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 _____ Ситченко К.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2020

Цель работы.

Изучить парадигмы программирования и на практике использовать ООП на языке Python.

Задание.

Описать систему классов для градостроительной компании на языке Python.

Основные теоретические положения.

Объект – конкретная сущность предметной области.

Класс – тип объекта.

Методы класса – функция, которая принадлежит классу. В языке Python первый аргумент метода – self, экземпляр класса. При описании метода пишется явно. При вызове метода передается неявно. Методы имеют доступ к полям экземпляра класса.

Конструктор – метод, который вызывается при создании экземпляра класса. Конструктор ничего не возвращает. Конструктор может быть не описан, тогда создастся пустой объект. Конструктор может быть только один, но может иметь переменные по умолчанию.

Поле (атрибут) объекта – некоторая переменная, которая лежит в области видимости и доступна во внешней программе через синтаксис <имя_объекта>.<поле>. Поля объекта устанавливаются в методах класса через обращение к экземпляру self.

Некоторые основные принципы ООП:

- Инкапсуляция (сокрытие внутренней реализации от пользователя; сокрытие деталей реализации за интерфейсом объекта)
- Наследование (повторное использование и последующее расширение одним классом атрибутов другого класса; класс, определенный через наследование от другого класса, называется производным классов, классом-

потомком, классом-наследником; класс, от которого новый класс наследуется, называется предком, базовым классом или суперклассом)

- Полиморфизм (способность функции обрабатывать разные типы данных, если эти данные могут поддерживать соответствующий интерфейс; возможность обрабатывать объекты разных типов одинаковым образом, не задумываясь о типе каждого объекта)

В Python существует возможность переопределения не только методов класса, но и операторов выражений. Вы можете создать свой тип данных и определить для его экземпляров операции сложения/сравнения/извлечения среза и т.д.

Парадигма программирования – совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Существуют императивная и декларативная парадигмы.

Лямбда-выражения – это специальный элемент синтаксиса для создания анонимных (т.е. без имени) функций по месту их использования. Используя лямбда-выражения можно объявлять функции в любом месте кода, в том числе внутри других функций. Лямбда-выражение может иметь неограниченное количество аргументов, однако производит только одно действие и чаще всего используется только в одном месте кода. В Python лямбда-выражения упрощают запись и использование однострочных операций.

Функция filter(<функция>, <объект>) возвращает объект-итератор, состоящий из тех элементов итерируемого объекта <объект>, для которых <функция> является истиной. Функция <функция> применяется для каждого элемента итерируемого объекта <объект>. После того, как к итератору произошло обращение, из него извлекаются элементы. Чтобы воспользоваться результатами работы функции filter и после обращения к объекту-итератору, нужно обернуть вызов функции filter в функцию list(): list(filter(..., ...)).

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, lambda-выражение применяется для каждого элемента итерируемого объекта <объект>. После того, как к итератору произошло обращение, из него извлекаются элементы.

Выполнение работы.

Иерархия классов, описанных в программе:

Родитель: HouseScheme

Наследники: CountryHouse, Apartment

Родитель: list

Наследники: CountryHouseList, ApartmentList

1. Создание класса HouseScheme

Для объектов данного класса было необходимо инициализировать такие поля объекта как *self.amout_of_rooms* (количество жилых комнат, *self.square* (жилая площадь), *self.combined_bathroom* (совмещенный санузел). Для этого был использован метод-конструктор *__init__*. Если жилая площадь отрицательна или тип переменной *combined_bathroom* не является булевым, то программа выведет на экран исключение ValueError с текстом "Invalid value"

2. Создание класса CountryHouse

К полям класса-родителя (HouseScheme), для вызова которых использовалась функция *super()*, добавились поля: *self.amount_of_floors* (количество этажей), *self.plottage* (площадь участка), чья инициализация проходила с помощью метода-конструктора *__init__*. В данном классе переопределен метод *__str__()*, который выводит на экран строку с

информацией об объекте класса, и определен метод *__eq__()*, который возвращает True, если два объекта класса равны и False иначе.

3. Создание класса Apartment

Так же как и в классе CountryHouse, в класс Apartment были переданы родительские поля объекта, к которым добавились поля *self.floor* (этаж) и *self.windows_direction* (куда выходят окна), чья инициализация проходила с помощью метода-конструктора *__init__*. В данном классе переопределен метод *__str__()*, который выводит на экран строку с информацией об объекте класса.

4. Создание класса CountryHouseList

В данном классе с помощью метода-конструктора *__init__* было инициализировано поле объекта *self.name*. Для этого класса был переопределен метод *append(p_object)*: в случае, если *p_object* - деревенский дом, элемент добавляется в список, иначе выбрасывается исключение TypeError с текстом: Invalid type <тип_объекта p_object>. Также был определен метод *total_square*, который считает общую жилую площадь объектов, входящих в список.

5. Создание класса ApartmentList

В данном классе с помощью метода-конструктора *__init__* было инициализировано поле объекта *self.name*. Для этого класса был переопределен метод *extend(iterable)*: в случае, если элемент *iterable* - объект класса Apartment, этот элемент добавляется в список, иначе не добавляется. Также определен метод *floor_view*, который выводит на экран квартиры, этаж которых входит в переданный диапазон и окна которых выходят в одном из переданных направлений. Для реализации метода была использована функция *filter()*.

Метод *__str__()* будет использован в случаях, когда требуется привести объект к строковому типу.

Не переопределенные методы класса list будут работать как для CountryHouseList, так и для ApartmentList, так как они являются

наследниками класса list, а значит для них будут работать все методы класса list. Например, CountryHouseList.clear очистит список, а ApartmentList.reverse развернет список.

Разработанный программный код см. в приложении А.

Тестирование.

Для тестирования программы в нее были добавлены следующие данные:

```
a = CountryHouse(5, 200, False, 2, 400)
b = CountryHouse(7, 200, True, 1, 400)
print(a)
print(b)
print(a.__eq__(b))

chl = CountryHouseList("Village")
chl.append(a)
print(chl.total_square)

c = Apartment(3, 80, True, 13, 'S')
e = Apartment(3, 80, True, 14, 'N')
f = Apartment(3, 80, True, 9, 'S')
g = Apartment(4, 85, False, 13, 'E')
h = Apartment(4, 85, False, 10, 'W')
print(c)

al = ApartmentList("Block")
al.extend([c, e, f, g, h])
print(al.floor_view([10, 15], ['S', 'N']))
```

Программа сработала верно и вывела результат, представленный ниже:

Country House: Количество жилых комнат 5, Жилая площадь 200, Совмещенный санузел False, Количество этажей 2, Площадь участка 400.

Country House: Количество жилых комнат 7, Жилая площадь 200, Совмещенный санузел True, Количество этажей 1, Площадь участка 400.

True

<bound method CountryHouseList.total_square of [<__main__.CountryHouse object at 0x000001DC3D5DDA30>]>

Apartment: Количество жилых комнат 3, Жилая площадь 80, Совмещенный санузел True, Этаж 13, Окна выходят на S.

S: 13

N: 14

При добавлении во входные данные строки `d = Apartment(4, 85, False, 16, 'W')` программа выводила исключение ValueError: Invalid value, так как указанное количество этажей не совпадало с установленным диапазоном.

При добавлении во входные данные строки `chl.append(c)` программа выводила исключение TypeError: Invalid type <class '__main__.Apartment'>, так как тип переменной не совпадает с заданным в методе типом(Country_House).

Выводы.

Были изучены парадигмы программирования.

С помощью объектно-ориентированной парадигмы на языке Python была разработана программа, содержащая систему классов градостроительной компании. Также были использованы исключения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

1. Название файла: main.py

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


class CountryHouse(HouseScheme):
  def __init__(self, amout_of_rooms, square, combined_bathroom, amount_of_floors, plottage):
    super().__init__(amout_of_rooms, square, combined_bathroom)
    self.amount_of_floors = amount_of_floors
    self.plottage = plottage

  def __str__(self):
    return "Country House: Количество жилых комнат {}, Жилая площадь {}, Совмещенный санузел {}, Количество этажей {}, Площадь участка {}.".format(self.amout_of_rooms, self.square, self.combined_bathroom, self.amount_of_floors, self.plottage)

  def __eq__(self, another):
    if type(another) != CountryHouse: raise ValueError("Invalid value")
    return self.square == another.square and self.plottage == another.plottage and abs(self.amount_of_floors - another.amount_of_floors) <= 1


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

  def __str__(self):
    return "Apartment: Количество жилых комнат {}, Жилая площадь {}, Совмещенный санузел {}, Этаж {}, Окна выходят на {}.".format(self.amout_of_rooms, self.square, self.combined_bathroom, self.floor, self.windows_direction)


class CountryHouseList(list):
  def __init__(self, name):
```

```
      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):
    total_square = 0
    for item in self:
      total_square += item.square
    return total_square


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

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

  def floor_view(self, floors, windows_directions):
    apartment_verification    =    list(filter(lambda    element:
floors[0]     <=     element.floor     <=     floors[1]     and
element.windows_direction in windows_directions, self))
    for item in apartment_verification:
      print('{}: {}'.format(item.windows_direction,item.floor))
```