Лабораторная работа №5 Классы в Питоне

1. Цель и порядок работы

Цель работы – изучить возможности работы с классами в Питоне.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по языку программирования Питон, выполняя указанные в тексте примеры.

Выполнить индивидуальное задание.

2. Краткая теория

2.1 Классы и объекты

Класс является шаблоном или формальным описанием объекта, а объект представляет экземпляр этого класса, его реальное воплощение. С точки зрения кода класс объединяет набор функций и переменных, которые выполняют определенную задачу. Функции класса 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])
```

Пример, определим простейший класс *Person*:

```
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 называется *self*. Через эту ссылку внутри класса 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.

Конструкторы

Для создания объекта класса используется конструктор. В предыдущем примере в классе *Person*, использовался конструктор по умолчанию, который неявно имеют все классы:

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

Конструктор в классах можно определить с помощью специального метода, который называется *__init()*. Пример, 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 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*. 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*.

Деструктор

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

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

```
del person1     # удаление из памяти
```

Удалять объект необязательно, так как после окончания работы скрипта все объекты автоматически удаляются из памяти.

Для определения деструктора необходимо реализовать встроенную функцию *__del__*, которая 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
```

<u>Определение классов в модулях и подключение</u>

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

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")
```

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(65)
```

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.

2.2 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, age):
        self.name = name    # устанавливаем имя
        self.age = age      # устанавливаем возраст

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


tom = Person("Tom", 23)
tom.name = "Человек-паук"       # изменяем атрибут name
tom.age = -129                  # изменяем атрибут age
tom.display_info()              # Имя: Человек-паук     Возраст: -129
```

В данном случае 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, age):
        self.__name = name    # устанавливаем имя
        self.__age = age      # 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", 23)

tom.__age = 43          # Атрибут age не изменится
tom.display_info()      # Имя: Tom  Возраст: 23
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 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 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 уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод называют сеттер или мьютейтор (*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*.

<u>Аннотации свойств</u>

Python имеет также еще один - 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_геттера.setter*.

Перепишем класс Person с использованием аннотаций:

```
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)

tom = Person("Tom", 23)
tom.display_info()     # Имя: Tom  Возраст: 23
tom.age = -3486        # Недопустимый возраст
print(tom.age)         # 23
tom.age = 36
tom.display_info()     # Имя: Tom  Возраст: 36
```

Свойство-сеттер определяется после свойства-геттера. 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, можно обратится через выражение *tom.age*.

2.3 Наследование

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

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
```

Расмотрим пример наследования на основе классов *Person*, который представляет человека, и класс *Employee* - класс работника.

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

2.4 Полиморфизм

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

Например:

```
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* и *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 вместо 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.

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 выше описанную 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()
```

2.5 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:

3 Контрольные вопросы

Понятие класса в языке Питон.

Поля и методы класса.

Конструкторы и деструкторы в Питоне.

Свойства и аннотации.

Что понимается под термином «наследование»? Приведите примеры на язык Питон.

Что понимается под термином «полиморфизм»? Приведите примеры на язык Питон.

Что понимается под термином «инкапсуляция»? Приведите примеры на язык Питон.

4 Задание

Выбрать задание согласно варианта.

Порядок выполнения работы:

- разработать поля, методы и свойства для каждого из определяемых классов;
- все поля классов должны быть приватными;
- реализовать для каждого класса конструктор и деструктор;

- свойства по изменению и отображению значения полей;
- метод поиска информации из списка данных объектов по определенным критериям;
- переопределенный метод *__str__()* для вывода информации об объекте;
- реализовать меню по работе со списком объектов которое должно включать: добавление, редактирование, удаление объекта; отображение данных об объекте; поиска информации по определенным критериям.

Реализовать программу на *Python* в соответствии с вариантом исполнения. Описание классов должно быть в отдельном модуле. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Построить иерархию классов в соответствии с вариантом задания:

1. Студент, преподаватель, персона, заведующий кафедрой.
2. Служащий, персона, рабочий, инженер.
3. Рабочий, кадры, инженер, администрация.
4. Деталь, механизм, изделие, узел.
5. Организация, страховая компания, нефтегазовая компания, завод.
6. Журнал, книга, печатное издание, учебник.
7. Тест, экзамен, выпускной экзамен, испытание.
8. Место, область, город, мегаполис.
9. Игрушка, продукт, товар, молочный продукт.
10. Квитанция, накладная, документ, счет.
11. Автомобиль, поезд, транспортное средство, экспресс.
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
13. Республика, монархия, королевство, государство.
14. Млекопитающее, парнокопытное, птица, животное.
15. Корабль, пароход, парусник, корвет.
16. Самолет, автомобиль, корабль, транспортное средство.
17. Точка, линия, фигура плоская, фигура объемная.
18. Картина, рисунок, репродукция, пейзаж.
19. Статья, раздел, журнал, издательство.
20. Квартира, дом, улица, населенный пункт.