Лабораторная работа №3

Объектно-ориентированное программирование Python

Цель: настоящей работы состоит в том, чтобы изучить объектно-ориентированное программирование с использованием Python

.

Задачи:

Выполнение практической работы предполагает решение ***следующих задач***:

1. Создание и использование модификаторов доступа
2. Свойства Полиморфизм; Наследование; Инкапсуляция.
3. Подготовка отчета, содержащего минимальный объем информации по каждому этапу выполнения работы.

Последовательность выполнения работы :

Использован материал И.В. Мироновой, Программирование на языке Python. Часть 2: Учебное пособие - М.: Финансовый университет, департамент анализа данных, принятия решений и финансовых технологий, 2019. -51 с.

Модификаторы доступа

Модификаторы доступа в Python используются для модификации области видимости переменных по умолчанию. Есть три типа модификаторов доступов в Python ООП:

1. публичный — **public**;
2. приватный — **private**;
3. защищенный — **protected**.

Доступ к переменным с модификаторами публичного доступа открыт из любой точки вне класса, доступ к приватным переменным открыт **только внутри класса**, и в случае с защищенными переменными, доступ открыт только внутри того же пакета.

Для создания **приватной переменной**, вам нужно проставить префикс двойного подчеркивание __ с названием переменной.

Для создания защищенной переменной, вам нужно проставить префикс из одного нижнего подчеркивания _ с названием переменной. Для публичных переменных, вам не нужно проставлять префиксы вообще.

Давайте взглянем на **публичные**, **приватные** и **защищенные** переменные в действии. Выполните следующий скрипт:

```
1 class Car:
2     def __init__(self):
3         print ("Двигатель заведен")
4         self.name = "corolla"
5         self.__make = "toyota"
6         self._model = 1999
```

Здесь мы создали простой класс `Car` с конструктором и тремя переменными: `name`, `make`, и `model` (название, марка и модель).

Переменная `name` является **публичной**, в то время как переменные `make` и `model` являются **приватными** и **защищенными**, соответственно.

Давайте создадим объект класса `Car` и попытаемся получить доступ к переменной `name`. Выполним следующий скрипт:

```
1 car_a = Car()
2 print(car_a.name)
```

Так как `name` является публичной переменной, мы можем получить к ней доступ не из класса. В выдаче вы увидите значение переменной `name`, выведенное в консоли.

Теперь попробуем вывести значение переменной `make`. Выполняем следующий скрипт:

```
1 print(car_a.make)
```

В выдаче мы получим следующее уведомление об ошибке:

```
1 AttributeError: 'Car' object has no attribute 'make'
```

Мы рассмотрели большую часть основных **концепций объектно-ориентированного программирования** в предыдущих двух секциях. Теперь, поговорим о столбах объектно-ориентированного программирования:

- Полиморфизм;
- Наследование;
- Инкапсуляция.

Наследование и полиморфизм

Наследование в объектно-ориентированном программировании очень похоже на наследование в реальной жизни, где ребенок наследует те или иные характеристики его родителей в дополнение к его собственным характеристикам.

В объектно-ориентированном программировании, наследование означает отношение `IS-A`. Например, болид — это транспорт. Наследование это одна

из самых удивительных концепций объектно-ориентированного программирования, так как оно подразумевает **повторное использование**. Основная идея наследования в объектно-ориентированном программировании заключается в том, что **класс может наследовать характеристики другого класса**. Класс, который наследует другой класс, называется **дочерним классом** или производным классом, и класс, который дает наследие, называется **родительским**, или основным.

Рассмотрим на очень простой пример наследования. Выполним следующий скрипт:

```
1 # Создание класса Vehicle
2 class Vehicle:
3     def vehicle_method(self):
4         print("Это родительский метод из класса Vehicle")
5
6 # Создание класса Car, который наследует Vehicle
7 class Car(Vehicle):
8     def car_method(self):
9         print("Это метод из дочернего класса")
```

В скрипте выше мы создаем два класса: `Vehicle` и `Car`, который наследует класс `Vehicle`. Чтобы наследовать класс, вам нужно только вписать название родительского класса внутри скобок, которая следует за названием дочернего класса. Класс `Vehicle` содержит метод `vehicle_method()`, а дочерний класс содержит метод `car_method()`. Однако, так как класс `Car` наследует класс `Vehicle`, он также наследует и метод `vehicle_method()`.

Рассмотрим это на практике и выполним следующий скрипт:

```
1 car_a = Car()
2 car_a.vehicle_method() # Вызываем метод родительского класса
```

В этом скрипте мы создали объект класса `Car` вызывали метод `vehicle_method()` при помощи объекта класса `Car`. Вы можете обратить внимание на то, что класс `Car` не содержит ни одного метода `vehicle_method()`, но так как он унаследовал класс `Vehicle`, который содержит `vehicle_method()`, класс `Car` также будет использовать его. Выдача выглядит следующим образом:

```
1 Это родительский метод из класса Vehicle
```

Множественное наследование Python

В Python, **родительский класс** может иметь несколько **дочерних**, и, аналогично, **дочерний класс** может иметь несколько **родительских классов**. Давайте рассмотрим первый сценарий. Выполним следующий скрипт:

```
1  # создаем класс Vehicle
2  class Vehicle:
3      def vehicle_method(self):
4          print("Это родительский метод из класса Vehicle")
5
6  # создаем класс Car, который наследует Vehicle
7  class Car(Vehicle):
8      def car_method(self):
9          print("Это дочерний метод из класса Car")
10
11 # создаем класс Cycle, который наследует Vehicle
12 class Cycle(Vehicle):
13     def cycleMethod(self):
14         print("Это дочерний метод из класса Cycle")
```

В этом скрипте, родительский класс `Vehicle` наследуется двумя дочерними классами — `Car` и `Cycle`. Оба дочерних класса будут иметь доступ

к `vehicle_method()` родительского класса. Запустите следующий скрипт, чтобы увидеть это лично:

```
car_a = Car()
car_a.vehicle_method() # вызов метода родительского класса
car_b = Cycle()
car_b.vehicle_method() # вызов метода родительского класса
```

В выдаче вы увидите выдачу метода `vehicle_method()` дважды, как показано ниже:

```
Это родительский метод из класса Vehicle
Это родительский метод из класса Vehicle
```

Вы можете видеть, как **родительский класс** наследуется двумя **дочерними классами**. Таким же образом, дочерний класс может иметь несколько родительских. Посмотрим на пример:

```
class Camera:
    def camera_method(self):
        print("Это родительский метод из класса Camera")

class Radio:
    def radio_method(self):
        print("Это родительский метод из класса Radio")

class CellPhone(Camera, Radio):
    def cell_phone_method(self):
        print("Это дочерний метод из класса CellPhone")
```

В скрипте выше мы создали три класса: `Camera`, `Radio`, и `CellPhone`. Классы `Camera` и `Radio` наследуются классом `CellPhone`. Это значит, что

класс `CellPhone` будет иметь доступ к методам классов `Camera` и `Radio`. Следующий скрипт подтверждает это:

```
1 cell_phone_a = CellPhone()
2 cell_phone_a.camera_method()
3 cell_phone_a.radio_method()
```

Выдача будет выглядеть следующим образом:

```
1 Это родительский метод из класса Camera
2 Это родительский метод из класса Radio
```

Полиморфизм

Термин полиморфизм буквально означает наличие нескольких форм. В контексте объектно-ориентированного программирования, полиморфизм означает способность объекта вести себя по-разному.

Полиморфизм в программировании реализуется через перегрузку метода, либо через его переопределение.

Перегрузка метода

Перегрузка метода относится к свойству метода вести себя по-разному, в зависимости от количества или типа параметров. Взглянем на очень простой пример **перегрузки метода**. Выполним следующий скрипт:

Python

```
# создаем класс Car
class Car:
1   def start(self, a, b=None):
2       if b is not None:
3           print (a + b)
4       else:
5           print (a)
```

```
6
7
```

В скрипте выше, если метод `start()` вызывается передачей одного аргумента, параметр будет выведен на экран. Однако, если мы передадим 2 аргумента методу `start()`, он внесет оба аргумента и выведет результат суммы.

Попробуем с одним аргументом для начала:

```
1 car_a = Car()
2 car_a.start(10)
```

В выдаче мы можем видеть 10. Теперь попробуем передать два аргумента:

```
1 car_a.start(10, 20)
```

В выдаче вы увидите 30.

Переопределение метода

Переопределение метода относится к наличию метода с одинаковым названием в дочернем и родительском классах. **Определение метода** отличается в родительском и дочернем классах, но название остается тем же. Давайте посмотрим на простой пример **переопределения метода в Python**.

```
1  # создание класса Vehicle
2  class Vehicle:
3      def print_details(self):
4          print("Это родительский метод из класса Vehicle")
5
6  # создание класса, который наследует Vehicle
```

```
class Car(Vehicle):
    def print_details(self):
        print("Это дочерний метод из класса Car")

# создание класса Cycle, который наследует Vehicle
class Cycle(Vehicle):
    def print_details(self):
        print("Это дочерний метод из класса Cycle")
```

В скрипте выше, классы `Cycle` и `Car` наследуют класс `Vehicle`. Класс `Vehicle` содержит метод `print_details()`, который переопределен дочерним классом. Теперь, если вы вызовите метод `print_details()`, выдача будет зависеть от объекта, через который вызывается метод. Выполните следующий скрипт, чтобы понять суть на деле:

```
car_a = Vehicle()
car_a. print_details()

car_b = Car()
car_b.print_details()

car_c = Cycle()
car_c.print_details()
```

Выдача будет выглядеть вот так:

```
Это родительский метод из класса Vehicle
Это дочерний метод из класса Car
Это дочерний метод из класса Cycle
```

Как вы видите, выдача отличается, к тому же метод `print_details()` вызывается через производные классы одного и того

же базового класса. Однако, так как дочерние классы переопределены методом родительского класса, методы ведут себя по-разному.

Возможность кода работать с разными типами данных называют **полиморфизмом**. Рассмотрим следующий пример:

```
from math import pi

class Circle:
    def __init__(self, radius):
        self.r = radius

    def perim(self):
        return 2 * pi * self.r


class Rect:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def perim(self):
        return 2 * (self.a + self.b)

def print_info(x):
    print(f"perimeter = {x.perim():.5f}")
```

В данном случае у нас имеются 2 разных класса с одинаковым интерфейсом (метод perim) и полиморфная функция print_info, которая печатает информацию о периметре фигуры. Мы можем использовать эту функцию, например, так:

```
c1 = Circle(1)
print_info(c1)  #  perimeter = 6.28319
r1 = Rect(2, 5)
print_info(r1)  # perimeter = 14.00000
```

Как видим, одна и та же функция работает и для окружности, и для прямоугольника. Причем для каждого объекта вызывается метод из его класса. Это и есть полиморфизм. Какой метод вызывать в каждом

конкретном случае Pyton определяет с помощью специального атрибута __class__, который у каждого объекты содержит ссылку на его класс.

Чтобы полиморфизм работал, в Python достаточно, чтобы методы имели одинаковые имена, одинаковое количество параметров, одинаково возвращали (или не возвращали) значения. Кроме того, в разных классах методы с одинаковыми именами должны выполнять одинаковые по смыслу действия, хотя и по-разному.

Еще одна базовая концепция объектно-ориентированного программирования (наряду с инкапсуляцией и полиморфизмом) это – наследование. Наследование позволяет создавать новые классы из существующих. При этом реализуется отношение вида «класс B является частным случаем класса A». Класс A называется базовым классом (или суперклассом), а B – производным классом (или подклассом). В программе это записывается следующим образом:

```
class B(A):
        …
```

Созданный класс можно использовать в качестве базового для создания следующего класса. Используя один и тот же базовый класс можно создать несколько разных производных классов. Таким образом механизм наследования позволяет создавать целые иерархии классов.

Основное преимущество наследования состоит в том, что в новом (производном) классе будут доступны атрибуты и методы базового класса. Их не нужно создавать повторно.

При обращении к методу (или атрибуту) экземпляра производного класса метод сначала ищется в исходном (производном) классе. Если метода там нет, он ищется в базовом классе. Эти шаги повторяются до тех пор, пока метод не будет найден, или пока не дойдем до класса, который ни от кого не наследуется.

В производном классе можно создать новые методы и атрибуты, в том

числе может потребоваться создать метод с таким же именем, как в базовом классе, например, конструктор (имя конструктора во всех классах одинаковое). Это называется переопределением метода. При вызове алгоритм поиска метода всегда один и тот же независимо от того, был он переопределен или нет: если метод найден в текущем классе, то он и выполняется, если не найден, то ищется в базовом классе.

Таким образом, если вы в новом классе не создали новый конструктор, то при создании экземпляра этого нового класса будет вызываться конструктор базового класса. Если же вы создали в новом классе свой конструктор, то он и вызовется. Но учтите, что в этом случае конструктор базового класса автоматически вызываться не будет, хотя в большинстве случаев нам нужно, чтобы он тоже выполнился.

Для того, чтобы в производном классе B вызвать одноименный метод базового класса A, существует несколько способов. Рассмотрим два из них (на примере конструктора):

A.__init__(self)

super().__init__()

В первом случае явно указывается имя класса, в котором нужно найти метод, и в первом параметре передается ссылка на экземпляр класса (self). Во втором – используется функция super(). В этом случае self передавать не нужно, а поиск метода будет выполняться во всех базовых классах.

В качестве примера рассмотрим 3 класса: базовый класс People и производные Teacher и Student. Методы базового класса __init__() и __str__() переопределяются в производных классах таким образом, что в них вызываются методы с таким же именем базового класса. В примере продемонстрированы оба способа вызова нужного метода базового класса.

Обратите внимание на полиморфный метод info(), реализованный в базовом классе. Он работает и в производных классах.

```
class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f'Имя: {self.name}, Возраст: {self.age}'

    def info(self):
        print(self.__class__.__name__ + ': ' + str(self))


class Teacher(People):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary

    def __str__(self):
        return f'{super().__str__()}, Зарплата: {self.salary}'



class Student(People):
    def __init__(self, name, age, marks):
        People.__init__(self, name, age)
        self.marks = marks

    def __str__(self):
        return f'{super().__str__()}, Оценки: {self.marks}'

t = Teacher('Иванова', 40, 30000)
s = Student('Петров', 20, [70, 75, 87])

members = [t, s]
for member in members:
    member.info()
```

Задания для самостоятельной работы

Специальные методы

В языке Python существует большое количество специальных методов, которые вы можете реализовать в своих классах. В литературе их также называют магическими методами. Имена таких методов начинаются

и заканчиваются двумя символами подчеркивания. Мы уже сталкивались с ними: init_(),_str_(). Имена специальных методов и их смысл определены создателями языка: создавать новые нельзя, можно только реализовывать существующие. Поэтому, если вы создадите свой метод, у которого имя имеет два символа подчеркивания в начале и в конце, он от этого не станет специальным, а вот если это имя совпадет случайно с именем специального метода, то вы можете получить непонятные побочныеэффекты и ошибки. Поэтому используйте такие имена только для реализации специальных методов.

Описание специальных методов следует искать в литературе и документации по языку. Обращать внимание нужно не только на название метода, но и на параметры и возвращаемое значение. Ваша реализация должна точно соответствовать этому описанию. Это очень важно, так какэти методы обычно вызываются неявно в отличие от обычных методов, поэтому их и называют магическими. Важно также четко понимать, в какой момент вызывается метод.

Типичное использование специальных методов – реализация различных операций для ваших объектов. Гораздо удобнее и естественнее написать a + b или a < b, чем использовать выражения вида a.add(b) или a.less(b). Для всех операторов языка существуют специальные методы, которые вы можете реализовать в своих классах. Однако придумать новые операторы и создать для них свои методы невозможно.

Ниже приводится пример класса, для которого реализована работа с некоторыми операторами. Какому оператору или функции соответствует метод указано в комментариях. Для двуместных операций (+, -, <, >, ==, !=) левым операндом является текущий объект (доступ через self), правый операнд передается через параметр метода. Обратите внимание, что методы __add_,_sub_создают и возвращают новый объект класса Vector2D, а методы iadd__,___isub___изменяют текущий объект и его же возвращают.

```python
import math

class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        #вызывается при выводе в интерактивной оболочке
        #вызывается функцией repr
        #вызывается при отсутствии __str__
        #обычно результат - строка, создающая объект
        return 'Vector2D({}, {})'.format(self.x, self.y)

    def __str__(self):
        #вызывается функциями str, print и format
        #обычно результат - строка, понятная человеку
        return '({}, {})'.format(self.x, self.y)
    def __add__(self, B):   #сложение A + B
        return Vector2D(self.x + B.x, self.y + B.y)

    def __sub__(self, B):   #вычитание (A - B)
        return Vector2D(self.x - B.x, self.y - B.y)
```

```
    def __iadd__(self, B):  # A += B
        self.x += B.x
        self.y += B.y
        return self

    def __isub__(self, B):   # A -= B
        self.x -= B.x
        self.y -= B.y
        return self

    def __neg__(self):   #унарный минус (-A)
        return Vector2D(-self.x, -self.y)

    def __len__(self):   #вызывается функцией len
        return math.sqrt(self.x**2 + self.y**2)

    def __lt__(self, B):   #A<B
        l1=math.sqrt(self.x**2 + self.y**2)
        l2=math.sqrt(B.x**2 + B.y**2)
        return l1 < l2

    def __gt__(self, B):   #A>B
        l1=math.sqrt(self.x**2 + self.y**2)
        l2=math.sqrt(B.x**2 + B.y**2)
        return l1 > l2

    def __eq__(self, B):   #A==B
        return self.x==B.x and self.y==B.y

    def __ne__(self, B):   #A!=B
        return self.x!=B.x or self.y!=B.y

a = Vector2D(3, 4)
print(a)
b = Vector2D(5, 6)
c=a+b-Vector2D(1, 1)
print(c)
print( -a)
print(a<b, a>b, a==b, a!=b)
```

Задания для самостоятельной работы

Имеется класс, в котором реализуется работа с обыкновенными дробями. Напишите реализацию тех методов, в которых она отсутствует. Проверьте работу этих методов.

```
class Cfract: #обыкновенная дробь

    def __reduce(a, b): #сокращаем дробь
        a1 = abs(a)
        b1 = b
        if a1 == 0:
            return 0, 1
        while a1 != b1:
            if a1 > b1:
                a1 = a1 - b1
            else:
                b1 = b1 - a1
        if a1>1:
            return a//a1, b//a1
        else:
            return a, b

    def __init__(self, w, n, d):
        #w - целая часть дроби, n - числитель, d - знаменатель
        self.__n = abs(w)*d + abs(n) #числитель неправильной дроби
        if (w < 0 and n >= 0) or (w == 0 and n < 0):
           self.__n = -self.__n
        self.__d = d #знаменатель
        self.__n, self.__d = Cfract.__reduce(self.__n, self.__d)

    def __str__(self):   #вызывается функциями str, print и format
        if self.__n == 0:
            return '0'
        elif abs(self.__n) < self.__d:
            return f'{self.__n}/{self.__d}'
        else:
            s = str(abs(self.__n) // self.__d)
            if self.__n < 0:
                s = '-' + s
            if abs(self.__n) % self.__d != 0:
                s += f' {abs(self.__n) % self.__d}/{self.__d}'
            return s

    def __float__(self):
        #преобразование в вещественное число с помощью float()
        pass
```

```
def __add__(self, y):   #сложение x + y
    if type(y) == Cfract:
        a = self.__n * y.__d + y.__n * self.__d
        b = self.__d * y.__d
        a,b = Cfract.__reduce(a, b)
        if a > 0:
            return Cfract(a//b, a % b, b)
        elif a < 0:
            return Cfract(-(abs(a) // b), abs(a) % b, b)
        else:
            return Cfract(0, 0, 1)
    elif type(y) == int:
        return self + Cfract(y, 0, 1)
    else:
        print('недопустимое значение операнда')

def __radd__(self, y):   #сложение y + x
    pass

def __iadd__(self, y):   #сложение и присваивание x +=y
    pass



def __sub__(self, y):   #вычитание x - y
    pass

def __rsub__(self, y):   #вычитание y - x
    pass

def __isub__(self, y):   #вычитание и присваивание x -= y
    pass

def __neg__(self):   #унарный минус -x
    pass



def __mul__(self, y):   #умножение x * y
    pass

def __rmul__(self, y):   #умножение y * x
    pass

def __imul__(self, y):   #умножение и присваивание x *= y
    pass
```

```
    def __truediv__(self, y):   #деление x / y
        pass

    def __rtruediv__(self, y):   #деление y / x
        pass

    def __itruediv__(self, y):   #деление и присваивание x /= y
        pass


    def __abs__(self):  #абсолютное значение
        pass

    def __lt__(self, y):   #x < y
        pass

    def __gt__(self, y):   #x > y
        pass

    def __eq__(self, y):   #x == y
        pass

    def __ne__(self, y):   #x != y
        pass

    def __le__(self, y):   #x <= y
        pass

    def __ge__(self, y):   #x >= y
        pass

a = Cfract(0,-2, 3)
print(a)
b = Cfract(-1,1,3)
print(b)
c=a+b+2
print(c)
```

Инкапсуляция

Инкапсуляция — это третий столп объектно-ориентированного программирования. **Инкапсуляция** просто означает скрытие данных. Как правило, в объектно-ориентированном программировании **один класс не должен иметь прямого доступа к данным другого класса**. Вместо этого, доступ должен контролироваться через методы класса.

Чтобы предоставить контролируемый доступ к данным класса в Python, используются **модификаторы доступа** и **свойства**. Мы уже ознакомились с тем, как действуют модификаторы доступа. В этом разделе мы посмотрим, как действуют **свойства**.

Предположим, что нам нужно убедиться в том, что модель автомобиля должна датироваться между 2000 и 2018 годом. Если пользователь пытается ввести значение меньше 2000 для модели автомобиля, значение автоматически установится как 2000, и если было введено значение выше 2018, оно должно установиться на 2018. Если значение находится между 2000 и 2018 — оно остается неизменным. Мы можем создать свойство атрибута модели, которое реализует эту логику. Взглянем на пример:

```
1  # создаем класс Car
2  class Car:
3
4      # создаем конструктор класса Car
5      def __init__(self, model):
6          # Инициализация свойств.
7          self.model = model
8
9      # создаем свойство модели.
10     @property
11     def model(self):
12         return self.__model
13
14     # Сеттер для создания свойств.
15     @model.setter
16     def model(self, model):
17         if model < 2000:
```

```
18            self.__model = 2000
19        elif model > 2018:
20            self.__model = 2018
21        else:
22            self.__model = model
23
24    def getCarModel(self):
25        return "Год выпуска модели " + str(self.model)
26
27 carA = Car(2088)
28 print(carA.getCarModel())
```

Свойство имеет три части. Вам нужно **определить атрибут**, который является моделью в скрипте выше. Затем, вам нужно **определить свойство атрибута**, используя декоратор `@property`. Наконец, вам нужно создать **установщик свойства**, который является дескриптором `@model.setter` в примере выше. Теперь, если вы попробуете ввести значение выше `2018` в атрибуте модели, вы увидите, что значение установлено на `2018`. Давайте проверим это. Выполним следующий скрипт:

```
1 car_a = Car(2088)
2 print(car_a.get_car_model())
```

Здесь мы передаем `2088` как значение для модели, однако, если вы введете значение для атрибута модели через функцию `get_car_model()`, вы увидите `2018` в выдаче.

Требования к результатам выполнения работы

Условия успешной сдачи лабораторной работы:

1. Подготовлено пошаговое описание решений представленных задач.
2. Разработана программная реализация представленных задач.
3. Подготовлено краткое описание разработанного программного кода.
4. Программная реализация представленных примеров выложены в личный репозиторий на GitHub.

Литература

Копырин, А. С. Программирование на Python : учебное пособие / А. С. Копырин, Т. Л. Салова. — Сочи : СГУ, 2018. — 48 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/147665 (дата обращения: 15.07.2021). — Режим доступа: для авториз. пользователей.