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 программирования

Студентка гр. 0382 _____ Довченко М.К.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2020

Цель работы.

Освоение парадигм программирования и исключений в языке Python.

Задание.

Система классов для градостроительной компании

Базовый класс -- схема дома HouseScheme:

```
class HouseScheme:
    ''' Поля объекта класса HouseScheme:
    количество жилых комнат
    площадь (в квадратных метрах, не может быть отрицательной)
    совмещенный санузел (значениями могут быть или False, или True)
```

При создании экземпляра класса HouseScheme необходимо убедиться, что переданные в конструктор параметры удовлетворяют требованиям, иначе выбросить исключение ValueError с текстом

```
    'Invalid value'
    '''
```

Дом деревенский CountryHouse:

```
class CountryHouse: # Класс должен наследоваться от HouseScheme
    '''Поля объекта класса CountryHouse:
    количество жилых комнат
    жилая площадь (в квадратных метрах)
    совмещенный санузел (значениями могут быть или False, или True)
    количество этажей
    площадь участка
```

При создании экземпляра класса CountryHouse необходимо убедиться, что переданные в конструктор параметры удовлетворяют требованиям, иначе выбросить исключение ValueError с текстом

```
    'Invalid value'
    '''
    Метод __str__()
    '''Преобразование к строке вида:
```

Country House: Количество жилых комнат <количество жилых комнат>, Жилая площадь <жилая площадь>, Совмещенный санузел <совмещенный санузел>, Количество этажей <количество этажей>, Площадь участка <площадь участка>.

'''

Метод __eq__()

'''Метод возвращает True, если два объекта класса равны и False иначе.

Два объекта типа CountryHouse равны, если равны жилая площадь, площадь участка, при этом количество этажей не отличается больше, чем на '''

Квартира городская Apartment:

class Apartment: # Класс должен наследоваться от HouseScheme

''' Поля объекта класса Apartment:

количество жилых комнат

площадь (в квадратных метрах)

совмещенный санузел (значениями могут быть или False, или True)

этаж (может быть число от 1 до 15)

куда выходят окна (значением может быть одна из строк: N, S, W, E)

При создании экземпляра класса Apartment необходимо убедиться, что переданные в конструктор параметры удовлетворяют требованиям, иначе выбросить исключение ValueError с текстом

'Invalid value'

'''

Метод __str__()

'''Преобразование к строке вида:

Apartment: Количество жилых комнат <количество жилых комнат>, Жилая площадь <жилая площадь>, Совмещенный санузел <совмещенный санузел>, Этаж <этаж>, Окна выходят на <куда выходят окна>.

Переопределите список list для работы с домами:

Деревня:

class CountryHouseList: # список деревенских домов -- "деревня", наследуется от класса list

Конструктор:

'''1. Вызвать конструктор базового класса

2. Передать в конструктор строку name и присвоить её полю name созданного объекта'''

Метод 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>'''

Метод total_square():

'''Посчитать общую жилую площадь'''

Жилой комплекс:

class ApartmentList: # список городских квартир -- ЖК, наследуется от класса list

Конструктор:

'''1. Вызвать конструктор базового класса

2. Передать в конструктор строку name и присвоить её полю name созданного объекта

'''

Метод extend(iterable):

'''Переопределение метода extend() списка.

В случае, если элемент iterable - объект класса Apartment, этот элемент добавляется в список, иначе не добавляется.

'''

Метод floor_view(floors, directions):

'''В качестве параметров метод получает диапазон возможных этажей в виде списка (например, [1, 5]) и список направлений из ('N', 'S', 'W', 'E').

Метод должен выводить квартиры, этаж которых входит в переданный диапазон (для [1, 5] это 1, 2, 3, 4, 5) и окна которых выходят в одном из переданных направлений. Формат вывода:

<Направление_1>: <этаж_1>

<Направление_2>: <этаж_2>

...

Направления и этажи могут повторятся. Для реализации используйте функцию filter().

'''

Основные теоретические положения.

Лямбда-выражения – это специальный элемент синтаксиса для создания анонимных 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. Синтаксис определения следующий: *lambda аргумент1, аргумент2,..., аргументN : выражение.*

Еще одна очень полезная реализация функционального программирования – 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.

Объектно-ориентированная парадигма базируется на нескольких принципах: наследование, инкапсуляция, полиморфизм. Наследование позволяет повторно использовать функциональность базового класса, при этом не меняя базовый класс, а также расширять ее, добавляя новые атрибуты. 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.

Объект - конкретная сущность предметной области, тогда как класс - это тип объекта. Классы содержат атрибуты, которые подразделяются на поля и методы. Под методом понимают функцию, которая определена внутри класса.

Функция *isinstance(obj_, class_):*

Функция возвращает *True,* если *obj_* является экземпляром класса *class_* или если *class_* является суперклассом для класса, объектом которого является *obj_*.

Функция *issubclass(class1, class2):*

Функция возвращает *True*, если *class1* является наследником класса *class2* (или наследником наследника, с любым уровнем вложенности). Класс считается наследником самого себя.

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. Исключения в Python реализованы на основе ООП-парадигмы с использованием наследования и других принципов. Любое исключение – это объект. У объекта-исключения есть определенный тип, то есть, как мы уже знаем, определенный класс. Классы исключений выстроены в специальную иерархию. Есть основной класс *BaseException* - базовое исключение, от которого берут начало все остальные.

Выполнение работы.

Класс *HouseScheme()*. Не наследуется, является родителем для классов *Apartment* и *CountryHouse*. В конструкторе происходит инициализация полей объектов класса *count_rooms*(количество жилых комнат), *square*(жилая площадь), *bathroom*(совмещенный санузел). Параметры, переданные в конструктор, проверяются: если они не удовлетворяют условиям, с помощью

raise создаётся и выбрасывается исключение *ValueError* с текстом 'Invalid value' .

Класс *CountryHouse(HouseScheme)*. Наследник класса *HouseScheme*, не является родителем. В конструкторе происходит наследование полей объектов класса *HouseScheme* - *count_rooms , square, bathroom* и инициализируются поля – *count_floors*(количество этажей), *square2*(площадь участка). Параметры, переданные в конструктор, проверяются: если они не удовлетворяют условиям, с помощью *raise* создаётся и выбрасывается исключение *ValueError* с текстом 'Invalid value' .

Далее происходит переопределение метода *__str__(self)*, который возвращает строку заданного формата, и метода *__eq__(self, object2)*, который возвращает *True*, если два объекта класса, переданные в метод, равны, и *False,* если нет.

Класс *Apartment(HouseScheme)*. Наследуется от класса *HouseScheme*, не является родителем. В конструкторе наследуются поля объекта класса *HouseScheme* - c*ount_rooms , square, bathroom* и инициализируются поля – *floor*(этаж), *window*(куда выходят окна). Параметры, переданные в конструктор, проверяются: если они не удовлетворяют условиям, с помощью *raise* создаётся и выбрасывается исключение *ValueError* с текстом 'Invalid value' .

Далее переопределяется метод *__str__(self)*, который возвращает строку заданного формата.

Класс *CountryHouseList(list)*. Наследник класса *list*, не является родителем. В конструкторе инициализируется поле объекта класса – *name* (полю класса присваивается аргумент-строки *name*). Далее происходит переопределение метода *append(self, p_object)*, в котором проверяется переданный аргумент. Если аргумент *p_object* удовлетворяет заданным условиям, то элемент добавляется в список, иначе выбрасывается исключение *TypeError* с текстом: *'Invalid type <тип_объекта p_object>'*. Также происходит переопределение метод *total_square(self)*, в котором происходит вычисление

общей жилой площади текущего объекта класса. Полученное значение возвращается.

Класс *ApartmentList(list)*. Наследуется от класса *list*, не является родителем. В конструкторе инициализируется поле объекта класса – *name*(полю класса присваивается аргумент-строки *name*). Далее переопределяется метод списка - *extend(self, iterable)*, в котором осуществляется проверка, элемента *iterable*. Если элемент - объект класса *Apartment,* то он добавляется в список. Переопределяется метод *floor_view(self, floors, directions),* который в качестве параметров получает диапазон возможных этажей в виде списка и список направлений из ('N', 'S', 'W', 'E'). Выводятся квартиры, подходящие заданным условиям, окна которых выходят в одном из переданных направлений. Происходит преобразование в строку заданного формата.

1. Иерархия описанных классов.

CountryHouse(потомок) – *HouseScheme*(родитель)

Apartment(потомок) – *HouseScheme*(родитель)

CountryHouseList(потомок) - *list*(родитель)

ApartmentList(потомок) - *list*(родитель)

2. Методы, которые были переопределены:

def __init__(self,);

def __str__(self);

def __eq__(self, object2);

def append(self, p_object);

def extend(self, iterable).

3. Метод *__str__*() будет вызван:

При вызове функции *str*() - приведении к типу “строка” в явном виде, или неявном, как, например, при вызове функции *print()*.

4.Будут ли работать непереопределенные методы класса *list* для *CountryHouseList* и *ApartmentList*? Объясните почему и приведите примеры.

Будут, но если не переопределять - будут работать в их базовом формате, как обычные функции класса *list*, ведь он является родителем классов *CountryHouseList* и *ApartmentList*.

Пример: метод *list.clear*(), если его не переопределить, будет очищен нынешний список, являющийся объектом класса *CountryHouseList* или *ApartmentList*.

Разработанный программный код см. в приложении А.

Тестирование.

Экземпляры классов:

house1 = CountryHouse(5, 40, True, 2, 850)

house2 = CountryHouse(5, 40, True, 3, 850)

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	print(house1 == house2)	True	Верный вывод.
2.	print(house1)	Country House: Количество жилых комнат 5, Жилая площадь 40, Совмещенный санузел True, Количество этажей 2, Площадь участка 850.	Верный вывод.
3.	list1 = CountryHouseList('Country') list1.append(house1) print(list1)	[<__main__.CountryHouse object at 0x000002820B350190>]	Верный вывод.

Выводы.

Были освоены исключения и объектно-ориентированная парадигма в языке Python.

Разработана программа, в которой описаны классы и их методы, были учтены случаи возникновения исключений.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb3.py

```
class HouseScheme:
    def __init__(self, count_rooms, square, bathroom):
        if (count_rooms > 0) and (square > 0) and (type(bathroom)
== bool):
            self.count_rooms = count_rooms
            self.square = square
            self.bathroom = bathroom
        else:
            raise ValueError('Invalid value')


class CountryHouse(HouseScheme):
          def __init__(self, count_rooms, square, bathroom,
count_floors, square2):
        super().__init__(count_rooms, square, bathroom)
        if count_floors > 0 and square2 > 0:
            self.count_floors = count_floors
            self.square2 = square2
        else:
            raise ValueError('Invalid value')

    def __str__(self):
         return 'Country House: Количество жилых комнат {}, Жилая
площадь {}, Совмещенный санузел {}, Количество этажей {}, Площадь
участка {}.'.format(
                    self.count_rooms, self.square, self.bathroom,
self.count_floors, self.square2)

    def __eq__(self, other):
            if self.square == other.square and self.square2 ==
other.square2 and abs(
                self.count_floors - other.count_floors) <= 1:
            return True
        else:
            return False


class Apartment(HouseScheme):
       def __init__(self, count_rooms, square, bathroom, floor,
window):
        super().__init__(count_rooms, square, bathroom)
         if floor >= 1 and floor <= 15 and window in ['N', 'S',
'W', 'E']:
            self.floor = floor
            self.window = window
        else:
            raise ValueError('Invalid value')

    def __str__(self):
```

```
                return 'Apartment: Количество жилых комнат {}, Жилая площадь {}, Совмещенный санузел {}, Этаж {}, Окна выходят на {}.'.format(
                self.count_rooms, self.square, self.bathroom, self.floor, self.window)


    class CountryHouseList(list):
        def __init__(self, name):
            super().__init__()
            self.name = name

        def append(self, p_object):
            if issubclass(type(p_object), CountryHouse):
                super().append(p_object)
            else:
                raise TypeError(f'Invalid type {type(p_object)}')

        def total_square(self):
            __total_area = 0
            for i in self:
                __total_area += i.square
            return __total_area


    class ApartmentList(list):
        def __init__(self, name):
            super().__init__()
            self.name = name

        def extend(self, iterable):
               super().extend(filter(lambda x: type(x) == Apartment, iterable))


        def floor_view(self, floors, directions):
            checking = list(filter(lambda i: (i.window in directions) and (i.floor in list(range(floors[0], floors[1] + 1))), self))
            for i in checking:
                print('{}: {}'.format(i.window, i.floor))
```