МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Информатика»
Тема: Парадигмы программировния

Студент гр. 0382 _____ Злобин А. С.

Преподаватель _____ Шевская Н. В.

Санкт-Петербург
2020

Цель работы.

Создать систему классов для градостроительной компании, используя парадигму ООП на языке Python.

Задание.

Создать систему классов для градостроительной компании:

- HouseScheme - базовый класс (схема дома);
  - Поля класса:
    - количество жилых комнат
    - жилая площадь (в квадратных метрах)
    - совмещенный санузел (значениями могут быть или False, или True) При несоответствии переданного значения вызвать исключение.
- CountryHouse — деревенский дом (наследник HouseScheme);
  - Поля класса:
    - все поля класса HouseScheme
    - количество этажей
    - площадь участка При несоответствии переданного значения вызвать исключение.
  - Переопределяемые методы:
    - __str__()
    - __eq__()
- Apartment — квартира городская (наследник HouseScheme);
  - Поля класса:
    - все поля класса HouseScheme
    - этаж (может быть число от 1 до 15)
    - куда выходят окна (значением может быть одна из строк: N, S, W, E)) При несоответствии переданного значения вызвать исключение.

- Определяемые методы:
  - __str__()
- CountryHouseList — список деревенских домов (наследник list);
  - Методы: 2
    - Конструктор:
      1. Вызвать конструктор базового класса
      2. Передать в конструктор строку name и присвоить её полю name созданного объекта
    - append(p_object): Переопределение метода append() списка. В случае, если p_object - деревенский дом, элемент добавляется в список, иначе выбрасывается исключение TypeError с текстом: Invalid type <тип_объекта p_object>тип_объекта p_object>
    - total_square()I: Посчитать общую жилую площадь.
- ApartmentList — список городских квартир (наследник list);
  - Методы:
    - Конструктор:
      1. Вызвать конструктор базового класса
      2. Передать в конструктор строку name и присвоить её полю name созданного объекта
    - extend(iterable): Переопределение метода extend() списка. В случае, если элемент iterable - объект класса Apartment, этот элемент добавляется в список, иначе не добавляется.
    - floor_view(floors, directions): В качестве параметров метод получает диапазон возможных этажей в виде списка. Метод должен выводить квартиры, этаж которых входит в диапазон и окна которых выходят в одном из направлений

Основные теоретические положения.

Объектно-ориентированная парадигма базируется на нескольких принципах: наследование, инкапсуляция, полиморфизм. Наследование - специальный механизм, при котором мы можем расширять классы, усложняя их функциональность. В наследовании могут участвовать минимум два класса: суперкласс (или класс-родитель, или базовый класс) - это такой класс, который был расширен. Все расширения, дополнения и усложнения класса-родителя реализованы в классе-наследнике (или производном классе, или классе потомке) - это второй участник механизма наследования. Наследование позволяет повторно использовать функциональность базового класса, при этом не меняя базовый класс, а также расширять ее, добавляя новые атрибуты. В парадигме ООП основными понятиями являются:

1) Объект — конкретная сущность предметной области;

2) Класс — тип объекта.

Синтаксис создания класс в python:

class <Название класса>(<название класса посредника>):

<тело класса>

Синтаксис создания объекта класса:

<название объекта> = <тназвание класса>()

Классы содержат атрибуты, которые подразделяются на поля и методы.

Методы — функции, определяемые внутри класса. Синтаксис создания методов аналогичен синтаксису создания функции, первый аргумент которой всегда self Синтаксис обращения к методу:

<название объекта класса или класса>.<название метода>()

Поля — переменные, определяемые внутри класса. Синтаксис создания полей аналогичен синтаксису создания переменных. Синтаксис обращения к полю:

<название объекта класса или класса>.<название поля>

Существует специальный метод для создания объекта класса — конструктор. Конструктор вызывается при создании объекта класса. Синтаксис создания конструктора:

def __init__(self, <var 1>, <var 2> ...):

Для обращения к элементу класса посредника используется функция super():

super().<название элемента>

Специальный метод __str__(self) отвечает за строковое представление объекта.

Специальный метод __eq__(self, other) предназначен для перегрузки бинарного оператора «==».

Также существует возможность переопределения методов класса посредника, для этого необходимо создать метод с таким же названием в классе наследнике.

Исключения - это специальный класс объектов в языке Python. Исключения предназначены для управления поведением программой, когда возникает ошибка, или, другими словами, для управления теми участками программного кода, где может возникнуть ошибка.

Классы исключений выстроены в специальную иерархию. Есть основной класс BaseException - базовое исключение, от которого берут начало все остальные. Берут начало – в контексте ООП-парадигмы – наследуются. От BaseException наследуются системные и обычные исключения. Системными исключениями являются: SystemExit, GeneratorExit и KeyboardInterrupt. У этих исключений нет встроенных наследников; вмешиваться в работу системных 5 исключений не рекомендуется. Вторая группа наследников класса BaseE)xception – это обычные исключения – класс E)xception. Встроенные наследники класса Exception: AttributeError, SyntaxError, TypeError, ValueError.

Для самостоятельного вызова исключение используется оператор raise.

Функция filter(). Синтаксис функции:

filter(<функция>, <объект>)

Функция <функция> применяется для каждого элемента итерируемого объекта <объект> и возвращает объект-итератор, состоящий из тех элементов итерируемого объекта <объект>, для которых <функция> является истиной.

Лямбда-выражения – это специальный элемент синтаксиса для создания анонимных (т.е. не имеющих имени) функций сразу в том месте, где эту функцию необходимо вызвать. Используя лямбда-выражения можно объявлять функции в любом месте кода, в том числе внутри других функций.

Пример:

lambda a,b: a + b

Для функции filter(<функция>, <объект>) в качестве аргумента <функция> может быть передано lambda-выражение.

Принцип работы функции filter остается такой же: функция возвращает объект-итератор, состоящий из тех элементов итерируемого объекта <объект> , для которых <функция> является истиной.

Как и в случае с обычной функцией в качества аргумента, lambdaвыражение применяется для каждого элемента итерируемого объекта <объект>.

Выполнение работы.

1) Класс HouseScheme.

В этом классе необходимо было инициализировать следующие поля объекта класса:

• self.rooms — количество комнат;

• self.area — жилая площадь;

• self.shared_bath — совмещенный санузел;

Инициализация этих полей происходит в методе-конструкторе, в котором предусмотрена передача необходимых аргументов. Если аргументы соответствуют требованиям (площадь не может быть отрицательной, а

значения аргумента «shared_bath» имеют тип bool), то соответствующим полям присваиваются их значения. В противном случае оператором raise вызывается исключение ValueError с текстом «Invalid value»Invalid value».

2) Класс CountryHouse. Наследник класса HouseScheme.

Помимо полей класса-родителя, в этом классе необходимо было инициализировать следующие поля объекта класса:

• self.floors — количество этажей;

• self.area — площадь земельного участка;

Инициализация происходит аналогично инициализации в классе HouseScheme. Также в этом классе необходимо переопределить метод str и бинарный оператор «==». Переопределённый метод str возвращает отформатированную строку с информацией об объекте. Оператор «==» переопределён следующим образом:

```
def __eq__(self, second):
        if type(second) != CountryHouse:
            raise ValueError("Invalid value")
        return self.area == second.area and self.land_area == second.land_area and abs(self.floors - second.floors) <= 1
```

3) Класс Apartment. Наследник класса HouseScheme. Помимо полей класса-родителя, в этом классе необходимо было инициализировать следующие поля объекта класса:

• self.floor — этаж, на котором располагается квартира;

• self.windows_of_Apartment — направление, куда выходят окна квартиры;

Инициализация происходит аналогично инициализации в классе HouseScheme. Также в этом классе необходимо переопределить метод str. Переопределённый метод str возвращает отформатированную строку с информацией об объекте.

4) Класс CountryHouseList. Наследник класса list. В этом классе необходимо было определить поле объекта класса self.name. Также в конструкторе данного класса, как и в предыдущих классах, имеющих класс-родитель, вызывается конструктор класса-родителя. Переопределение метода append происходит следующим образом:

```
def append(self, p_object):
        if type(p_object) == CountryHouse:
            super().append(p_object)
        else:
            raise TypeError("Invalid type {}".format(type(p_object)))
```

Также в этом классе определяется метод total_square для подсчёта общей площади объектов класса CountryHouse, входящих в список. В этом методе с помощью цикла for происходит суммирование всех площадей. Метод возвращает полученную сумму.

5) Класс ApartmentList. Наследник класса list. В этом классе необходимо было определить поле объекта класса self.name. Также в конструкторе данного класса, как и в предыдущих классах, имеющих класс-родитель, вызывается конструктор класса-родителя. Переопределение метода extend происходит следующим образом:

```
def extend(self, iterable):
    super().extend(list(filter(lambda i: type(i) == Apartment, iterable)))
```

Также определён метод floor_view, который позволяет узнать, выходят ли окна квартиры (объекта класса Apartment) на одну из требуемых сторон и находится ли квартира на одном из требуемых этажей. Иерархия классов:

- Родитель: HouseScheme
  - Наследники: Apartment, CountryHouse,
- Родитель: list
  - Наследники: ApartmentList, CountryHouseList.

Методы базовых классов, которые переопределялись при выполнении:

• __str__(self,..);

• __eq__(self, other) (бинарный оператор «==»);

• append(self, p_object);

• extend(self, lst);

• __init__(self,…);

Описание конкретных методов можно прочитать в описании классов. Метод __str__() будет вызван при приведении объекта класса к строковому типу (например при вызове функции str() или при использовании в функции print()). Не переопределённые методы класса list для классов CountryHouseList и ApartmentList работать будут, потому что ApartmentList и CountryHouseList являются наследниками класса list, поэтому все методы, присущие классу list будут также работать для всех объектов классов-наследников. Например не переопределённый метод self.count() вернёт количество элементов текущего списка — объекта класса ApartmentList или CountryHouse.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	-	Country House: Количество жилых комнат 10, Жилая площадь 100, Совмещенный санузел True, Количество этажей 2, Площадь	Программа работает верно

		участка 1000. False True	

Вывод:

В ходе работы было выполнено задание — построена система классов для градостроительной компании. Задание выполнялось при помощи ООП парадигмы на языке Python. Реализована система классов, представлена иерархия классов, переопределены все необходимые методы классов, инициализированы поля объектов классов. Также в программном коде были предусмотрены возможности возникновения исключительных ситуаций и реализован вывод соответствующих исключений. Также были использованы такие элементы парадигмы функционального программирования на Python как lamda-выражения и функция filter().

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class HouseScheme:
	def __init__(self, rooms, area, bath):
		if area >= 0 and type(bath) == bool:
			self.rooms = rooms
			self.area = area
			self.shared_bath = bath
		else:
			raise ValueError("Invalid value")
class CountryHouse(HouseScheme):
	def __init__(self, *CountryHouse_input):
		if len(CountryHouse_input) != 5 or CountryHouse_input[0] <= 0 or CountryHouse_input[1] <= 0 or isinstance(CountryHouse_input[2], bool) != True or CountryHouse_input[3] <= 0 or CountryHouse_input[4] < 0:
			raise ValueError('Invalid value')
		super().__init__(CountryHouse_input[0], CountryHouse_input[1], CountryHouse_input[2])
		self.floors = CountryHouse_input[3]
		self.land_area = CountryHouse_input[4]

	def __str__(self):
		return "Country House: Количество жилых комнат {}, Жилая площадь {}, Совмещенный санузел {}, Количество этажей {}, Площадь участка {}.".format(self.rooms, self.area, self.shared_bath, self.floors, self.land_area)

	def __eq__(self, second):
		if type(second) != CountryHouse:
		    raise ValueError("Invalid value")
		return self.area == second.area and self.land_area == second.land_area and abs(self.floors - second.floors) <= 1


class Apartment(HouseScheme):
	def __init__(self, *Apartment_input):
		if len(Apartment_input) != 5 or Apartment_input[0] <= 0 or Apartment_input[1] <= 0 or isinstance(Apartment_input[2], bool) != True or Apartment_input[3] <= 0 or Apartment_input[3] > 15 or (Apartment_input[4] in ['N', 'S', 'W', 'E']) != True:
			raise ValueError('Invalid value')
		super().__init__(Apartment_input[0], Apartment_input[1], Apartment_input[2])
		self.floor = Apartment_input[3]
		self.windows_of_Apartment = Apartment_input[4]

	def __str__(self):
		return "Apartment: Количество жилых комнат {}, Жилая площадь {}, Совмещенный санузел {}, Этаж {}, Окна выходят на {}.".format(self.rooms, self.area, self.shared_bath, self.floor, self.windows_of_Apartment)
```

```
     class CountryHouseList(list):
          def __init__(self, name):
               super().__init__()
               self.name = name
          def append(self, p_object):
               if type(p_object) == CountryHouse:
                    super().append(p_object)
               else:
                    raise           TypeError("Invalid           type
{}".format(type(p_object)))

          def total_square(self):
               all_area = 0
               for i in range(len(self)):
                    all_area += self[i].area
               return all_area


     class ApartmentList(list):
          def __init__(self, name):
               super().__init__()
               self.name = name
          def extend(self, iterable):
               super().extend(list(filter(lambda    i:    type(i)    ==
Apartment, iterable)))
          def floor_view(self, floors, directions):
               for x in list(filter(lambda i: floors[0] <= i.floor <=
floors[1] and i.windows_of_Apartment in directions, self)):
                    print("{}:     {}".format(x.windows_of_Apartment,
x.floor))
```