг): "))

            animal = Animal(species, head_count, daily_gain)
            days = int(input("Введите количество дней для расчета прироста: "))
            animal.calculate_total_gain(days)

            animals.append(animal)

        except ValueError as e:
            print(f"Ошибка ввода: {e}")
```

```
    total_growth = sum(animal.total_gain for animal in animals)
    print(f"Общий прирост поголовья за указанный период: {total_growth} кг")


if __name__ == "__main__":
    main()
```

Юнит-тесты

```
import unittest

class TestAnimal(unittest.TestCase):
    def test_calculate_total_gain(self):
        animal = Animal("Коровы", 10, 1.5)
        animal.calculate_total_gain(5)
        self.assertEqual(animal.total_gain, 75)

    def test_calculate_total_gain_negative_days(self):
        animal = Animal("Коровы", 10, 1.5)
        with self.assertRaises(ValueError):
            animal.calculate_total_gain(-5)

    def test_animal_initialization(self):
        animal = Animal("Овцы", 20, 0.5)
        self.assertEqual(animal.species, "Овцы")
        self.assertEqual(animal.head_count, 20)
        self.assertEqual(animal.daily_gain, 0.5)

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

Объяснение кода

1. Класс Animal:

Содержит информацию о виде животного, количестве голов и среднем суточном привесе.

Метод calculate_total_gain рассчитывает общий прирост за определенное количество дней.

2. Основная программа:

В цикле запрашивает у пользователя ввод данных о животных.

Обрабатывает исключения, если введены некорректные значения.

Подсчитывает и выводит общий прирост поголовья за указанный период.

3. Юнит-тесты:

Проверяют корректность работы методов класса Animal, включая обработку исключений.

Задача 3: Расчет потребности в кормах

Разработайте программу, которая рассчитывает потребность в кормах для животных на ферме. Используя структуру, программа должна хранить информацию о виде животного, суточной норме корма на одну голову и общем поголовье. Используя циклы, фермер сможет вводить данные по различным видам животных, а программа будет рассчитывать и выводить общую потребность в кормах за определенный период.

Основная программа на Python (с использованием исключений):

```
class Animal:
    def __init__(self, species: str, daily_feed: float, head_count: int):
        self.species = species
        self.daily_feed = daily_feed
        self.head_count = head_count

    def total_feed_needed(self, days: int) -> float:
        if days < 0:
            raise ValueError("Количество дней не может быть отрицательным.")
        return self.daily_feed * self.head_count * days


def main():
    animals = []

    while True:
        try:
            species = input("Введите вид животного (или 'exit' для выхода): ")
            if species.lower() == 'exit':
                break

            daily_feed = float(input("Введите суточную норму корма на одну голову (в кг): "))
            head_count = int(input("Введите количество голов: "))

            animal = Animal(species, daily_feed, head_count)
            days = int(input("Введите количество дней для расчета потребности в кормах: "))
            total_feed = animal.total_feed_needed(days)

            animals.append(animal)
```

```
            print(f"Потребность в кормах для {species} за {days} дней: {total_feed}
кг")

        except ValueError as e:
            print(f"Ошибка ввода: {e}")

    overall_feed = sum(animal.total_feed_needed(days) for animal in animals)
    print(f"Общая потребность в кормах за указанный период: {overall_feed} кг")


if __name__ == "__main__":
    main()
```

Юнит-тесты

```
import unittest

class TestAnimal(unittest.TestCase):
    def test_total_feed_needed(self):
        animal = Animal("Коровы", 10.0, 5)
        self.assertEqual(animal.total_feed_needed(3), 150.0)

    def test_total_feed_needed_negative_days(self):
        animal = Animal("Овцы", 5.0, 10)
        with self.assertRaises(ValueError):
            animal.total_feed_needed(-2)

    def test_animal_initialization(self):
        animal = Animal("Куры", 1.5, 100)
        self.assertEqual(animal.species, "Куры")
        self.assertEqual(animal.daily_feed, 1.5)
        self.assertEqual(animal.head_count, 100)

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

Объяснение кода

1. Класс Animal:

Содержит информацию о виде животного, суточной норме корма на одну голову и общем поголовье.

Метод total_feed_needed рассчитывает общую потребность в кормах за заданное количество дней.

2. Основная программа:

В цикле запрашивает у пользователя ввод данных о животных.

Обрабатывает исключения, если введены некорректные значения.

Подсчитывает и выводит потребность в кормах для каждого вида животного и общую потребность за указанный период.

3. Юнит-тесты:

Проверяют корректность работы методов класса Animal, включая обработку исключений и правильность расчетов.

Задача 4: Управление складом

Разработайте программу для управления складскими запасами. Программа должна позволять менеджеру склада вести учет различных товаров. Используя структуру данных, программа должна хранить информацию о наименовании товара, количестве на складе, цене за единицу и общем объеме стоимости товара на складе. Менеджер сможет вводить данные по нескольким товарам, а программа будет рассчитывать и выводить общую стоимость всех товаров на складе.

Требования:

1. Создать структуру для хранения информации о товаре.
2. Реализовать ввод данных о товарах с использованием циклов.
3. Рассчитать общий объем стоимости всех товаров.
4. Вывести итоговую информацию на экран.

Основная программа на Python (с использованием исключений):

```
class Product:
    def __init__(self, name: str, quantity: int, unit_price: float):
        self.name = name
        self.quantity = quantity
        self.unit_price = unit_price

    def total_value(self) -> float:
        """Возвращает общую стоимость товара на складе."""
        return self.quantity * self.unit_price


class Inventory:
    def __init__(self):
        self.products = []

    def add_product(self, product: Product):
        """Добавляет продукт в инвентарь."""
        self.products.append(product)

    def total_inventory_value(self) -> float:
        """Возвращает общую стоимость всех товаров на складе."""
        return sum(product.total_value() for product in self.products)

    def display_inventory(self):
        """Выводит информацию о товарах на складе."""
```

```
        print(f"\n{'Наименование':<20}{'Количество':<15}{'Цена за единицу':<20}
{'Общая стоимость':<20}")
        for product in self.products:
            total_value = product.total_value()
            print(f"{product.name:<20}{product.quantity:<15}{product.unit_price:<20}
{total_value:<20}")


def main():
    inventory = Inventory()

    while True:
        try:
            name = input("Введите наименование товара (или 'exit' для выхода): ")
            if name.lower() == 'exit':
                break

            quantity = int(input("Введите количество на складе: "))
            unit_price = float(input("Введите цену за единицу: "))

            product = Product(name, quantity, unit_price)
            inventory.add_product(product)

        except ValueError as e:
            print(f"Ошибка ввода: {e}. Пожалуйста, попробуйте еще раз.")

    # Вывод итоговой информации о товарах на складе
    inventory.display_inventory()

    # Вывод общей стоимости всех товаров на складе
    total_value = inventory.total_inventory_value()
    print(f"\nОбщая стоимость всех товаров на складе: {total_value:.2f} руб.")


if __name__ == "__main__":
    main()
```

Юнит-тесты

```
import unittest


class TestInventory(unittest.TestCase):
```

```
    def setUp(self):
        """Создаем инвентарь и добавляем тестовые продукты."""
        self.inventory = Inventory()
        self.inventory.add_product(Product("Товар 1", 10, 100.0))
        self.inventory.add_product(Product("Товар 2", 5, 200.0))

    def test_total_inventory_value(self):
        """Проверяем правильность расчета общей стоимости."""
        self.assertEqual(self.inventory.total_inventory_value(), 2000.0)

    def test_product_total_value(self):
        """Проверяем правильность расчета общей стоимости отдельного
товара."""
        product = Product("Товар 3", 3, 150.0)
        self.assertEqual(product.total_value(), 450.0)

    def test_add_product(self):
        """Проверяем добавление продукта в инвентарь."""
        product = Product("Товар 4", 2, 300.0)
        self.inventory.add_product(product)
        self.assertEqual(len(self.inventory.products), 3)

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

Объяснение кода

1. Класс Product:

Содержит информацию о товаре (наименование, количество и цена за единицу).

Метод total_value возвращает общую стоимость товара.

2. Класс Inventory:

Хранит список продуктов и методы для добавления продуктов, расчета общей стоимости и вывода информации о продуктах.

3. Функция main:

Позволяет пользователю вводить данные о товарах в цикле.

Обрабатывает исключения при вводе данных.

Выводит итоговую информацию о товарах и общую стоимость.

4. Юнит-тесты:

Используют библиотеку unittest для проверки функциональности классов Product и Inventory.

Включают тесты для проверки расчета общей стоимости и добавления продуктов

Задача 5: Управление библиотечным каталогом
Разработайте программу для управления каталогом книг в библиотеке. Программа должна позволять библиотекарю вести учет различных книг. Используя структуру данных, программа должна хранить информацию о названии книги, авторе, жанре, количестве экземпляров и общем количестве доступных книг. Библиотекарь сможет вводить данные по нескольким книгам, а программа будет рассчитывать и выводить общее количество книг в библиотеке.

Требования:
1. Создать структуру для хранения информации о книге.
2. Реализовать ввод данных о книгах с использованием циклов.
3. Рассчитать общее количество всех книг.
4. Вывести итоговую информацию на экран.

Роли:
1.Программисты пишут код
2.Аналитик – анализирует задачу и предлагает решение (ищет формулы и составляет схему и алгоритм задачи) контролирует весь процесс разработки, следит чтобы все требования поставленной задачи были выполнены.
3.Тестировщик поверяет на работоспособность разработанную задачу, участвует в обсуждении технического решения

Основная программа на Python (с использованием исключений):

```
class Book:
    def __init__(self, title: str, author: str, genre: str, total_copies: int):
        self.title = title
        self.author = author
        self.genre = genre
        self.total_copies = total_copies
        self.available_copies = total_copies  # Изначально все экземпляры доступны

    def __str__(self):
        return f"{self.title} by {self.author} | Genre: {self.genre} | Total Copies: 
{self.total_copies} | Available Copies: {self.available_copies}"


class LibraryCatalog:
    def __init__(self):
        self.books = []
```

```
    def add_book(self, book: Book):
        """Добавляет книгу в каталог."""
        self.books.append(book)

    def total_books(self) -> int:
        """Возвращает общее количество всех книг в библиотеке."""
        return sum(book.total_copies for book in self.books)

    def display_catalog(self):
        """Выводит информацию о всех книгах в каталоге."""
        print(f"\n{'Название':<30}{'Автор':<20}{'Жанр':<15}{'Всего
экземпляров':<20}{'Доступно':<15}")
        for book in self.books:
            print(f"{book.title:<30}{book.author:<20}{book.genre:<15}
{book.total_copies:<20}{book.available_copies:<15}")


def main():
    catalog = LibraryCatalog()

    while True:
        try:
            title = input("Введите название книги (или 'exit' для выхода): ")
            if title.lower() == 'exit':
                break

            author = input("Введите автора книги: ")
            genre = input("Введите жанр книги: ")
            total_copies = int(input("Введите количество экземпляров: "))

            if total_copies < 0:
                raise ValueError("Количество экземпляров не может быть
отрицательным.")

            book = Book(title, author, genre, total_copies)
            catalog.add_book(book)

        except ValueError as e:
            print(f"Ошибка ввода: {e}. Пожалуйста, попробуйте еще раз.")

    # Вывод итоговой информации о книгах в каталоге
    catalog.display_catalog()

    # Вывод общего количества всех книг в библиотеке
    total_value = catalog.total_books()
```

```
    print(f"\nОбщее количество всех книг в библиотеке: {total_value}")


if __name__ == "__main__":
    main()
```

Юнит-тесты

```
import unittest

class TestLibraryCatalog(unittest.TestCase):

    def setUp(self):
        self.catalog = LibraryCatalog()

    def test_add_book(self):
        book = Book("1984", "George Orwell", "Dystopian", 5)
        self.catalog.add_book(book)
        self.assertEqual(len(self.catalog.books), 1)

    def test_total_books(self):
        self.catalog.add_book(Book("1984", "George Orwell", "Dystopian", 5))
        self.catalog.add_book(Book("Brave New World", "Aldous Huxley",
"Dystopian", 3))
        self.assertEqual(self.catalog.total_books(), 8)

    def test_display_catalog(self):
        self.catalog.add_book(Book("1984", "George Orwell", "Dystopian", 5))
        self.catalog.add_book(Book("Brave New World", "Aldous Huxley",
"Dystopian", 3))
        # Проверяем, что метод не вызывает ошибок
        try:
            self.catalog.display_catalog()
        except Exception as e:
            self.fail(f"display_catalog raised an exception: {e}")

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

Объяснение юнит-тестов

1. setUp: Метод, который выполняется перед каждым тестом. Здесь мы создаем новый экземпляр LibraryCatalog.

2. test_add_book: Проверяет, что книга была успешно добавлена в каталог.

3. test_total_books: Проверяет правильность подсчета общего количества книг.

4. test_display_catalog: Убеждается, что метод вывода каталога не вызывает ошибок при выполнении.