**ЛАБОРАТОРНАЯ РАБОТА № 8**

**Объектно-ориентированное программирование в Python**

**Цель работы:**

1. Освоить принципы ООП в Python;
2. Получить практические навыки проектирования и реализации классов.

**Краткие теоретические сведения:**

**Основные понятия объектно-ориентированного программирования**

Объектно-ориентированное программирование (ООП) является методологией разработки программного обеспечения, в основе которой лежит понятие класса и объекта, при этом сама программа создается как некоторая совокупность объектов, которые взаимодействую друг с другом и с внешним миром. Каждый объект является экземпляром некоторого класса. Классы образуют иерархии. Выделяют три основных “столпа” ООП- это инкапсуляция, наследование и полиморфизм.

**Инкапсуляция**

Под инкапсуляцией понимается сокрытие деталей реализации, данных и т.п. от внешней стороны. Например, можно определить класс “холодильник”, который будет содержать следующие данные: производитель, объем, количество камер хранения, потребляемая мощность и т.п., и методы: открыть/закрыть холодильник, включить/выключить, но при этом реализация того, как происходит непосредственно включение и выключение пользователю вашего класса не доступна, что позволяет ее менять без опасения, что это может отразиться на использующей класс «холодильник» программе. При этом класс становится новым типом данных в рамках разрабатываемой программы. Можно создавать переменные этого нового типа, такие переменные называются объекты.

**Наследование**

Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.

Примером базового класса, демонстрирующего наследование, можно определить класс “автомобиль”, имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – “грузовой автомобиль”, он будет содержать те же атрибуты и методы, что и класс “автомобиль”, и дополнительные свойства: количество осей, мощность компрессора и т.п..

**Полиморфизм**

Полиморфизм позволяет одинаково обращаться с объектами, имеющими однотипный интерфейс, независимо от внутренней реализации объекта. Например, с объектом класса “грузовой автомобиль” можно производить те же операции, что и с объектом класса “автомобиль”, т.к. первый является наследником второго, при этом обратное утверждение неверно (во всяком случае не всегда). Другими словами полиморфизм предполагает разную реализацию методов с одинаковыми именами. Это очень полезно при наследовании, когда в классе наследнике можно переопределить методы класса родителя.

**Классы в *Python***

**Создание классов и объектов**

Создание класса в *Python* начинается с инструкции *class*. Вот так будет выглядеть минимальный класс.

**class** C:

**pass**

Класс состоит из объявления (инструкция *class*), имени класса (нашем случае это имя *C*) и тела класса, которое содержит атрибуты и методы (в нашем минимальном классе есть только одна инструкция *pass*).

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

***имя\_объекта = имя\_класса()***

**Статические и динамические атрибуты класса**

Как уже было сказано выше, класс может содержать атрибуты и методы. Атрибут может быть статическим и динамическим (уровня объекта класса). Суть в том, что для работы со статическим атрибутом, вам не нужно создавать экземпляр класса, а для работы с динамическим – нужно. Пример:

**class** Rectangle:

default\_color = "green"

**def** \_\_init\_\_(self, width, height):

self.width = width

self.height = height

В представленном выше классе, атрибут *default\_color* – это статический атрибут, и доступ к нему, как было сказано выше, можно получить не создавая объект класса *Rectangle*.

>>> Rectangle.default\_color

'green'

*width* и *height* – это динамические атрибуты, при их создании было использовано ключевое слово *self*. Пока просто примите это как должное, более подробно про *self* будет рассказано ниже. Для доступа к *width* и *height* предварительно нужно создать объект класса *Rectangle:*

>>> rect = Rectangle(10, 20)

>>> rect.width

10

>>> rect.height

20

Если обратиться через класс, то получим ошибку:

>>> Rectangle.width

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

AttributeError: type object 'Rectangle' has no attribute 'width'

При этом, если вы обратитесь к статическому атрибуту через экземпляр класса, то все будет ОК, до тех пор, пока вы не попытаетесь его поменять.

Проверим ещё раз значение атрибута *default\_color:*

>>> Rectangle.default\_color

'green'

Присвоим ему новое значение:

>>> Rectangle.default\_color = "red"

>>> Rectangle.default\_color

'red'

Создадим два объекта класса *Rectangle* и проверим, что *default\_color* у них совпадает:

>>> r1 = Rectangle(1,2)

>>> r2 = Rectangle(10, 20)

>>> r1.default\_color

'red'

>>> r2.default\_color

'red'

Если поменять значение *default\_color* через имя класса *Rectangle*, то все будет ожидаемо: у объектов *r1* и *r2* это значение изменится, но если поменять его через экземпляр класса, то у экземпляра будет создан атрибут с таким же именем как статический, а доступ к последнему будет потерян:

Меняем *default\_color* через *r1*:

>>> r1.default\_color = "blue"

>>> r1.default\_color

'blue'

При этом у *r2* остается значение статического атрибута:

>>> r2.default\_color

'red'

>>> Rectangle.default\_color

'red'

Вообще напрямую работать с атрибутами – не очень хорошая идея, лучше для этого использовать свойства.

**Методы класса**

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса и выполняющая определенную работу.

Методы бывают статическими, классовыми (среднее между статическими и обычными) и уровня класса (будем их называть просто словом метод). Статический метод создается с [декоратором](https://devpractice.ru/python-lesson-19-decorators/) *@staticmethod*, классовый – с декоратором *@classmethod*, первым аргументом в него передается *cls*, обычный метод создается без специального декоратора, ему первым аргументом передается *self:*

**class** MyClass:

@staticmethod

**def** ex\_static\_method():

print("static method")

@classmethod

**def** ex\_class\_method(cls):

print("class method")

**def** ex\_method(self):

print("method")

Статический и классовый метод можно вызвать, не создавая экземпляр класса, для вызова *ex\_method()* нужен объект:

>>> MyClass.ex\_static\_method()

static method

>>> MyClass.ex\_class\_method()

**class** method

>>> MyClass.ex\_method()

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

TypeError: ex\_method() missing 1 required positional argument: 'self'

>>> m = MyClass()

>>> m.ex\_method()

method

**Конструктор класса и инициализация экземпляра класса**

В *Python* разделяют конструктор класса и метод для инициализации экземпляра класса. Конструктор класса это метод *\_\_new\_\_(cls, \*args, \*\*kwargs)* для инициализации экземпляра класса используется метод *\_\_init\_\_(self)*. При этом, как вы могли заметить *\_\_new\_\_* – это классовый метод, а *\_\_init\_\_* таким не является. Метод *\_\_new\_\_* редко переопределяется, чаще используется реализация от базового класса *object* (см. раздел Наследование), *\_\_init\_\_* же наоборот является очень удобным способом задать параметры объекта при его создании.

Создадим реализацию класса *Rectangle* с измененным конструктором и инициализатором, через который задается ширина и высота прямоугольника:

**class** Rectangle:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

print("Hello from \_\_new\_\_")

**return** super().\_\_new\_\_(cls)

**def** \_\_init\_\_(self, width, height):

print("Hello from \_\_init\_\_")

self.width = width

self.height = height

>>> rect = Rectangle(10, 20)

Hello **from** \_\_new\_\_

Hello **from** \_\_init\_\_

>>> rect.width

10

>>> rect.height

20

**Что такое *self*?**

До этого момента вы уже успели познакомиться с ключевым словом *self*. *self* – это ссылка на текущий экземпляр класса, в таких языках как *Java*, *C#* аналогом является ключевое слово *this*. Через *self* вы получаете доступ к атрибутам и методам класса внутри него:

**class** Rectangle:

**def** \_\_init\_\_(self, width, height):

self.width = width

self.height = height

**def** area(self):

**return** self.width \* self.height

В приведенной реализации метод *area* получает доступ к атрибутам *width* и *height* для расчета площади. Если бы в качестве первого параметра не было указано *self*, то при попытке вызвать *area* программа была бы остановлена с ошибкой.

**Уровни доступа атрибута и метода**

Если вы знакомы с языками программирования *Java*, *C#*, *C+*+ то, наверное, уже задались вопросом: “а как управлять уровнем доступа?”. В перечисленных языка вы можете явно указать для переменной, что доступ к ней снаружи класса запрещен, это делается с помощью ключевых слов (*private*, *protected* и т.д.). В *Python* таких возможностей нет, и любой может обратиться к атрибутам и методам вашего класса, если возникнет такая необходимость. Это существенный недостаток этого языка, т.к. нарушается один из ключевых принципов ООП – инкапсуляция. Хорошим тоном считается, что для чтения/изменения какого-то атрибута должны использоваться специальные методы, которые называются *getter*/*setter*, их можно реализовать, но ничего не помешает изменить атрибут напрямую. При этом есть соглашение, что метод или атрибут, который начинается с нижнего подчеркивания, является скрытым, и снаружи класса трогать его не нужно (хотя сделать это можно).

Внесем соответствующие изменения в класс *Rectangle*:

**class** Rectangle:

**def** \_\_init\_\_(self, width, height):

self.\_width = width

self.\_height = height

**def** get\_width(self):

**return** self.\_width

**def** set\_width(self, w):

self.\_width = w

**def** get\_height(self):

**return** self.\_height

**def** set\_height(self, h):

self.\_height = h

**def** area(self):

**return** self.\_width \* self.\_height

В приведенном примере для доступа к *\_width* и *\_height* используются специальные методы, но ничего не мешает вам обратиться к ним (атрибутам) напрямую.

>>> rect = Rectangle(10, 20)

>>> rect.get\_width()

10

>>> rect.\_width

10

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую вы к нему уже не обратитесь (простым образом). Модифицируем наш класс *Rectangle*:

**class** Rectangle:

**def** \_\_init\_\_(self, width, height):

self.\_\_width = width

self.\_\_height = height

**def** get\_width(self):

**return** self.\_\_width

**def** set\_width(self, w):

self.\_\_width = w

**def** get\_height(self):

**return** self.\_\_height

**def** set\_height(self, h):

self.\_\_height = h

**def** area(self):

**return** self.\_\_width \* self.\_\_height

Попытка обратиться к *\_\_width* напрямую вызовет ошибку, нужно работать только через *get\_width()*:

>>> rect = Rectangle(10, 20)

>>> rect.\_\_width

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

AttributeError: 'Rectangle' object has no attribute '\_\_width'

>>> rect.get\_width()

10

Но на самом деле это сделать можно, просто этот атрибут теперь для внешнего использования носит название: *\_Rectangle\_\_width*:

>>> rect.\_Rectangle\_\_width

10

>>> rect.\_Rectangle\_\_width = 20

>>> rect.get\_width()

20

**Свойства**

Свойством называется такой метод класса, работа с которым подобна работе с атрибутом. Для объявления метода свойством необходимо использовать декоратор *@property*.

Важным преимуществом работы через свойства является то, что вы можете осуществлять проверку входных значений, перед тем как присвоить их атрибутам.

Сделаем реализацию класса *Rectangle* с использованием свойств:

**class** Rectangle:

**def** \_\_init\_\_(self, width, height):

self.\_\_width = width

self.\_\_height = height

@property

**def** width(self):

**return** self.\_\_width

@width.setter

**def** width(self, w):

**if** w > 0:

self.\_\_width = w

**else**:

**raise** ValueError

@property

**def** height(self):

**return** self.\_\_height

@height.setter

**def** height(self, h):

**if** h > 0:

self.\_\_height = h

**else**:

**raise** ValueError

**def** area(self):

**return** self.\_\_width \* self.\_\_height

Теперь работать с *width* и *height* можно так, как будто они являются атрибутами:

>>> rect = Rectangle(10, 20)

>>> rect.width

10

>>> rect.height

20

Можно не только читать, но и задавать новые значения свойствам:

>>> rect.width = 50

>>> rect.width

50

>>> rect.height = 70

>>> rect.height

70

Если вы обратили внимание: в *setter*’ах этих свойств осуществляется проверка входных значений, если значение меньше нуля, то будет выброшено исключение *ValueError*:

>>> rect.width = -10

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

File "test.py", line 28, **in** width

**raise** ValueError

ValueError

**Наследование**

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка может быть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в *Python* можно его использовать. По умолчанию все классы в *Python* являются наследниками от *object*, явно этот факт указывать не нужно.

Синтаксически создание класса с указанием его родителя выглядит так:

***class* имя\_класса(имя\_родителя1, [имя\_родителя2,…, имя\_родителя\_n])**

Переработаем наш пример так, чтобы в нем присутствовало наследование:

**class** Figure:

**def** \_\_init\_\_(self, color):

self.\_\_color = color

@property

**def** color(self):

**return** self.\_\_color

@color.setter

**def** color(self, c):

self.\_\_color = c

**class** Rectangle(Figure):

**def** \_\_init\_\_(self, width, height, color):

super().\_\_init\_\_(color)

self.\_\_width = width

self.\_\_height = height

@property

**def** width(self):

**return** self.\_\_width

@width.setter

**def** width(self, w):

**if** w > 0:

self.\_\_width = w

**else**:

**raise** ValueError

@property

**def** height(self):

**return** self.\_\_height

@height.setter

**def** height(self, h):

**if** h > 0:

self.\_\_height = h

**else**:

**raise** ValueError

**def** area(self):

**return** self.\_\_width \* self.\_\_height

Родительским классом является *Figure*, который при инициализации принимает цвет фигуры и предоставляет его через свойства. *Rectangle* – класс наследник от *Figure*. Обратите внимание на его метод *\_\_init\_\_*: в нем первым делом вызывается конструктор (хотя это не совсем верно, но будем говорить так) его родительского класса:

*super().\_\_init\_\_(color)*

*super* – это ключевое слово, которое используется для обращения к родительскому классу.

Теперь у объекта класса *Rectangle* помимо уже знакомых свойств *width* и *height* появилось свойство *color:*

>>> rect = Rectangle(10, 20, "green")

>>> rect.width

10

>>> rect.height

20

>>> rect.color

'green'

>>> rect.color = "red"

>>> rect.color

'red'

**Полиморфизм**

Как уже было сказано во введении в рамках ООП полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод *info()*, который печатает сводную информацию по объекту класса *Figure* и переопределим этот метод в классе *Rectangle*, добавим  в него дополнительные данные:

**class** Figure:

**def** \_\_init\_\_(self, color):

self.\_\_color = color

@property

**def** color(self):

**return** self.\_\_color

@color.setter

**def** color(self, c):

self.\_\_color = c

**def** info(self):

print("Figure")

print("Color: " + self.\_\_color)

**class** Rectangle(Figure):

**def** \_\_init\_\_(self, width, height, color):

super().\_\_init\_\_(color)

self.\_\_width = width

self.\_\_height = height

@property

**def** width(self):

**return** self.\_\_width

@width.setter

**def** width(self, w):

**if** w > 0:

self.\_\_width = w

**else**:

**raise** ValueError

@property

**def** height(self):

**return** self.\_\_height

@height.setter

**def** height(self, h):

**if** h > 0:

self.\_\_height = h

**else**:

**raise** ValueError

**def** info(self):

print("Rectangle")

print("Color: " + self.color)

print("Width: " + str(self.width))

print("Height: " + str(self.height))

print("Area: " + str(self.area()))

**def** area(self):

**return** self.\_\_width \* self.\_\_height

Посмотрим, как это работает

>>> fig = Figure("orange")

>>> fig.info()

Figure

Color: orange

>>> rect = Rectangle(10, 20, "green")

>>> rect.info()

Rectangle

Color: green

Width: 10

Height: 20

Area: 200

Таким образом, класс наследник может расширять функционал класса родителя.

**Порядок выполнения лабораторной работы:**

1. Создать UML диаграмму классов с их полями и методами;
2. Создать набор классов для хранения информации в соответствии с выбранной темой на основе созданной диаграммы по следующим правилам:
   1. Классы должны содержать как минимум два атрибута и один метод;
   2. Хотя бы один класс должен содержать конструктор заполнения полей с клавиатуры;
   3. В программе должен присутствовать механизм взаимодействия экземпляров классов.

**Задание на лабораторную работу:**

Реализовать программный продукт симуляции взаимодействия сущностей выбранной индивидуально в прошлом задании темы.

*Например*, в теме «Квест» могут присутствовать следующие сущности:

Главный герой (раса, имя, уровень жизни, интеллект, наносимый урон),

Противник (раса, уровень жизни, интеллект, наносимый урон),

Квест (название, сложность, награды).

На рис. 1 представлен пример класса Главный герой:



*Рисунок 1. Класс Главный герой*

**Контрольные вопросы:**

1. Какие особенности вы можете выделить в ООП Python в сравнении с Java?
2. Какие методы позволяют управлять атрибутами класса?
3. Какими могут быть атрибуты?
4. Как создать пустой класс?