ЛАБОРАТОРНАЯ РАБОТА №1. ОСНОВЫ ООП

Цель: освоить базовые принципы объектно-ориентированного программирования (ООП) в Python: классы и объекты, инкапсуляцию, наследование, полиморфизм, магические методы.

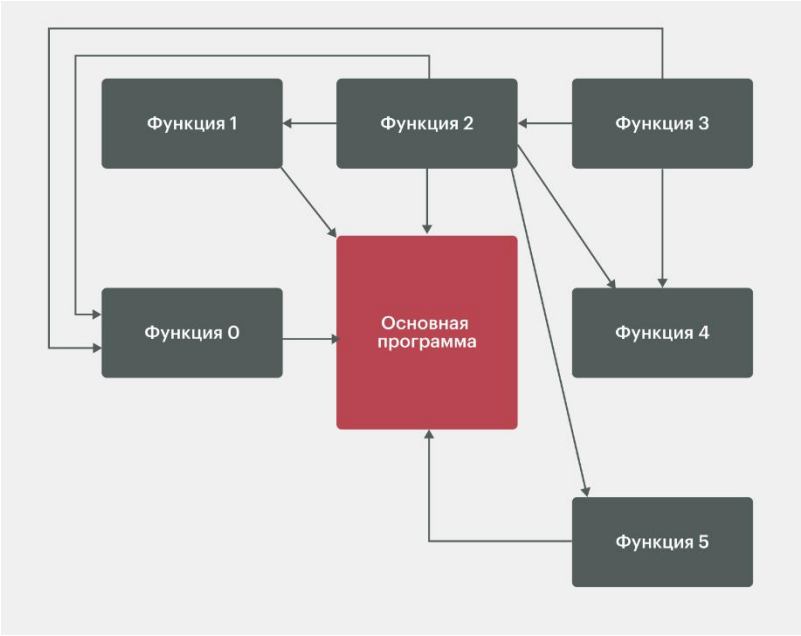
Теоретическая часть

Объекты

Основное понятие в ООП — объект. Это такой своеобразный контейнер, в котором сложены данные и прописаны действия, которые можно с этими данными совершать.

Чтобы понять, чем объекты так полезны и для чего их изобрели, сравним ООП с другой методикой разработки — процедурной. В ней весь код можно поделить на два вида: основную программу и вспомогательные функции, которые могут вызываться как программой, так и другими функциями.

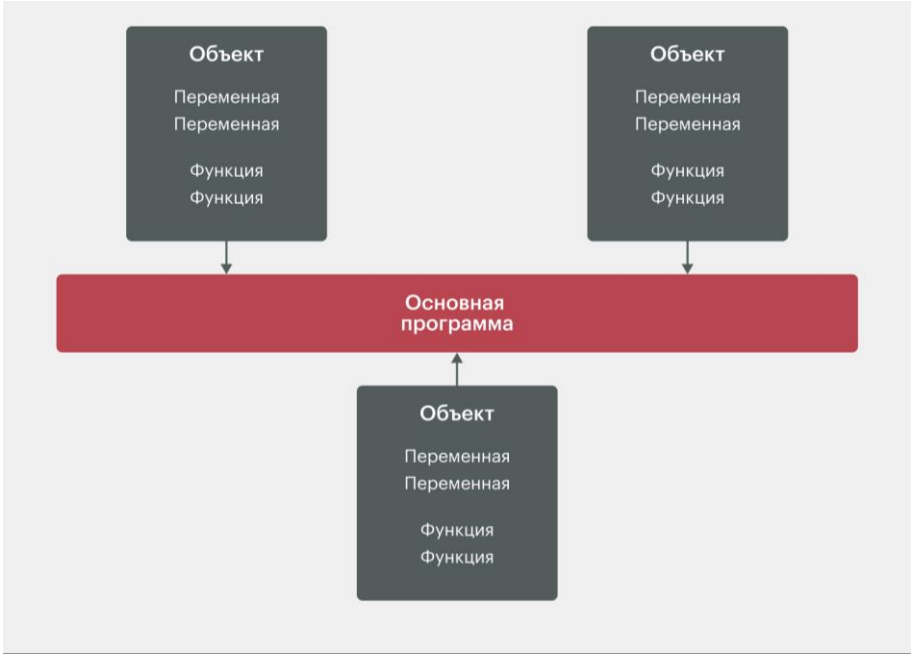
У такого программирования есть существенный недостаток — части кода сильно зависят друг от друга. Например, основная программа вызывает функцию, та вызывает вторую, та, в свою очередь, — третью. При этом, допустим, вторую функцию могут параллельно вызывать ещё несколько других, а также основная программа. Схематически вся эта процедурная путаница представлена на рисунке ниже.



Если мы изменим какую-нибудь функцию, то остальные части кода могут быть к этому не готовы — и сломаются. Тогда придётся переписывать ещё и их, а они, в свою очередь, завязаны на другие функции. В общем, проще будет написать новую программу с нуля.

Кроме того, в процедурном программировании нередко приходится дублировать код и писать похожие функции с небольшими различиями. Например, чтобы поддерживать совместимость разных частей программы друг с другом.

Логика ООП совершенно иная: к основной программе подключаются не функции, а **объекты**, внутри которых уже лежат собственные переменные и функции. Так выстраивается более иерархичная структура. Переменные внутри объектов называются **полями**, или **атрибутами**, а функции — **методами**.



Объекты независимы друг от друга и самодостаточны, так что, если мы сломаем что-то в одном объекте, это никак не отразится на других. Более того: даже если мы полностью изменим содержание объекта, но сохраним его поведение, весь код продолжит работать.

Классы

Каждый объект в ООП строится по определённому классу — абстрактной модели, описывающей, из чего состоит объект и что с ним можно делать.

Например, у нас есть класс «Кошка», обладающий атрибутами «порода», «окрас», «возраст» и методами «мяукать», «мурчать», «умываться», «спать». Присваивая атрибутам определённые значения, можно создавать вполне конкретные объекты. Допустим:

Порода = абиссинская.

Окрас = рыжий.

Возраст = 4.

Таким образом мы можем создать сколь угодно много разных кошек.

При этом любой объект класса «Кошка» (неважно, рыжая она, серая или чёрная) будет мяукать, мурчать, умываться и спать — если мы пропишем соответствующие методы.

ООП в Python

Чтобы создать класс нужно описать следующую конструкцию:

```
class SomeClass:
    # поля и методы класса SomeClass
```

Свойства классов устанавливаются с помощью простого присваивания:

```
class SomeClass:
    attr1 = 42
    attr2 = "Hello, World"
```

Методы объявляются как простые функции:

```
class SomeClass(object):
    def method1(self, x):
        # код метода
```

Обратите внимание на первый аргумент – self – общепринятое имя для ссылки на объект, в контексте которого вызывается метод. Этот параметр обязателен и отличает метод класса от обычной функции.

Все пользовательские атрибуты сохраняются в атрибуте __dict__, который является словарем.

Чтобы создать объект – экземпляр класса (инстанцировать) нужно выполнить:

```
class SomeClass(object):
    attr1 = 42
```

```
    def method1(self, x):
        return 2*x
obj = SomeClass()
obj.method1(6) # 12
obj.attr1 # 42
```

Можно создавать разные инстансы одного класса с заранее заданными параметрами с помощью **инициализатора** (специальный метод __init__). Для примера возьмем класс Point (точка пространства), объекты которого должны иметь определенные координаты:

```
class Point(object):
    def __init__(self, x, y, z):
        self.coord = (x, y, z)
p = Point(13, 14, 15)
p.coord # (13, 14, 15)
```

Специальные методы. Жизненный цикл объекта

Кроме инициализатора __init__ есть еще и метод __new__, который непосредственно создает новый экземпляр класса. Первым параметром он принимает ссылку на сам класс:

```
class SomeClass(object):
    def __new__(cls):
        print("new")
        return super(SomeClass, cls).__new__(cls)
    def __init__(self):
        print("init")
obj = SomeClass();
# new
# init
```

Также можно управлять не только созданием объекта, но и его удалением. Специально для этого предназначен метод-деструктор __del__.

```
class SomeClass(object):
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print('удаляется объект {} класса SomeClass'.format(self.name))
obj = SomeClass("John");
del obj # удаляется объект John класса SomeClass
```

На практике деструктор используется редко, в основном для тех ресурсов, которые требуют явного освобождения памяти при удалении объекта. Не следует совершать в нем сложные вычисления.

Принципы ООП на Python. Инкапсуляция

Часто нужно скрывать внутреннюю реализацию класса и давать доступ через контролируемый интерфейс.

В Python нет «жёстких» модификаторов доступа как в Java/C++, но есть соглашения:

- name — публичное поле/метод;
- _name — «защищённое» (для внутреннего использования);
- __name — «приватное» с *name mangling* (переименованием внутри класса).

```
class SomeClass:
    def _private(self):
        print("Это внутренний метод объекта")
obj = SomeClass()
obj._private() # это внутренний метод объекта
```

Если поставить перед именем атрибута два подчеркивания, к нему нельзя будет обратиться напрямую. Но все равно остается обходной путь:

```
class SomeClass():
    def __init__(self):
```

```
        self.__param = 42 # защищенный атрибут
obj = SomeClass()
obj.__param # AttributeError: 'SomeClass' object has no attribute '__param'
obj._SomeClass__param # 42
```

Принципы ООП на Python. Наследование

Наследование позволяет создать новый класс на основе существующего:

- базовый класс (родитель);
- производный класс (наследник).

Наследник:

- получает методы/атрибуты родителя;
- может переопределять (override) методы.

```
class Mammal():
   className = 'Mammal'
class Dog(Mammal):
   species = 'Canis lupus'
dog = Dog()
dog.className # Mammal
```

Принципы ООП на Python. Полиморфизм

Этот принцип позволяет применять одни и те же команды к объектам разных классов, даже если они выполняются по-разному. Например, помимо класса «Кошка», у нас есть никак не связанный с ним класс «Попугай» — и у обоих есть метод «спать». Несмотря на то, что кошки и попугаи спят по-разному (кошка сворачивается клубком, а попугай сидит на жёрдочке), для этих действий можно использовать одну команду.

Допустим у нас есть два класса — «Кошка» и «Попугай»:

```
class Cat:
  def sleep(self):
    print('Свернулся в клубок и сладко спит.')
```

```
class Parrot:
  def sleep(self):
    print('Сел на жёрдочку и уснул.')
```

А теперь пусть у нас есть метод, который ожидает, что ему на вход придёт объект, у которого будет метод sleep:

```
def homeSleep(animal):
    animal.sleep()
```

Это будет работать:

```
cat = Cat()
parrot = Parrot()
homeSleep(cat) # Свернулся в клубок и сладко спит.
homeSleep(parrot) # Сел на жёрдочку и уснул.
```

Хотя классы разные, их одноимённые методы работают похожим образом. Это и есть полиморфизм.

Еще один пример наследования и полиморфизма.

Пусть есть базовый класс – Фигура и два наследника – Прямоугольник и Круг. Каждая фигура имеет метод – Площадь. Попробуем реализовать вызов этого метода с использованием полиморфизма.

```
from math import pi
class Shape:
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def area(self):
        return self.w * self.h
class Circle(Shape):
    def __init__(self, r):
```

```
        self.r = r
    def area(self):
        return pi * self.r ** 2
shapes = [Rectangle(3, 4), Circle(5)]
for s in shapes:
    print(type(s).__name__, "->", s.area())
```

Пример демонстрирует: базовый класс с интерфейсом area(); разные реализации у наследников; единый цикл обработки — полиморфизм.

Задания

Для всех задач нужно реализовать соответствующие классы, методы и атрибуты. Протестировать работу на нескольких примерах.

1. Класс “Студент”

   Поля: ФИО, группа, список оценок.

   Методы: добавить оценку, средний балл, вывести информацию.
2. Класс “Прямоугольник”

   Поля: ширина и высота.

   Методы: площадь, периметр, сравнение прямоугольников по площади.
3. Класс “Таймер”

   Поля: секунды.

   Методы: добавить/убавить секунды, преобразовать в формат HH:MM:SS. Защита от отрицательного времени.
4. “Кошелёк”

   Поля: валюта (строка), баланс (скрытый).

   Доступ к балансу только через методы.

   Операции: пополнение, списание, запрет ухода в минус.
5. Класс “Книга” + класс “Библиотека”

   Book: автор, название, год, статус “в наличии”.

Library: хранит список книг, умеет добавлять, искать по автору/году, выдавать и возвращать книги.

6. “Транспорт”

   Базовый Transport: скорость, вместимость.

   Наследники: Car, Bus, Bicycle.

   Переопределить метод move(distance) (время в пути с учётом особенностей типа).

7. “Фигуры”

   Создать базовый Shape и 3–4 фигуры (круг, прямоугольник, треугольник, правильный многоугольник).

   Реализовать area() и perimeter().

   Написать функцию, которая принимает список фигур и возвращает фигуру с максимальной площадью.

8. “Вектор”

   Класс Vector2D(x, y):

   +, - (сложение/вычитание),

   * на число (масштабирование),

   magnitude как длина вектора.

9. СЛОЖНО! Классы: Account, Client, ATM.

   Логика:

- создание клиента/счёта,
- внесение/снятие,
- перевод между счетами,
- журнал операций (список объектов Transaction или строк).

Дополнительно: обработка ошибок (недостаточно средств, неверные суммы), аккуратный интерфейс через методы.