

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 №1
дисциплины
«Объектно-ориентированное программирование»
Вариант 17

Выполнила:
Текеева Алима Даутовна
3 курс, группа ИВТ-б-о-23-2,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем», очная
форма обучения

(подпись)

Проверил:
Доцент департамента цифровых,
робототехнических систем и
электроники института перспективной
инженерии Воронкин Р.А

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2025 г.

Тема: Элементы объектно-ориентированного программирования в языке Python.

Цель работы: приобретение навыков по работе с классами и объектами при написании программ с помощью языка программирования Python.

Ссылка на репозиторий: <https://github.com/aranzaal/lab-1-oop>

Порядок выполнения работы:

После изучения теоретического материала из МУ рассмотрели пример с классом Book, демонстрирующий использование атрибутов класса и механизмов работы с экземплярами класса. Атрибуты material, cover и all_books определены на уровне класса, то есть каждый его экземпляр будет иметь доступ к этим значениям, даже если они не определены в его собственной области. В функции main() создаётся объект my_book, представляющий конкретный экземпляр класса Book. При обращении к атрибутам material, cover и all_books программа выводит значения, унаследованные от класса.

Код программы:

```
# Класс Book
class Book:
    # Атрибуты класса
    material = "paper"
    cover = "paperback"
    all_books = []

def main():
    # Экземпляр класса Book
    my_book = Book()
    print(" Экземпляр класса Book: \n")
    print(
        f" Material: {my_book.material}\n Cover: \
        {my_book.cover}\n Books: {my_book.all_books}"
    )

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

```

PS C:\Users\alima>
    > & C:/Users/alima/anaconda3/python.exe "c:/Зсем/Объектно-ориентированное программирование/book.py"
Экземпляр класса Book:

Material: paper
Cover:      paperback
Books: []
PS C:\Users\alima> 

```

Рисунок 1. Вывод экземпляра класса

Далее рассмотрели класс `River`, иллюстрирующий использование атрибутов экземпляра и методов экземпляра. Атрибут класса `all_rivers` является общей структурой данных, предназначенной для хранения всех созданных объектов типа `River`. Конструктор `__init__` принимает три параметра (название реки, её длину и `self`, представляющий конкретный экземпляр класса и позволяющий получить доступ к его атрибутам и методам).

Код программы:

```

class River:
    all_rivers = []

    def __init__(self, name: str, length: int) -> None:
        self.name = name
        self.length = length
        River.all_rivers.append(self)

    # Метод
    def get_info(self) -> str:
        return f"Длина {self.name} равна {self.length} км"

def main() -> None:
    volga = River("Волга", 3530)
    seine = River("Сена", 776)
    nile = River("Нил", 6852)

    for river in River.all_rivers:
        print(f"Name: {river.name}, Length: {river.length}")
        print(f"\n {volga.get_info()}\n {seine.get_info()}\n {nile.get_info()}")

if __name__ == "__main__":
    main()

```

```
PS C:\Users\alima> & C:/Users/alima/anaconda3/python.exe "c:/3sem/Объектно-ориентированное программирование/river.py"
Name: Волга, Length: 3530
Name: Сена, Length: 776
Name: Нил, Length: 6852

Длина Волга равна 3530 км
Длина Сена равна 776 км
Длина Нил равна 6852 км
```

Рисунок 2. Вывод данных экземпляра класса с использованием метода `get_info`

Рассмотрим уровни доступа на новом классе `Rectangle`. В приведенном примере для доступа к `_width` и `_height` используются специальные методы, но ничего не мешает обратиться к этим атрибутам напрямую.

Код программы:

```
class Rectangle:
    def __init__(self, width: int, height: int) -> None:
        self._width = width
        self._height = height

    def get_width(self) -> int:
        return self._width

    def set_width(self, w) -> None:
        self._width = w

    def get_height(self) -> int:
        return self._height

    def set_height(self, h) -> None:
        self._height = h

    def area(self) -> int:
        return self._width * self._height

def main() -> None:
    rect = Rectangle(10, 5)
    print(f"Width: {rect.get_width()}")
    print(f"Height: {rect.get_height()}")
    print(f"Area: {rect.area()}")

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

```
PS C:\Users\alima> & C:/Users/alima/anaconda3/python.exe "c:/3sem/Объектно-ориентированное программирование/rectangle.py"
Width: 10
Height: 5
Area: 50
_
```

Рисунок 3. Вывод данных экземпляра класса

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую, без использования внешнего атрибута, к нему уже не обратиться.

Код программы:

```
class Rectangle:
    def __init__(self, width: int, height: int) -> None:
        self.__width = width
        self.__height = height

    def get_width(self) -> int:
        return self.__width

    def set_width(self, w) -> None:
        self.__width = w

    def get_height(self) -> int:
        return self.__height

    def set_height(self, h) -> None:
        self.__height = h

    def area(self) -> int:
        return self.__width * self.__height

def main() -> None:
    rect = Rectangle(10, 5)
    print(f"Width: {rect.get_width()}")
    print(f"Height: {rect.get_height()}")
    print(f"Width: {rect._Rectangle__width}")
    print(f"Height: {rect._Rectangle__height}")
    print(f"Area: {rect.area()}")

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

```
PS C:\Users\alima> & C:/Users/alima/anaconda3/python.exe "c:/Зсем/Объектно-ориентированное программирование/rectangle1.py"
Width: 10
Height: 5
Width: 10
Height: 5
Area: 50
```

Рисунок 4. Вывод данных экземпляра класса

Рассмотрим класс Rectangle с использованием механизма свойств для. Параметры `__width` и `__height` объявлены как приватные, что обеспечивает максимальный уровень защиты внутреннего состояния объекта и исключает

возможность их несанкционированного изменения извне. Для доступа к данным используются свойства `width` и `height`, оформленные посредством декораторов `@property` и соответствующих сеттеров. Такая конструкция позволяет обращаться к атрибутам экземпляра в привычной форме (`rect.width`), сохраняя при этом возможность выполнять проверку корректности входных данных при их изменении. В сеттерах реализован простой механизм валидации: новое значение должно быть положительным; в противном случае возбуждается исключение `ValueError`.

Код программы:

```
class Rectangle:
    def __init__(self, width: int, height: int) -> None:
        self.__width = width
        self.__height = height

    @property
    def width(self) -> int:
        return self.__width

    @width.setter
    def width(self, w) -> None:
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self) -> int:
        return self.__height

    @height.setter
    def height(self, h) -> None:
        if h > 0:
            self.__height = h
        else:
            raise ValueError

    def area(self) -> int:
        return self.__width * self.__height

def main() -> None:
    rect = Rectangle(10, 5)
    print(f"Width: {rect.width}")
```

```

print(f"Height: {rect.height}")
print(f"Area: {rect.area()}")

if __name__ == "__main__":
    main()

```

```

PS C:\Users\alima> & C:/Users/alima/anaconda3/python.exe "c:/3sem/Объектно-ориентированное программирование/rectangle2.py"
Width: 10
Height: 5
Area: 50

```

Рисунок 5. Вывод данных экземпляра класса

Индивидуальное задание:

Задание 1. Парой называется класс с двумя полями, которые обычно имеют имена `first` и `second`. Требуется реализовать тип данных с помощью такого класса. Во всех заданиях обязательно должны присутствовать:

- метод инициализации `__init__`; метод должен контролировать значения аргументов на корректность;
- ввод с клавиатуры `read`;
- вывод на экран `display`.

Реализовать внешнюю функцию с именем `make_тип()`, где `тип` - тип реализуемой структуры. Функция должна получать в качестве аргументов значения для полей структуры и возвращать структуру требуемого типа. При передаче ошибочных параметров следует выводить сообщение и заканчивать работу.

В раздел программы, начинающийся после инструкции (`if __name__ == '__main__':` добавить код, демонстрирующий возможности разработанного класса.

17. Поле `first` — целое положительное число, координата курсора/указателя по горизонтали; поле `second` — целое положительное число, координата курсора по вертикали. Реализовать метод `changex()` — изменение горизонтальной координаты курсора; реализовать метод `change_y()` — изменение вертикальной координаты курсора. Методы должны проверять выход за границу экрана.

Рисунок 6. Вариант индивидуального задания

Создание класса `Cursor` с приватными атрибутами `first` и `second`:

```

class Cursor:
    def __init__(self,
                  first: int = 0,
                  second: int = 0,
                  width: int = 1920,
                  height: int = 1200) -> None:
        if width <= 0 or height <= 0:
            raise ValueError("Размеры экрана должны быть положительными")

        self.__width = int(width)
        self.__height = int(height)

        if not self._in_range(first, second):
            raise ValueError("Начальные координаты выходят за границы экрана")

        self.__first = int(first)
        self.__second = int(second)

    def _in_range(self, x: int, y: int) -> bool:
        return 0 <= x < self.__width and 0 <= y < self.__height

```

Реализация методов read и display для ввода и вывода данных.

```

def read(self) -> None:
    x = int(input("Введите координату X (по горизонтали): "))
    y = int(input("Введите координату Y (по вертикали): "))

    if not self._in_range(x, y):
        raise ValueError("Координаты выходят за границы экрана")

    self.__first = x
    self.__second = y

def display(self) -> None:
    print(f"Координаты курсора: ({self.__first}, {self.__second})")
    print(f"Размер экрана: {self.__width} x {self.__height}")

```

Свойства для чтения полей:

```

@property
def first(self) -> int:
    return self.__first

@property
def second(self) -> int:
    return self.__second

```

Реализация методов changex() и changey():

```

def changex(self, dx: int) -> None:
    new_x = self.__first + int(dx)

```

```

    if not self._in_range(new_x, self.__second):
        raise ValueError("Новая координата X выходит за границы экрана")

    self.__first = new_x

def changey(self, dy: int) -> None:
    new_y = self.__second + int(dy)

    if not self._in_range(self.__first, new_y):
        raise ValueError("Новая координата Y выходит за границы экрана")

    self.__second = new_y

```

Основная программа:

```

from cursor import Cursor

if __name__ == "__main__":
    try:
        cursor = Cursor(2, 2)
        cursor.display()

        cursor.read()
        cursor.display()

        print("Попробуем изменить координаты курсора:")
        dx = int(input("Смещение по X: "))
        dy = int(input("Смещение по Y: "))

        cursor.changex(dx)
        cursor.changey(dy)
        cursor.display()
    except ValueError as e:
        print("Ошибка:", e)

```

```

Координаты курсора: (16, 16)
Размер экрана: 80 x 25
PS C:\Зсем\Объектно-ориентированное программирование\lab 1> & C:/Users/alima/anaconda3/python.exe
Координаты курсора: (2, 2)
Размер экрана: 1920 x 1200
Введите координату X (по горизонтали): 200
Введите координату Y (по вертикали): 200
Координаты курсора: (200, 200)
Размер экрана: 1920 x 1200
Попробуем изменить координаты курсора:
Смещение по X: 228
Смещение по Y: 180
Координаты курсора: (428, 380)
Размер экрана: 1920 x 1200
PS C:\Зсем\Объектно-ориентированное программирование\lab 1> █

```

Рисунок 7. Результат работы класса

```

PS C:\3sem\Объектно-ориентированное программирование\lab 1> pytest tests/test_cursor.py
>>
===== test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0 -- C:\Users\alima\AppData\Local\Programs\Python\Python314\python.exe
cachedir: .pytest_cache
rootdir: C:\3sem\Объектно-ориентированное программирование\lab 1
configfile: pyproject.toml
collected 9 items

tests/test_cursor.py::test_initialization_default PASSED [ 11%]
tests/test_cursor.py::test_initialization_custom PASSED [ 22%]
tests/test_cursor.py::test_initialization_out_of_bounds PASSED [ 33%]
tests/test_cursor.py::test_changex_within_bounds PASSED [ 44%]
tests/test_cursor.py::test_changex_out_of_bounds PASSED [ 55%]
tests/test_cursor.py::test_changey_within_bounds PASSED [ 66%]
tests/test_cursor.py::test_changey_out_of_bounds PASSED [ 77%]
tests/test_cursor.py::test_read PASSED [ 88%]
tests/test_cursor.py::test_read_out_of_bounds PASSED [100%]

===== 9 passed in 0.03s =====

```

Рисунок 8. Тест задания

Задание 2. Составить программу с использованием классов и объектов для решения задачи. Во всех заданиях, помимо указанных в задании операций, обязательно должны быть реализованы следующие методы:

- метод инициализации `_init_`;
- ввод с клавиатуры `read`;
- вывод на экран `display`.

Номер варианта необходимо уточнить у преподавателя. В раздел программы, начинающийся после инструкции `if _name_ == '__main__':` добавить код, демонстрирующий возможности разработанного класса.

2. Создать класс `ModelWindow` для работы с моделями экранных окон. В качестве полей задаются: заголовок окна, координаты левого верхнего угла, размер по горизонтали, размер по вертикали, цвет окна, состояние «видимое/невидимое», состояние «с рамкой/без рамки». Координаты и размеры указываются в целых числах. Реализовать операции: передвижение окна по горизонтали, по вертикали; изменение высоты и/или ширины окна изменение цвета; изменение состояния, опрос состояния. Операции передвижения и изменения размера должны осуществлять проверку на пересечение границ экрана. Функция вывода на экран должна индуцировать состояние полей объекта.

Рисунок 9. Вариант индивидуального задания

Код программы:

```

from __future__ import annotations

class ModelWindow:

    def __init__(
        self,
        title: str = "Окно",
        x: int = 0,
        y: int = 0,
        width: int = 1920,
        height: int = 1200,
        color: str = "серый",
    ):

```

```

        visible: bool = True,
        framed: bool = True,
    ) -> None:
        self.__title = str(title)
        self.__x = int(x)
        self.__y = int(y)
        self.__width = int(width)
        self.__height = int(height)
        self.__color = str(color)
        self.__visible = bool(visible)
        self.__framed = bool(framed)

    @property
    def title(self) -> str:
        return self.__title

    @property
    def x(self) -> int:
        return self.__x

    @property
    def y(self) -> int:
        return self.__y

    @property
    def width(self) -> int:
        return self.__width

    @property
    def height(self) -> int:
        return self.__height

    @property
    def color(self) -> str:
        return self.__color

    @property
    def visible(self) -> str:
        return "Да" if self.__visible else "Нет"

    @property
    def framed(self) -> str:
        return "Да" if self.__framed else "Нет"

```

Реализация методов read и display для ввода и вывода данных.

```

def read(self) -> None:
    self.__title = input("Заголовок окна: ")
    self.__x = int(input("Координата X левого верхнего угла: "))
    self.__y = int(input("Координата Y левого верхнего угла: "))

```

```

self.__width = int(input("Ширина окна: "))
self.__height = int(input("Высота окна: "))
self.__color = input("Цвет окна: ")

visible_str = input("Окно видимо? (Да/Нет): ").strip().lower()
self.__visible = visible_str == "да"

framed_str = input("Окно с рамкой? (Да/Нет): ").strip().lower()
self.__framed = framed_str == "да"

def display(self) -> None:
    print("Модель окна:")
    print(f"  Заголовок: {self.__title}")
    print(f"  Позиция:   ({self.__x}, {self.__y})")
    print(f"  Размер:    {self.__width} x {self.__height}")
    print(f"  Цвет:      {self.__color}")
    print(f"  Видимо:    {self.visible}")
    print(f"  С рамкой:  {self.framed}")

```

Реализация операций изменения положения и/или размеров окна, изменения состояния (видимость, рамка), проверку выхода окна за границы экрана при таких изменениях и метод, который «индицирует состояние».

```

def move(self, dx: int, dy: int) -> None:
    self.__x += dx
    self.__y += dy

def resize(self, new_width: int, new_height: int) -> None:
    if new_width > 0:
        self.__width = new_width
    if new_height > 0:
        self.__height = new_height

def set_visible(self, state: bool) -> None:
    self.__visible = bool(state)

def set_framed(self, state: bool) -> None:
    self.__framed = bool(state)

```

Проверка пересечения двух окон по их прямоугольным областям:

```

def intersects(self, other: ModelWindow) -> bool:
    return not (
        self.__x + self.__width < other.__x or
        other.__x + other.__width < self.__x or
        self.__y + self.__height < other.__y or
        other.__y + other.__height < self.__y
    )

```

Основная программа:

```
from window_package import ModelWindow

def bold(text: str) -> str:
    return f"\033[1m{text}\033[0m"

if __name__ == '__main__':
    print(bold("Создание первого окна"))
    win1 = ModelWindow()
    win1.read()

    print("\n" + bold("Создание второго окна"))
    win2 = ModelWindow()
    win2.read()

    print("\n" + bold("Параметры первого окна"))
    win1.display()

    print("\n" + bold("Параметры второго окна"))
    win2.display()

    print("\n" + bold("Перемещение первого окна на (8, 2)"))
    win1.move(8, 2)
    win1.display()

    print("\n" + bold("Изменение размера первого окна на 200 x 200"))
    win1.resize(200, 200)
    win1.display()

    print("\n" + bold("Скрываем первое окно"))
    win1.set_visible(False)
    win1.display()

    print("\n" + bold("Убираем рамку у первого окна"))
    win1.set_framed(False)
    win1.display()

    print("\n" + bold("Проверка пересечения окон"))
    if win1.intersects(win2):
        print("Окна пересекаются.")
    else:
        print("Окна не пересекаются.")
```

```
PS C:\3sem\Объектно-ориентированное программирование\lab 1> & C:/Users/alima/anaconda3/python.exe
```

Создание первого окна

Заголовок окна: Первое окно
Координата X левого верхнего угла: 82
Координата Y левого верхнего угла: 82
Ширина окна: 820
Высота окна: 220
Цвет окна: красный
Окно видимо? (Да/Нет): Да
Окно с рамкой? (Да/Нет): Нет

Создание второго окна

Заголовок окна: Второе окно
Координата X левого верхнего угла: 90
Координата Y левого верхнего угла: 90
Ширина окна: 340
Высота окна: 652
Цвет окна: голубой
Окно видимо? (Да/Нет): Нет
Окно с рамкой? (Да/Нет): Да

Параметры первого окна

Модель окна:
Заголовок: Первое окно
Позиция: (82, 82)
Размер: 820 x 220
Цвет: красный
Видимо: Да
С рамкой: Нет

Параметры второго окна

Модель окна:
Заголовок: Второе окно
Позиция: (90, 90)
Размер: 340 x 652
Цвет: голубой
Видимо: Нет
С рамкой: Да

Рисунок 10. Начальное состояние окна и ввод новых данных

Перемещение первого окна на (8, 2)

Модель окна:

Заголовок: Первое окно
Позиция: (90, 84)
Размер: 820 x 220
Цвет: красный
Видимо: Да
С рамкой: Нет

Изменение размера первого окна на 200 x 200

Модель окна:

Заголовок: Первое окно
Позиция: (90, 84)
Размер: 200 x 200
Цвет: красный
Видимо: Да
С рамкой: Нет

Скрываем первое окно

Модель окна:

Заголовок: Первое окно
Позиция: (90, 84)
Размер: 200 x 200
Цвет: красный
Видимо: Нет
С рамкой: Нет

Убираем рамку у первого окна

Модель окна:

Заголовок: Первое окно
Позиция: (90, 84)
Размер: 200 x 200
Цвет: красный
Видимо: Нет
С рамкой: Нет

Проверка пересечения окон

Окна пересекаются.

PS C:\3sem\Объектно-ориентированное программирование\lab 1> █

Рисунок 11. Результат внесённых изменений

```

PS C:\3sem\Объектно-ориентированное программирование\lab 1> pytest tests/test_modelwindow.py
===== test session starts =====
platform win32 -- Python 3.14.2, pytest-9.0.2, pluggy-1.6.0 -- C:\Users\alima\AppData\Local\Programs\Python\Python314\python.exe
cachedir: .pytest_cache
rootdir: C:\3sem\Объектно-ориентированное программирование\lab 1
configfile: pyproject.toml
collected 5 items

tests/test_modelwindow.py::test_initialization_defaults PASSED [ 20%]
tests/test_modelwindow.py::test_move_and_resize PASSED [ 40%]
tests/test_modelwindow.py::test_set_visible_and_framed PASSED [ 60%]
tests/test_modelwindow.py::test_intersects PASSED [ 80%]
tests/test_modelwindow.py::test_read PASSED [100%]

===== 5 passed in 0.05s =====

```

Рисунок 12. Тест задания

Вывод: в процессе выполнения лабораторной работы были изучены и реализованы основные принципы объектно-ориентированного программирования в Python. Созданы классы с использованием конструктора, методов ввода и вывода, а также методов обработки и изменения данных. Отработаны понятия атрибутов класса и объекта, инкапсуляции, проверки корректности данных и управления состоянием объектов.

Контрольные вопросы:

1. Как осуществляется объявление класса в языке Python?

Класс объявляется с помощью ключевого слова `class`, за которым следует имя класса и двоеточие. Внутри по отступу пишутся атрибуты и методы.

```

class MyClass:
    def __init__(self, ...):
        ...
    def method(self, ...):
        ...

```

2. Чем атрибуты класса отличаются от атрибутов экземпляра?

Атрибуты класса задаются прямо в теле класса и общие для всех объектов: если их не переопределять в экземпляре, все объекты видят одно и то же значение.

Атрибуты экземпляра записываются через `self` (обычно в `__init__`: `self.x = 10`) и хранятся отдельно у каждого объекта, то есть для разных экземпляров могут быть разными и не влияют на других.

3. Каково назначение методов класса?

Методы класса определяют поведение объектов этого класса. Они являются функциями, которые описывают, что объекты класса могут делать. Методы могут работать как с атрибутами экземпляра, так и с атрибутами класса.

4. Для чего предназначен метод `__init__()` класса?

Метод `__init__()` — это конструктор экземпляра класса. Он автоматически вызывается при создании нового объекта (экземпляра) класса.

Его основное назначение — инициализировать атрибуты нового объекта, то есть задать ему начальное состояние.

5. Каково назначение `self`?

`self` — это ссылка на текущий экземпляр класса. Через `self` методы получают доступ к атрибутам и другим методам этого конкретного объекта. Это первый параметр любого метода экземпляра, хотя сам язык не запрещает назвать его иначе, но это сильно противоречит общепринятым соглашениям.

6. Как добавить атрибуты в класс?

Есть два уровня:

- Атрибут класса — прямо в теле класса, без `self`:

```
class Coord:
    screen_width = 1920    # общий для всех окон
```

- Атрибут экземпляра — внутри методов (обычно в `__init__`), через `self`:

```
class Coord:
    def __init__(self, x, y):
        self.x = x          # свой x у каждого объекта
        self.y = y
```

Атрибуты экземпляра можно добавлять и позже, в других методах (`self.color = "red"`), но в учебных задачах обычно всё задают в `__init__`.

7. Как осуществляется управление доступом к методам и атрибутам в языке Python?

В Python нет жёстких модификаторов (`private`, `public`), как в C++/Java, поэтому используют соглашения и `name mangling`:

- имена без подчёркиваний (`x`, `move`) считаются публичными — к ним можно свободно обращаться снаружи;
- имя с одним подчёркиванием (`_helper`) означает «служебное» — не рекомендуется трогать снаружи, это деталь реализации;
- имя с двумя подчёркиваниями в начале (`__secret`) при компиляции «переписывается» в `_ИмяКласса__secret`, что усложняет к нему доступ и защищает от случайного переопределения.

8. Каково назначение функции `isinstance`?

Функция `isinstance(object, classinfo)` проверяет, является ли объект `object` экземпляром класса `classinfo` или экземпляром любого из его подклассов (дочерних классов). Возвращает `True` или `False`. Она используется для проверки типа объекта.