Лабораторная работа №4

Тема: «Объектно-ориентированное программирование на языке Python. Инкапсуляция данных, наследование, полиморфизм. Создание экземпляров объектов».

Python поддерживает объектно-ориентированную парадигму программирования, а это значит, что мы можем определить компоненты программы в виде классов.

Классы и объекты

Класс является шаблоном или формальным описанием объекта, а объект представляет экземпляр этого класса, его реальное воплощение. Можно провести следующую аналогию: у всех у нас есть некоторое представление о человеке – наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон – этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

С точки зрения кода класс объединяет набор функций и переменных, которые выполняют определенную задачу. Функции класса еще называют методами. Они определяют поведение класса. А переменные класса называют атрибутами – они хранят состояние класса

Класс определяется с помощью ключевого слова ***class***:

```
class название_класса:
    методы_класса
```

Для создания объекта класса используется следующий синтаксис:

```
название_объекта = название_класса([параметры])
```

Например, определим простейший класс ***Person***, который будет представлять человека:

```
class Person:
    name = "Tom"

    def display_info(self):
        print("Привет, меня зовут", self.name)

person1 = Person()
person1.display_info()         # Привет, меня зовут Tom

person2 = Person()
person2.name = "Sam"
person2.display_info()         # Привет, меня зовут Sam
```

Класс ***Person*** определяет атрибут ***name***, который хранит имя человека, и метод ***display_info***, с помощью которого выводится информация о человеке.

При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который, согласно условностям, называется ***self*** (в ряде языков программирования есть своего рода аналог – ключевое слово ***this***). Через эту ссылку внутри класса мы можем обратиться к методам или атрибутам этого же класса. В частности, через выражение ***self.name*** можно получить имя пользователя.

После определения класс ***Person*** создаем пару его объектов – ***person1*** и ***person2***. Используя имя объекта, мы можем обратиться к его методам и атрибутам. В данном случае у каждого из объектов вызываем метод ***display_info()***, который выводит строку на консоль, и у второго объекта также изменяем атрибут ***name***. При этом при вызове метода ***display_info*** не надо передавать значение для параметра ***self***.

Конструкторы

Для создания объекта класса используется конструктор. Так, выше, когда мы создавали объекты класса ***Person***, мы использовали конструктор по умолчанию, который неявно имеют все классы:

```
person1 = Person()
person2 = Person()
```

Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется ***__init()***. К примеру, изменим класс ***Person***, добавив в него конструктор:

```
class Person:
    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def display_info(self):
        print("Привет, меня зовут", self.name)


person1 = Person("Tom")
person1.display_info()        # Привет, меня зовут Tom
person2 = Person("Sam")
person2.display_info()        # Привет, меня зовут Sam
```

В качестве первого параметра конструктор также принимает ссылку на текущий объект – ***self***. Нередко в конструкторах устанавливаются атрибуты класса. Так, в данном случае в качестве второго параметра в конструктор передается имя пользователя, которое устанавливается для атрибута ***self.name***. Причем для атрибута необязательно определять в классе переменную ***name***, как это было в предыдущей версии класса ***Person***. Установка значения ***self.name = name*** уже неявно создает атрибут ***name***.

```
person1 = Person("Tom")
person2 = Person("Sam")
```

Деструктор

После окончания работы с объектом мы можем использовать оператор ***del*** для удаления его из памяти:

```
person1 = Person("Tom")
del person1     # удаление из памяти
#  person1.display_info()  # Этот метод работать не будет, так как
person1 уже удален из памяти
```

Стоит отметить, что после окончания работы скрипта все объекты автоматически удаляются из памяти.

Кроме того, мы можем определить в классе деструктор, реализовав встроенную функцию ***__del__***, который будет вызываться либо в результате вызова оператора ***del***, либо при автоматическом удалении объекта.

Например:

```
class Person:
    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def __del__(self):
        print(self.name,"удален из памяти")
    def display_info(self):
        print("Привет, меня зовут", self.name)


person1 = Person("Tom")
person1.display_info()  # Привет, меня зовут Tom
del person1     # удаление из памяти
person2 = Person("Sam")
person2.display_info()  # Привет, меня зовут Sam
```

Консольный вывод:

```
Привет, меня зовут Tom
Tom удален из памяти
Привет, меня зовут Sam
Sam удален из памяти
```

Определение классов в модулях и подключение

Как правило, классы размещаются в отдельных модулях и затем уже импортируются в основной скрипт программы. Пусть у нас будет в проекте два файла: файл ***main.py*** (основной скрипт программы) и ***classes.py*** (скрипт с определением классов).

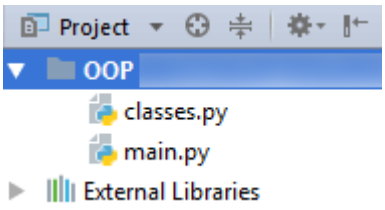

Рис. 1. Модули

В файле ***classes.py*** определим два класса:

```
class Person:
    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def display_info(self):
        print("Привет, меня зовут", self.name)


class Auto:
    def __init__(self, name):
        self.name = name

    def move(self, speed):
        print(self.name, "едет со скоростью", speed, "км/ч")
```

В дополнение к классу ***Person*** здесь также определен класс ***Auto***, который представляет машину и который имеет метод ***move*** и атрибут ***name***. Подключим эти классы и используем их в скрипте ***main.py***:

```
from classes import Person, Auto

tom = Person("Tom")
tom.display_info()

bmw = Auto("BMW")
bmw.move(120)
```

Подключение классов происходит точно также, как и функций из модуля. Мы можем подключить весь модуль выражением:

```
import classes
```

Либо подключить отдельные классы, как в примере выше.

В итоге мы получим следующий консольный вывод:

```
Привет, меня зовут Tom
BMW едет со скоростью 120 км/ч
```

Инкапсуляция

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

Например:

```
class Person:
    def __init__(self, name):
        self.name = name    # устанавливаем имя
        self.age = 1        # устанавливаем возраст

    def display_info(self):
        print("Имя:", self.name, "\tВозраст:", self.age)


tom = Person("Tom")
tom.name = "Человек-паук"       # изменяем атрибут name
tom.age = -129                  # изменяем атрибут age
tom.display_info()              # Имя: Человек-паук     Возраст: -129
```

Но в данном случае мы можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта.

С данной проблемой тесно связано понятие инкапсуляции. Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода.

Касательно инкапсуляции непосредственно в языке программирования Python, скрыть атрибуты класса можно, сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются свойствами.

Изменим выше определенный класс, определив в нем свойства:

```
class Person:
    def __init__(self, name):
        self.__name = name      # устанавливаем имя
        self.__age = 1          # устанавливаем возраст

    def set_age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age
```

```
        def get_name(self):
            return self.__name

        def display_info(self):
            print("Имя:", self.__name, "\tВозраст:", self.__age)

    tom = Person("Tom")

    tom.__age = 43              # Атрибут age не изменится
    tom.display_info()          # Имя: Tom  Возраст: 1
    tom.set_age(-3486)          # Недопустимый возраст
    tom.set_age(25)
    tom.display_info()          # Имя: Tom  Возраст: 25
```

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: ***self.__name***. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст:

```
    tom.__age = 43
```

А попытка получить его значение приведет к ошибке выполнения:

```
    print(tom.__age)
```

Однако все же нам может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

```
    def get_age(self):
        return self.__age
```

Данный метод еще часто называют геттер или аксессор.

Для изменения возраста определено другое свойство:

```
    def set_age(self, value):
        if value in range(1, 100):
            self.__age = value
        else:
            print("Недопустимый возраст")
```

Здесь мы уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод еще называют сеттер или мутатор (***mutator***).

Необязательно создавать для каждого приватного атрибута подобную пару свойств. Так, в примере выше имя человека мы можем установить только из конструктора. А для получение определен метод ***get_name***.

Аннотации свойств

Выше мы рассмотрели, как создавать свойства. Но Python имеет также еще один – более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом **@**.

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация ***имя_свойства_сеттера.setter***.

Перепишем класс Person с использованием аннотаций:

```
class Person:
    def __init__(self, name):
        self.__name = name  # устанавливаем имя
        self.__age = 1      # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)


tom = Person("Tom")

tom.display_info()      # Имя: Tom  Возраст: 1
tom.age = -3486         # Недопустимый возраст
print(tom.age)          # 1
tom.age = 36
tom.display_info()      # Имя: Tom  Возраст: 36
```

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково – ***age***. И поскольку геттер называется ***age***, то над сеттером устанавливается аннотация ***@age.setter***.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение ***tom.age***.

Наследование

Наследование позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного дизайна.

Ключевыми понятиями наследования являются подкласс и суперкласс. Подкласс наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (***base class***) или родительским (***parent class***), а подкласс – производным (***derived class***) или дочерним (***child class***).

Синтаксис для наследования классов выглядит следующим образом:

```
class подкласс (суперкласс):
    методы_подкласса
```

Например, в прошлых разделах был создан класс ***Person***, который представляет человека. Предположим, нам необходим класс работника, который работает на некотором предприятии. Мы могли бы создать с нуля новый класс, к примеру, класс ***Employee***. Однако он может иметь те же атрибуты и методы, что и класс ***Person***, так как сотрудник – это человек. Поэтому нет смысла определять в классе тот же функционал, что и в классе ***Person***. И в этом случае лучше применить наследование.

Итак, унаследуем класс ***Employee*** от класса ***Person***:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
```

```
        def age(self, age):
            if age in range(1, 100):
                self.__age = age
            else:
                print("Недопустимый возраст")

        @property
        def name(self):
            return self.__name

        def display_info(self):
            print("Имя:", self.__name, "\tВозраст:", self.__age)


    class Employee(Person):

        def details(self, company):
            # print(self.__name, "работает в компании", company) # так
нельзя, self.__name - приватный атрибут
            print(self.name, "работает в компании", company)


    tom = Employee("Tom", 23)
    tom.details("Google")
    tom.age = 33
    tom.display_info()
```

Класс ***Employee*** полностью перенимает функционал класса ***Person*** и в дополнении к нему добавляет метод ***details()***.

Стоит обратить внимание, что для ***Employee*** доступны через ключевое слово ***self*** все методы и атрибуты класса ***Person***, кроме закрытых атрибутов типа ***__name*** или ***__age***.

При создании объекта ***Employee*** мы фактически используем конструктор класса ***Person***. И кроме того, у этого объекта мы можем вызвать все методы класса ***Person***.

Полиморфизм

Полиморфизм является еще одним базовым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса.

Например, пусть у нас будет следующая иерархия классов:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)


class Employee(Person):
    # определение конструктора
    def __init__(self, name, age, company):
        Person.__init__(self, name, age)
        self.company = company

    # переопределение метода display_info
    def display_info(self):
        Person.display_info(self)
        print("Компания:", self.company)


class Student(Person):
    # определение конструктора
    def __init__(self, name, age, university):
        Person.__init__(self, name, age)
        self.university = university

    # переопределение метода display_info
    def display_info(self):
        print("Студент", self.name, "учится в университете",
self.university)

people = [Person("Tom", 23), Student("Bob", 19, "Harvard"),
Employee("Sam", 35, "Google")]
```

```
for person in people:
    person.display_info()
    print()
```

В производном классе ***Employee***, который представляет служащего, определяется свой конструктор, так как нам надо устанавливать при создании объекта еще и компанию, где работает сотрудник. Для этого конструктор принимает четыре параметра: стандартный параметр ***self***, параметры ***name***, ***age*** и параметр ***company***.

В самом конструкторе ***Employee*** вызывается конструктор базового класса ***Person***. Обращение к методам базового класса имеет следующий синтаксис:

```
суперкласс.название_метода(self [, параметры])
```

Поэтому в конструктор базового класса передаются имя и возраст. Сам же класс ***Employee*** добавляет к функционалу класса ***Person*** еще один атрибут – ***self.company***.

Кроме того, класс ***Employee*** переопределяет метод ***display_info()*** класса ***Person***, поскольку кроме имени и возраста необходимо выводить еще и компанию, в которой работает служащий. И чтобы повторно не писать код вывода имени и возраста, здесь также происходит обращение к методу базового класса – методу ***get_info***: ***Person.display_info(self)***.

Похожим образом определен класс ***Student***, представляющий студента. Он также переопределяет конструктор и метод ***display_info*** за тем исключением, что в методе ***display_info*** не вызывается версия этого метода из базового класса.

В основной части программы создается список из трех объектов ***Person***, в котором два объекта также представляют классы ***Employee*** и ***Student***. И в цикле этот список перебирается, и для каждого объекта в списке вызывается метод ***display_info***. На этапе выполнения программы Python учитывает иерархию наследования и выбирает нужную версию метода ***display_info()*** для каждого объекта. В итоге мы получим следующий консольный вывод:

```
Имя: Tom    Возраст: 23

Студент Bob учится в университете Harvard

Имя: Sam    Возраст: 35
Компания: Google
```

Проверка типа объекта

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции ***isinstance()*** мы можем проверить тип объекта. Эта функция принимает два параметра:

```
isinstance(object, type)
```

Первый параметр представляет объект, а второй – тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает ***True***. Например, возьмем вышеописанную иерархию классов:

```
for person in people:
    if isinstance(person, Student):
        print(person.university)
    elif isinstance(person, Employee):
        print(person.company)
    else:
        print(person.name)
    print()
```

Класс object. Строковое представление объекта

Начиная с 3-й версии Python все классы неявно имеют один общий суперкласс – ***object*** и все классы по умолчанию наследуют его методы.

Одним из наиболее используемых методов класса ***object*** является метод ***__str__()***. Когда необходимо получить строковое представление объекта или вывести объект в виде строки, то Python как раз вызывает этот метод. И при определении класса хорошей практикой считается переопределение этого метода.

К примеру, возьмем класс ***Person*** и выведем его строковое представление:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age

    @age.setter
```

```
        def age(self, age):
            if age in range(1, 100):
                self.__age = age
            else:
                print("Недопустимый возраст")

        def display_info(self):
            print("Имя:", self.__name, "\tВозраст:", self.__age)

    tom = Person("Tom", 23)
    print(tom)
```

При запуске программа выведет что-то наподобие следующего:

```
    <__main__.Person object at 0x0000017D2BEBDCF8>
```

Это не очень информативная информация об объекте. Теперь определим в классе ***Person*** метод ***__str__***:

```
    class Person:
        def __init__(self, name, age):
            self.__name = name  # устанавливаем имя
            self.__age = age  # устанавливаем возраст

        @property
        def name(self):
            return self.__name

        @property
        def age(self):
            return self.__age

        @age.setter
        def age(self, age):
            if age in range(1, 100):
                self.__age = age
            else:
                print("Недопустимый возраст")

        def display_info(self):
            print(self.__str__())

        def __str__(self):
            return "Имя: {} \t Возраст: {}".format(self.__name, self.__age)

    tom = Person("Tom", 23)
    print(tom)
```

Метод ***__str__()*** должен возвращать строку. И в данном случае мы возвращаем базовую информацию о человеке. И теперь консольный вывод будет другим:

```
Имя: Tom    Возраст: 23
```

Задания

Общее задание

1. Изучите теорию и ответьте на вопросы по теме лабораторной работы.
2. Необходимо проверять корректность вводимых данных и выводить соответствующие сообщения об ошибках.
3. Создать класс с полями, указанными в индивидуальном задании.
4. Реализовать в классе методы:
    a. конструктор по умолчанию;
    b. деструктор для освобождения памяти (с сообщением об уничтожении объекта);
    c. функции обработки данных, указанные в индивидуальном задании;
    d. функцию формирования строки информации об объекте.
5. Создать проект для демонстрации работы: сформировать объекты со значениями-константами и с введенными с клавиатуры значениями полей объекта. В основной ветке программы создайте три объекта класса. Вывести результаты работы на экран.
6. На основании предложенной предметной области спроектировать 3-4 класса, используя механизм наследования. Для каждого класса использовать отдельный модуль.
7. Предусмотреть у класса наличие полей, методов и свойств. Названия членов класса должны быть осмысленны и снабжены комментариями
8. Один из наследников должен перегружать метод родителя.
9. Один из классов должен содержать виртуальный метод, который переопределяется в одном наследнике и не переопределяется в другом.
10. Продемонстрировать работу всех объявленных методов.
11. Продемонстрировать вызов конструктора родительского класса при наследовании.

№ Варианта = (**№ПК** + 1) % 19 + 1

Дополнительное задание

Дополнительное задание выдается преподавателем на лабораторном занятии №4

Варианты

№	Класс-родитель и его поля	Функция-метод 1 обработки данных	Функция-метод 2 обработки данных
1	**Дата** (три числа): день, месяц, год	Определить, является ли год високосным (кратным 4)	Увеличить дату на 5 дней
2	**Работник**: фамилия, оклад, год поступления на работу	Вычислить стаж работы работника на данном предприятии	Сколько дней прошло после года поступления на работу
3	**Книга**: название, количество страниц, цена	Вычислить среднюю стоимость одной страницы	Увеличить цену книги в два раза, если название начинается со слова «Программирование»
4	**Время** (три числа): часы, минуты, секунды	Вычислить количество полных минут в указанном времени	Уменьшить время на 10 минут
5	**Товар**: наименование, цена, год выпуска	Определить, сколько лет назад был выпущен товар	Увеличить цену товара на 20%, если в наименовании товара есть слово «TV»
6	**Дата** (три числа): день, месяц, год	Увеличить год на 1	Уменьшить дату на 2 дня
7	**Работник**: фамилия, оклад, дата рождения	Вычислить возраст работника	Сколько календарных дней до исполнения работнику 50 лет

№	Класс-родитель и его поля	Функция-метод 1 обработки данных	Функция-метод 2 обработки данных
8	**Книга**: название, количество страниц, цена	Вычислить, сколько лет книге	Количество дней, прошедших после года издания книги
9	**Время** (три числа): часы, минуты, секунды	Вычислить количество секунд в указанном времени	Увеличить время на 5 секунд
10	**Работник**: фамилия, должность, оклад	Увеличить оклад на 15% (каждому работнику)	Работникам, у которых фамилия начинается с сочетания букв «Иван», присвоить должность «инженер».
11	**Книга**: название, количество страниц, цена	Увеличить количество страниц на 10	Уменьшить цену в два раза, если количество страниц больше 100 (после увеличения)
12	**Дата** (три числа): день, месяц, год	Определить, совпадают ли номер месяца и число дня	Увеличить дату на один месяц
13	**Товар**: наименование, цена в рублях, изготовитель	Пересчитать цену товара в евро	Увеличить цену товара в евро, если название товара содержит слово «Samsung».
14	**Время** (три числа): часы, минуты, секунды	Определить количество минут до полуночи (24:00:00)	Увеличить время 100 минут
15	**Правильная дробь**: числитель, знаменатель	Выразить значение дроби в процентах	Найти сумму цифр значения знаменателя
16	**Комната**: длина, ширина, высота (в метрах)	Площадь стен (вместе с окнами и дверьми)	Площадь стен без окна (размер 2×15 м) и двери (размер 2 ×8 м).

№	Класс-родитель и его поля	Функция-метод 1 обработки данных	Функция-метод 2 обработки данных
17	**Комплексное число**: действительная (a1) и мнимая (b1) части числа	Вычислить модуль комплексного числа	Вычислить аргумент комплексного числа в градусах
18	**Координаты изображения прямоугольника**: x1, y1, x2, y2	Вычислить площадь прямоугольника в пикселях	Вычислить длину диагонали прямоугольника в пикселях
19	**Параллелепипед**: длины сторон	Вычислить площадь поверхности	Вычислить объем параллелепипеда