

### 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 обработкой исключений):

```
class Crop:
    def __init__(self, name: str, area: float, yield_per_hectare: float):
        """
        Инициализация экземпляра класса Crop.

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

    def total_yield(self) -> float:
        """
        Расчет общего объема урожая.

        :return: Общий объем урожая (в тоннах).
        """
        return self.area * self.yield_per_hectare # Общий объем урожая


def main():
    crops = [] # Список для хранения культур

    while True:
        try:
            # Ввод названия культуры
            name = input("Введите название культуры (или 'exit' для выхода): ")
            if name.lower() == 'exit':
                break # Выход из цикла при вводе 'exit'

            # Ввод площади посева
            area = float(input("Введите площадь посева (в гектарах): "))
```

```

    if area < 0:
        raise ValueError("Площадь посева не может быть отрицательной.")

    # Ввод урожайности
    yield_per_hectare = float(input("Введите урожайность (в тоннах на
гектар): "))
    if yield_per_hectare < 0:
        raise ValueError("Урожайность не может быть отрицательной.")

    # Создание экземпляра Crop
    crop = Crop(name, area, yield_per_hectare)
    crops.append(crop) # Добавление культуры в список

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

# Расчет общего объема урожая
total_volume = sum(crop.total_yield() for crop in crops)
print(f"Общий объем урожая за сезон: {total_volume} тонн")

# Условие запуска программы
if __name__ == "__main__":
    main()

```

### Комментарии к коду

#### 1. Класс Crop:

Конструктор `__init__`: принимает название культуры, площадь посева и урожайность, и инициализирует соответствующие атрибуты.

Метод `total_yield`: вычисляет общий объем урожая путем умножения площади на урожайность.

#### 2. Функция `main`:

Создает пустой список `crops` для хранения экземпляров класса `Crop`.

Использует бесконечный цикл для ввода данных о культурах. Если введено "exit", цикл завершается.

Для каждой культуры запрашиваются данные (название, площадь посева и урожайность). Если введены отрицательные значения, выбрасывается исключение `ValueError`.

Создается экземпляр класса Crop и добавляется в список.

После завершения ввода данных программа суммирует общий объем урожая и выводит результат.

### Юнит-тесты

```
import unittest
```

```
class TestCrop(unittest.TestCase):
    def test_total_yield(self):
        """Тестирование расчета общего объема урожая."""
        crop = Crop("Пшеница", 10, 3)
        self.assertEqual(crop.total_yield(), 30) # Ожидаемый объем урожая: 10 * 3 =
30

    def test_negative_area(self):
        """Тестирование обработки отрицательной площади."""
        Crop("Рис", -5, 4)

    def test_negative_yield(self):
        """Тестирование обработки отрицательной урожайности."""
        Crop("Кукуруза", 5, -2)

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

### Комментарии к тестам

#### 1. Класс TestCrop:

Наследует от unittest.TestCase, что позволяет использовать функции тестирования.

#### 2. Метод test\_total\_yield:

Проверяет, правильно ли рассчитывается общий объем урожая для заданной площади и урожайности.

#### 3. Методы test\_negative\_area и test\_negative\_yield:

Проверяют, выбрасываются ли исключения ValueError, когда площадь или урожайность отрицательные.

## 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 обработкой исключений):

```
class Animal:
    def __init__(self, species: str, count: int, daily_gain: float):
        """
        Инициализация экземпляра класса Animal.

        :param species: Вид животного.
        :param count: Количество голов.
        :param daily_gain: Среднесуточный привес (в кг).
        """
        self.species = species # Вид животного
        self.count = count # Количество голов
        self.daily_gain = daily_gain # Среднесуточный привес

    def total_gain(self, days: int) -> float:
        """
        Расчет общего прироста за заданный период.

        :param days: Количество дней.
        :return: Общий прирост (в кг).
        """
        return self.count * self.daily_gain * days # Общий прирост

def main():
    animals = [] # Список для хранения животных

    while True:
        try:
            # Ввод вида животного
            species = input("Введите вид животного (или 'exit' для выхода): ")
            if species.lower() == 'exit':
                break # Выход из цикла при вводе 'exit'
```

```

# Ввод количества голов
count = int(input("Введите количество голов: "))
if count < 0:
    raise ValueError("Количество голов не может быть отрицательным.")

# Ввод среднесуточного привеса
daily_gain = float(input("Введите среднесуточный привес (в кг): "))
if daily_gain < 0:
    raise ValueError("Среднесуточный привес не может быть отрицательным.")

# Ввод количества дней
days = int(input("Введите количество дней: "))
if days < 0:
    raise ValueError("Количество дней не может быть отрицательным.")

# Создание экземпляра Animal
animal = Animal(species, count, daily_gain)
animals.append(animal) # Добавление животного в список

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

# Расчет общего прироста поголовья за период
total_gain = sum(animal.total_gain(days) for animal in animals)
print(f"Общий прирост поголовья за период: {total_gain} кг")

# Условие запуска программы
if __name__ == "__main__":
    main()

```

### Комментарии к коду

#### 1. Класс Animal:

Конструктор `__init__`: принимает вид животного, количество голов и среднесуточный привес, и инициализирует соответствующие атрибуты.

Метод `total_gain`: вычисляет общий прирост за заданное количество дней.

#### 2. Функция main:

Создает пустой список `animals` для хранения экземпляров класса `Animal`.

Использует бесконечный цикл для ввода данных о животных. Если введено "exit", цикл завершается.

Для каждого животного запрашиваются данные (вид, количество голов, среднесуточный привес и количество дней). Если введены отрицательные значения, выбрасывается исключение `ValueError`.

Создается экземпляр класса `Animal` и добавляется в список.

После завершения ввода данных программа суммирует общий прирост и выводит результат.

### Юнит-тесты

```
import unittest
class TestAnimal(unittest.TestCase):
    def test_total_gain(self):
        """Тестирование расчета общего прироста."""
        animal = Animal("Коровы", 10, 2) # 10 коров с привесом 2 кг в день
        self.assertEqual(animal.total_gain(5), 100) # Ожидаемый прирост: 10 * 2 * 5
        = 100
    def test_negative_count(self):
        """Тестирование обработки отрицательного количества голов."""
        Animal("Овцы", -3, 1)
    def test_negative_daily_gain(self):
        """Тестирование обработки отрицательного среднесуточного привеса."""
        Animal("Свиньи", 5, -1)
    def test_negative_days(self):
        """Тестирование обработки отрицательного количества дней."""
        animal = Animal("Куры", 20, 0.5)
        animal.total_gain(-10)
if __name__ == "__main__":
    unittest.main()
```

### Комментарии к тестам

#### 1. Класс `TestAnimal`:

Наследует от `unittest.TestCase`, что позволяет использовать функции тестирования.

#### 2. Метод `test_total_gain`:

Проверяет, правильно ли рассчитывается общий прирост для заданного количества голов и среднесуточного привеса за определенный период.

#### 3. Методы `test_negative_count`, `test_negative_daily_gain`, и `test_negative_days`:

Проверяют, выбрасываются ли исключения `ValueError`, когда количество голов, среднесуточный привес или количество дней отрицательные.

---

### 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 обработкой исключений):

```
class AnimalFeed:
    def __init__(self, species: str, daily_feed: float, total_heads: int):
        """
        Инициализация экземпляра класса AnimalFeed.

        :param species: Вид животного.
        :param daily_feed: Суточная норма корма на одну голову (в кг).
        :param total_heads: Общее поголовье.
        """
        self.species = species # Вид животного
        self.daily_feed = daily_feed # Суточная норма корма
        self.total_heads = total_heads # Общее поголовье

    def total_feed_needed(self, days: int) -> float:
        """
        Расчет общей потребности в кормах за заданный период.

        :param days: Количество дней.
        :return: Общая потребность в кормах (в кг).
        """
        return self.daily_feed * self.total_heads * days # Общая потребность

def main():
    animals = [] # Список для хранения информации о животных

    while True:
        try:
            # Ввод вида животного
            species = input("Введите вид животного (или 'exit' для выхода): ")
            if species.lower() == 'exit':
                break # Выход из цикла при вводе 'exit'
```

```

# Ввод суточной нормы корма
daily_feed = float(input("Введите суточную норму корма на одну голову (в
кг): "))
if daily_feed < 0:
    raise ValueError("Суточная норма корма не может быть
отрицательной.")

# Ввод общего поголовья
total_heads = int(input("Введите общее поголовье: "))
if total_heads < 0:
    raise ValueError("Общее поголовье не может быть отрицательным.")

# Ввод количества дней
days = int(input("Введите количество дней: "))
if days < 0:
    raise ValueError("Количество дней не может быть отрицательным.")

# Создание экземпляра AnimalFeed
animal_feed = AnimalFeed(species, daily_feed, total_heads)
animals.append(animal_feed) # Добавление информации о животном в
список

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

# Расчет общей потребности в кормах за период
total_feed = sum(animal.total_feed_needed(days) for animal in animals)
print(f"Общая потребность в кормах за период: {total_feed} кг")

# Условие запуска программы
if __name__ == "__main__":
    main()

```

### Комментарии к коду

#### 1. Класс AnimalFeed:

Конструктор `__init__`: принимает вид животного, суточную норму корма и общее поголовье, и инициализирует соответствующие атрибуты.

Метод `total_feed_needed`: вычисляет общую потребность в кормах за заданное количество дней.



## 2. Функция main:

Создает пустой список animals для хранения экземпляров класса AnimalFeed.

Использует бесконечный цикл для ввода данных о животных. Если введено "exit", цикл завершается.

Для каждого животного запрашиваются данные (вид, суточная норма корма, общее поголовье и количество дней). Если введены отрицательные значения, выбрасывается исключение ValueError.

Создается экземпляр класса AnimalFeed и добавляется в список.

После завершения ввода данных программа суммирует общую потребность в кормах и выводит результат.

### Юнит-тесты

```
import unittest

class TestAnimalFeed(unittest.TestCase):
    def test_total_feed_needed(self):
        """Тестирование расчета общей потребности в кормах."""
        animal_feed = AnimalFeed("Коровы", 10.0, 5) # 5 коров с нормой 10 кг
        self.assertEqual(animal_feed.total_feed_needed(7), 350.0) # Ожидаемая
        потребность: 10 * 5 * 7 = 350

    def test_negative_daily_feed(self):
        """Тестирование обработки отрицательной суточной нормы корма."""
        AnimalFeed("Овцы", -5.0, 10)

    def test_negative_total_heads(self):
        """Тестирование обработки отрицательного общего поголовья."""
        AnimalFeed("Свиньи", 3.0, -1)

    def test_negative_days(self):
        """Тестирование обработки отрицательного количества дней."""
        animal_feed = AnimalFeed("Куры", 1.5, 20)
        animal_feed.total_feed_needed(-10)

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

### Комментарии к тестам

### 1. Класс TestAnimalFeed:

Наследует от unittest.TestCase, что позволяет использовать функции тестирования.

### 2. Метод test\_total\_feed\_needed:

Проверяет, правильно ли рассчитывается общая потребность в кормах для заданного количества голов и суточной нормы за определенный период.

### 3. Методы test\_negative\_daily\_feed, test\_negative\_total\_heads, и test\_negative\_days:

Проверяют, выбрасываются ли исключения ValueError, когда суточная норма корма, общее поголовье или количество дней отрицательные.

---

#### 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 обработкой исключений):

```
class Product:
```

```
    def __init__(self, name: str, quantity: int, price_per_unit: float):
```

```
        """
```

```
        Инициализация экземпляра класса Product.
```

```
        :param name: Наименование товара.
```

```
        :param quantity: Количество товара на складе.
```

```
        :param price_per_unit: Цена за единицу товара.
```

```
        """
```

```
        self.name = name # Наименование товара
```

```
        self.quantity = quantity # Количество на складе
```

```
        self.price_per_unit = price_per_unit # Цена за единицу
```

```
    def total_value(self) -> float:
```

```
        """
```

```
        Расчет общей стоимости товара на складе.
```

```
        :return: Общая стоимость товара.
```

```
        """
```

```
        return self.quantity * self.price_per_unit # Общая стоимость
```

```
def main():
```

```
    products = [] # Список для хранения информации о товарах
```

```
    while True:
```

```

try:
    # Ввод наименования товара
    name = input("Введите наименование товара (или 'exit' для выхода): ")
    if name.lower() == 'exit':
        break # Выход из цикла при вводе 'exit'

    # Ввод количества товара
    quantity = int(input("Введите количество товара на складе: "))
    if quantity < 0:
        raise ValueError("Количество товара не может быть отрицательным.")

    # Ввод цены за единицу товара
    price_per_unit = float(input("Введите цену за единицу товара: "))
    if price_per_unit < 0:
        raise ValueError("Цена за единицу не может быть отрицательной.")

    # Создание экземпляра Product
    product = Product(name, quantity, price_per_unit)
    products.append(product) # Добавление информации о товаре в список

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

# Расчет общей стоимости всех товаров на складе
total_inventory_value = sum(product.total_value() for product in products)
print(f"Общая стоимость всех товаров на складе: {total_inventory_value:.2f}
у.е.")

# Условие запуска программы
if __name__ == "__main__":
    main()

```

## Комментарии к коду

### 1. Класс Product:

Конструктор `__init__`: принимает наименование товара, количество и цену за единицу, и инициализирует соответствующие атрибуты.

Метод `total_value`: вычисляет общую стоимость данного товара на складе.

### 2. Функция main:

Создает пустой список `products` для хранения экземпляров класса `Product`.

Использует бесконечный цикл для ввода данных о товарах. Если введено "exit", цикл завершается.

Для каждого товара запрашиваются данные (наименование, количество и цена). Если введены отрицательные значения, выбрасывается исключение ValueError.

Создается экземпляр класса Product и добавляется в список.

После завершения ввода данных программа суммирует общую стоимость всех товаров и выводит результат с двумя знаками после запятой.

### Юнит-тесты

```
import unittest

class TestProduct(unittest.TestCase):
    def test_total_value(self):
        """Тестирование расчета общей стоимости товара."""
        product = Product("Товар А", 10, 5.0) # 10 единиц по цене 5.0
        self.assertEqual(product.total_value(), 50.0) # Ожидаемая стоимость: 10 * 5 = 50

    def test_negative_quantity(self):
        """Тестирование обработки отрицательного количества товара."""
        Product("Товар В", -5, 10.0) # Ожидаем выброс исключения

    def test_negative_price(self):
        """Тестирование обработки отрицательной цены за единицу товара."""
        Product("Товар С", 10, -2.0) # Ожидаем выброс исключения

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

### Комментарии к тестам

#### 1. Класс TestProduct:

Наследует от unittest.TestCase, что позволяет использовать функции тестирования.

#### 2. Метод test\_total\_value:

Проверяет, правильно ли рассчитывается общая стоимость товара для заданного количества и цены за единицу.

### 3. Методы `test_negative_quantity` и `test_negative_price`:

Проверяют, выбрасываются ли исключения `ValueError`, когда количество или цена отрицательные.

---

### 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 проверяет на работоспособность разработанную задачу, участвует в обсуждении технического решения

---

### Основная программа на Python (с обработкой исключений):

```
class Book:
    def __init__(self, title: str, author: str, genre: str, total_copies: int):
        """
        Инициализация экземпляра класса Book.

        :param title: Название книги.
        :param author: Автор книги.
        :param genre: Жанр книги.
        :param total_copies: Общее количество экземпляров книги.
        """
        self.title = title # Название книги
        self.author = author # Автор книги
        self.genre = genre # Жанр книги
        self.total_copies = total_copies # Общее количество экземпляров книги

    def __str__(self):
        """
```

Строковое представление объекта Book.

:return: Информация о книге в виде строки.

"""

return f"{self.title} by {self.author} ({self.genre}) - {self.total\_copies} copies"

def main():

library = [] # Список для хранения информации о книгах

while True:

try:

# Ввод названия книги

title = input("Введите название книги (или 'exit' для выхода): ")

if title.lower() == 'exit':

break # Выход из цикла при вводе 'exit'

# Ввод автора книги

author = input("Введите автора книги: ")

# Ввод жанра книги

genre = input("Введите жанр книги: ")

# Ввод общего количества экземпляров книги

total\_copies = int(input("Введите общее количество экземпляров книги:

"))

if total\_copies < 0:

raise ValueError("Количество экземпляров не может быть отрицательным.")

# Создание экземпляра Book

book = Book(title, author, genre, total\_copies)

library.append(book) # Добавление информации о книге в список

except ValueError as e:

print(f"Ошибка ввода: {e}") # Обработка ошибок ввода

# Расчет общего количества всех книг в библиотеке

total\_books = sum(book.total\_copies for book in library)

# Вывод итоговой информации о книгах

print("\nИтоговая информация о книгах в библиотеке:")

for book in library:

print(book) # Вывод информации о каждой книге

print(f"\nОбщее количество книг в библиотеке: {total\_books}")



```
# Условие запуска программы
if __name__ == "__main__":
    main()
```

### Комментарии к коду

#### 1. Класс Book:

Конструктор `__init__`: принимает название, автора, жанр и общее количество экземпляров книги, и инициализирует соответствующие атрибуты.

Метод `__str__`: возвращает строковое представление объекта Book, которое будет удобно для вывода информации о книге.

#### 2. Функция main:

Создает пустой список `library` для хранения экземпляров класса Book.

Использует бесконечный цикл для ввода данных о книгах. Если введено "exit", цикл завершается.

Для каждой книги запрашиваются данные (название, автор, жанр и общее количество экземпляров). Если введено отрицательное значение для количества экземпляров, выбрасывается исключение `ValueError`.

Создается экземпляр класса Book и добавляется в список.

После завершения ввода данных программа суммирует общее количество экземпляров всех книг и выводит информацию на экран.

### Юнит-тесты

```
import unittest
```

```
class TestBook(unittest.TestCase):
```

```
    def test_total_copies(self):
```

```
        """Тестирование правильности хранения общего количества экземпляров
        книги."""
```

```
        book = Book("Тестовая книга", "Автор А", "Жанр А", 5)
```

```
        self.assertEqual(book.total_copies, 5) # Ожидаемое количество: 5
```

```
    def test_negative_copies(self):
```

```
        """Тестирование обработки отрицательного количества экземпляров."""
```

```
        Book("Тестовая книга", "Автор В", "Жанр В", -1) # Ожидаем выброс
        исключения
```

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

### Комментарии к тестам

#### 1. Класс TestBook:

Наследует от unittest.TestCase, что позволяет использовать функции тестирования.

#### 2. Метод test\_total\_copies:

Проверяет, правильно ли сохраняется общее количество экземпляров книги.

#### 3. Метод test\_negative\_copies:

Проверяет, выбрасывается ли исключение ValueError, когда количество экземпляров отрицательное.

<https://github.com/igor-timchenko/opppo4>

Ссылка на гитхаб.