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

Студент гр.0382 _____ Диденко Д.В.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2020

Цель работы.

Рассмотреть понятия парадигм программирования и освоить некоторые из них на практике.

Задание.

Написать систему классов для градостроительной компании по определенному шаблону.

Основные теоретические положения.

Итератор - это специальный объект, который делает проще переходы по элементам другого объекта. Итератор – это своего рода перечислитель для определенного объекта (например, списка, строки, словаря), который позволяет перейти к следующему элементу этого объекта, либо бросает исключение, если элементов больше нет.

Итерируемый объект – объект, по которому можно итерироваться (то есть который можно обходить в цикле, например, цикле for). Чем же тогда итерируемый объект отличается от итератора? Итератор – это надстройка над итерируемым объектом. Из итерируемого объекта всегда можно получить итератор с помощью встроенной функции iter()

Функция map().

Функция map() принимает на вход два параметра: функцию и последовательность (итерируемый объект, iterable). Функция работает следующим образом: применяет к элементам итерируемого объекта (объектов) переданную функцию. Функция map() в Python возвращает объект-итератор типа map.

lambda-выражения

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.

Функция filter()

На ряду с функцией map(), есть 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.

Функция zip().

Функция zip(*iterables) получает на вход несколько итерируемых объектов (чаще – списков) и возвращает объект-итератор (в Python 3, в более ранних версиях языка – list), состоящий из элементов-кортежей.

Объектно-ориентированное программирование.

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. Поле - это переменная, которая определена внутри класса, Конструктор - это специальный метод, который нужен для создания объектов класса.

1. Конструктор в языке Python всегда имеет название __init__().
2. При создании объекта вызывается конструктор.
3. Чтобы вызвать конструктор, мы используем название класса.
4. Конструктор - это метод, который ничего не возвращает.
5. Первый аргумент любого метода класса в языке Python - экземпляр класса (т.е. объект), для которого этот метод вызывается. Обычно он имеет название self; в других языках часто используется ключевое слово this.
6. В теле конструктора обычно происходит инициализация различных полей класса через обращение к экземпляру self.
7. Все методы имеют доступ к полям объекта.
8. В Python нельзя создать два конструктора с разным количеством аргументов, как, например, в языке C++, однако, используя механизм аргументов по умолчанию, можно добиться аналогичного поведения.

Создание класса.

Синтаксис создания класса:

class <Название_класса>:

<Тело_класса>

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.

Под инкапсуляцией часто понимают сокрытие внутренней реализации от пользователя. В других языках программирования это достигается использованием модификаторов доступа; таким образом, в описании класса мы можем указать, какой атрибут будет доступен извне, а какой нет.

Мы можем передать в такую функцию числа любого типа, строки, а также списки и кортежи. Таким образом, наша функция magic_sum() может работать с разными типами данных, если они поддерживают операцию сложения. Такое свойство функции говорит о том, что она полиморфна.

Исключения.

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

исключений не рекомендуется. Вторая группа наследников класса BaseException – это обычные исключения – класс Exception. Встроенные наследники класса Exception: AttributeError, SyntaxError, TypeError, ValueError.

Обработчик исключений try-except-else-finally.

В блок try помещают код, который может вызвать исключительную ситуацию (потенциально опасный код).

В блок except помещают код для обработки исключительной ситуации. Для одного try-блока может быть несколько except-блоков.

В блок finally помещают код, который должен выполниться в любом случае, вне зависимости от того, произошла исключительная ситуация или нет. Блок finally может быть только один.

В блок else помещают код, который должен выполниться, если в try-блоке не случилось исключительной ситуации (else-блок выполняется в случае, если утверждение "Исключительная ситуация произошла" ложно). Блок else может быть только один.

Чтобы самостоятельно сгенерировать исключительную ситуацию используют блок raise.

Выполнение работы.

Исходный код решения задачи см.в приложении А.

В решении задания описана система классов для градостроительной компании. Переопределены методы append() и extend() для двух подклассов класса list – CountryHouseList и ApartmentList. Создан родительский класс HouseScheme и его подклассы CountryHouse, Apartment. Для класса HouseScheme использовали метод __init__() с помощью которого инициализировали объект с полями count_bedroom, square, joined_bathroom. При этом поле square определено неотрицательными значениями (на отрицательные с помощью блока raise вызывается ошибка класса Exception - ValueError), а поле joined_bathroom двумя значениями – True или False, на остальные вызывается вышеупомянутое исключение.

В подклассе CountryHouse инициализирован объект с наследованными полями родительского класса и добавленными полями count_floor и square_place. Переопределен метод __str__(), вызывающийся при использовании функции print() вне класса. Определен метод __eq__(), принимающий на вход соответственно self и объект подкласса CountryHouse.

В подклассе Apartment инициализирован объект с наследованными полями родительского класса и добавленными полями floor(принимает значение от 1 до 15) и side_of_window (принимает одно из направлений: 'N', 'S', 'W', 'E'). Переопределен метод __str__(), вызывающийся при использовании функции print() вне класса.

В подклассе CountryHouseList передается в конструктор строка name и присваивается её значение полю name созданного объекта. Переопределен метод append(p_object) так, что в случае, если p_object – объект класса CountryHouse, элемент добавляется в список, иначе выбрасывается исключение TypeError с текстом: Invalid type <тип_объекта p_object>. Определен метод total_square(), считающий общую жилую площадь объектов, находящихся в списке, через обращение к полю объекта square.

В подклассе ApartmentList передается в конструктор строка name и присваивается её значение полю name созданного объекта. Переопределен метод extend(iterable) так, что, если элемент iterable - объект класса Apartment, этот элемент добавляется в список, иначе не добавляется. Определен метод floor_view(floors, directions), принимающий диапазон этажей floors и список сторон. Метод выводит квартиры, этаж которых входит в переданный диапазон и окна которых выходят в одном из переданных направлений. Метод реализован с помощью функции filter.

Непереопределенные методы класса list в подклассах CountryHouseList и ApartmentList работать будут. Так как они наследуются от родительского класса и никак не изменяются. Например, CountryHouseList(‘Name’).pop() – удалит последний элемент списка и вернет его значение, ApartmentList(‘Name’).append(a) – добавит а конец списка элемент а.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a = CountryHouse(6, 110, True,1, 120) print(a)	Country House: Количество жилых комнат 6, Жилая площадь 110, Совмещенный санузел True, Количество этажей 1, Площадь участка 120.	Программа работает верно
2.	print(a.count_bedroom) print(a.square) print(a.joined_bathroom) print(a.count_floor) print(a.square_place)	6 110 True 1 120	Программа работает верно
3.	b = CountryHouse(6, 110, True,1, 120) print(a.__eq__(b))	True	Программа работает верно
4.	c=Apartment(2,50,True,6,'N') print(c)	Apartment: Количество жилых комнат 2, Жилая площадь 50, Совмещенный санузел True, Этаж 6, Окна выходят на N.	Программа работает верно
5.	print(c.count_bedroom) print(c.square) print(c.joined_bathroom) print(c.floor) print(c.side_of_window)	2 50 True 6 N	Программа работает верно

6.	lst=CountryHouseList("Name") lst.append(a) print(lst)	[<__main__.CountryHouse object at 0x000001712AA1B3A0>]	Программа работает верно
7.	d = 23 lst=CountryHouseList("Name") lst.append(d) print(lst)	TypeError: Invalid type <class 'int'>	Программа работает верно
8.	w = [1,2,3,4,'werverv',c] lst2=ApartmentList("Name") lst2.extend(w) print(lst2)	[<__main__.Apartment object at 0x000001C9E503B490>]	Программа работает верно
9.	c=Apartment(2,50,True,4,'N') f=Apartment(2,50,True,6,'N') g=Apartment(2,50,True,3,'S') h=Apartment(2,50,True,2,'W') w = [c,f,g,h] lst2.extend(w) print(lst2.floor_view([1,6],['N']))	N: 4 N: 6	Программа работает верно

Выводы.

Было рассмотрено понятие парадигмы программирования. Некоторые парадигмы программирования освоили на практике, решив ряд задач с использованием Python.

Была представлена классификация парадигм с примерами языков программирования. Особое внимание было уделено реализации

функционального программирования на Python с решением задач на практике. Изучены ключевые вопросы теории объектно-ориентированного программирования, а также рассмотрены примеры задач на Python. Было определено понятие исключительной ситуации, решены задачи обработки исключительных ситуаций, а также генерация исключений на Python.

Разработана программа, содержащая систему классов для градостроительной компании. Использовались основные принципы ООП и применялись исключения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class HouseScheme:
    def __init__(self, count_bedroom, square, joined_bathroom):
        self.count_bedroom = count_bedroom
        if square < 0:
            raise ValueError('Invalid value')
        else:
            self.square = square
        if type(joined_bathroom) == bool or joined_bathroom == 'True'
or joined_bathroom == 'False':
            self.joined_bathroom = joined_bathroom
        else:
            raise ValueError('Invalid value')


class CountryHouse(HouseScheme):
    def __init__(self, count_bedroom, square, joined_bathroom,
count_floor, square_place):
        super().__init__(count_bedroom, square, joined_bathroom)
        self.count_floor = count_floor
        self.square_place = square_place

    def __str__(self):
        return 'Country House: Количество жилых комнат {}, Жилая площадь
{}, Совмещенный санузел {}, Количество этажей {}, Площадь участка
{}.'.format(self.count_bedroom,self.square,self.joined_bathroom,self.c
ount_floor,self.square_place)
    def __eq__(self,el):
        if self.square == el.square and self.square_place ==
el.square_place and abs(self.count_floor-el.count_floor) <= 1:
            return True
        else:
            return False

class Apartment(HouseScheme):
    def
__init__(self,count_bedroom,square,joined_bathroom,floor,side_of_windo
w):
        super().__init__(count_bedroom,square,joined_bathroom)
        if 1<=floor<=15:
            self.floor = floor
        else:
            raise ValueError('Invalid value')
        if side_of_window in ['N', 'S', 'W', 'E']:
            self.side_of_window = side_of_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_bedroom,self.square,self.joined_bathroom,self.f
loor,self.side_of_window)
```

```
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_squre = 0
        for i in self:
            total_squre+=i.square
        return total_squre

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