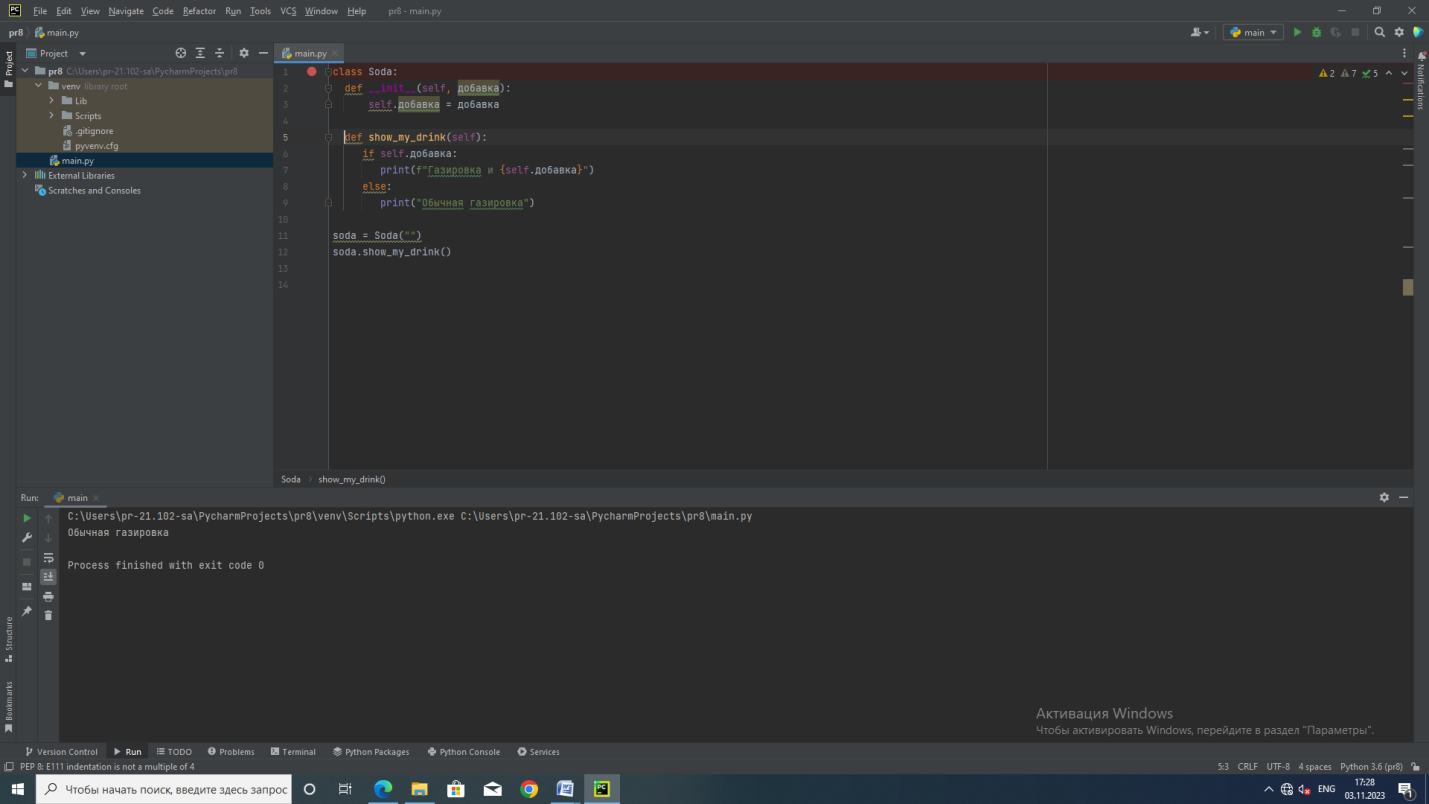
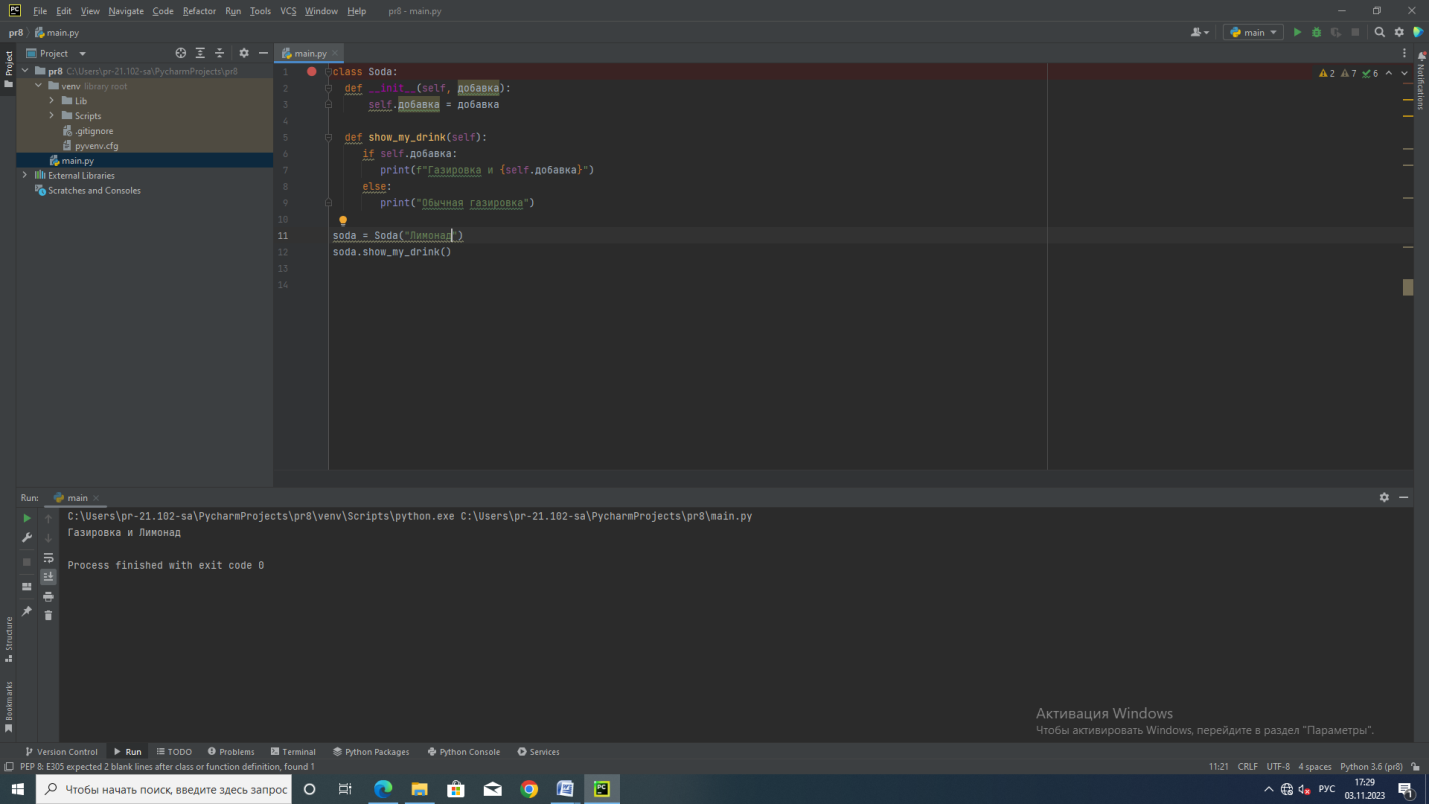
Саая А.О. ПР-21.102

Практическая работа №8

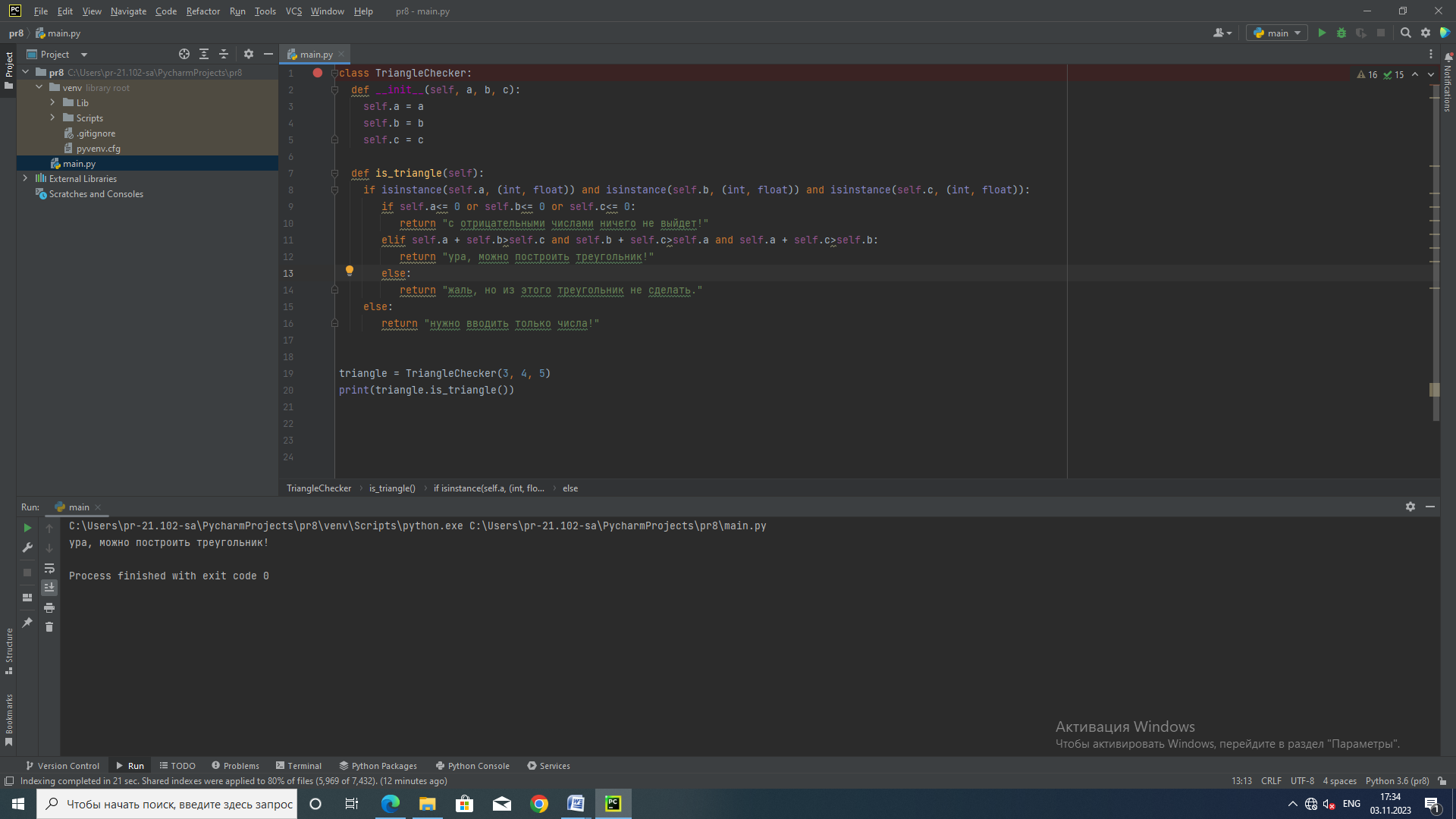
1. Создайте класс Soda (для определения типа газированной воды), принимающий 1 аргумент при инициализации (отвечающий за добавку к выбираемому лимонаду). В этом классе реализуйте метод show\_my\_drink(), выводящий на печать «Газировка и {ДОБАВКА}» в случае наличия добавки, а иначе отобразится следующая фраза: «Обычная газировка».

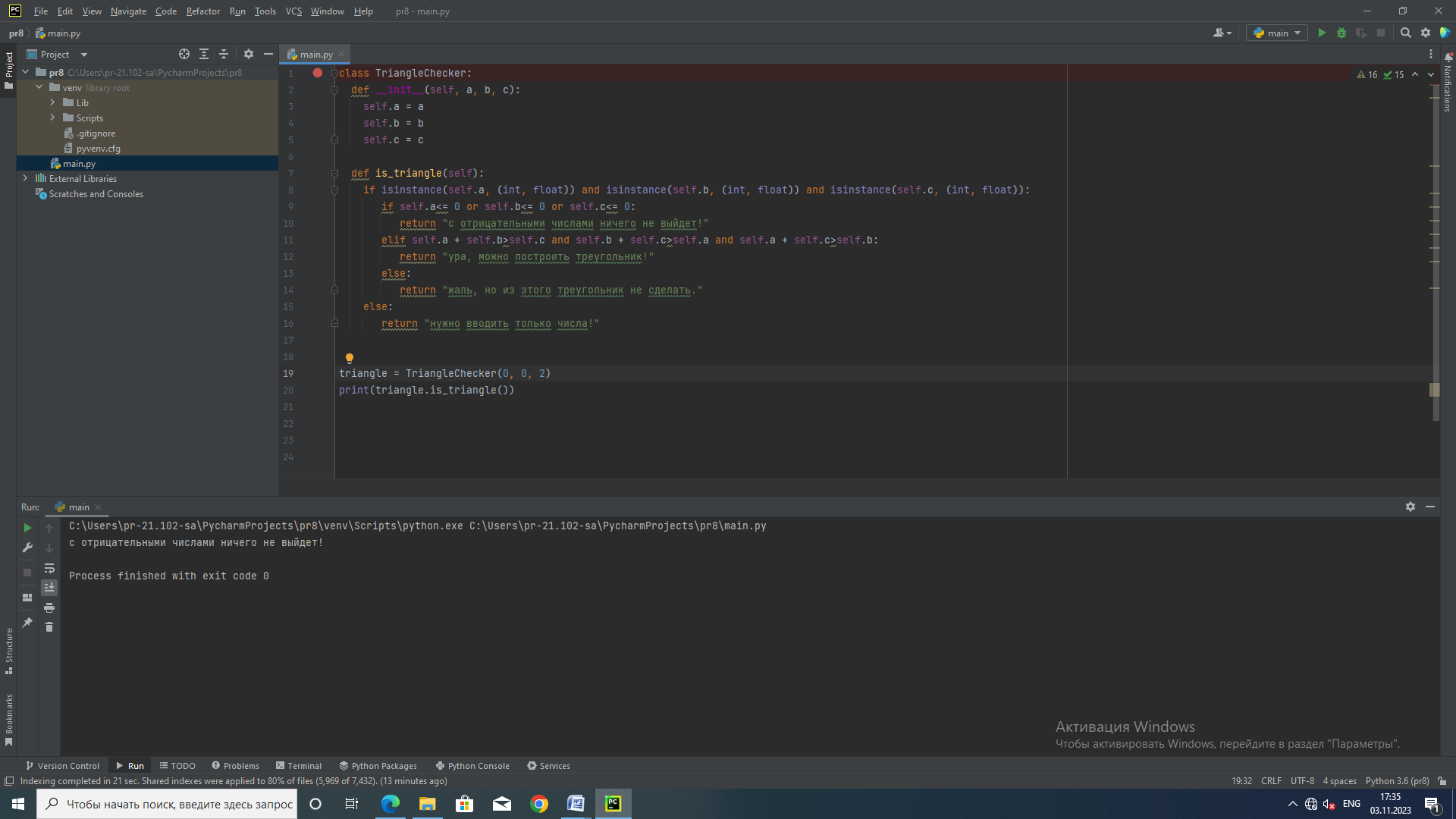
class Soda:  
 def \_\_init\_\_(self, добавка):#конструктор init принимает аргумент добавка  
 self.добавка = добавка  
  
 def show\_my\_drink(self):#метод, который выводит информацию о напитке  
 if self.добавка:   
 print(f"Газировка и {self.добавка}")  
 else:  
 print("Обычная газировка")  
  
soda = Soda("")#экземпляр класса  
soda.show\_my\_drink() #вызов метода

1. Николаю требуется проверить, возможно ли из представленных отрезков условной длины сформировать треугольник. Для этого он решил создать класс TriangleChecker, принимающий только положительные числа. С помощью метода is\_triangle() возвращаются следующие значения (в зависимости от ситуации): – Ура, можно построить треугольник!; – С отрицательными числами ничего не выйдет!; – Нужно вводить только числа!; – Жаль, но из этого треугольник не сделать.

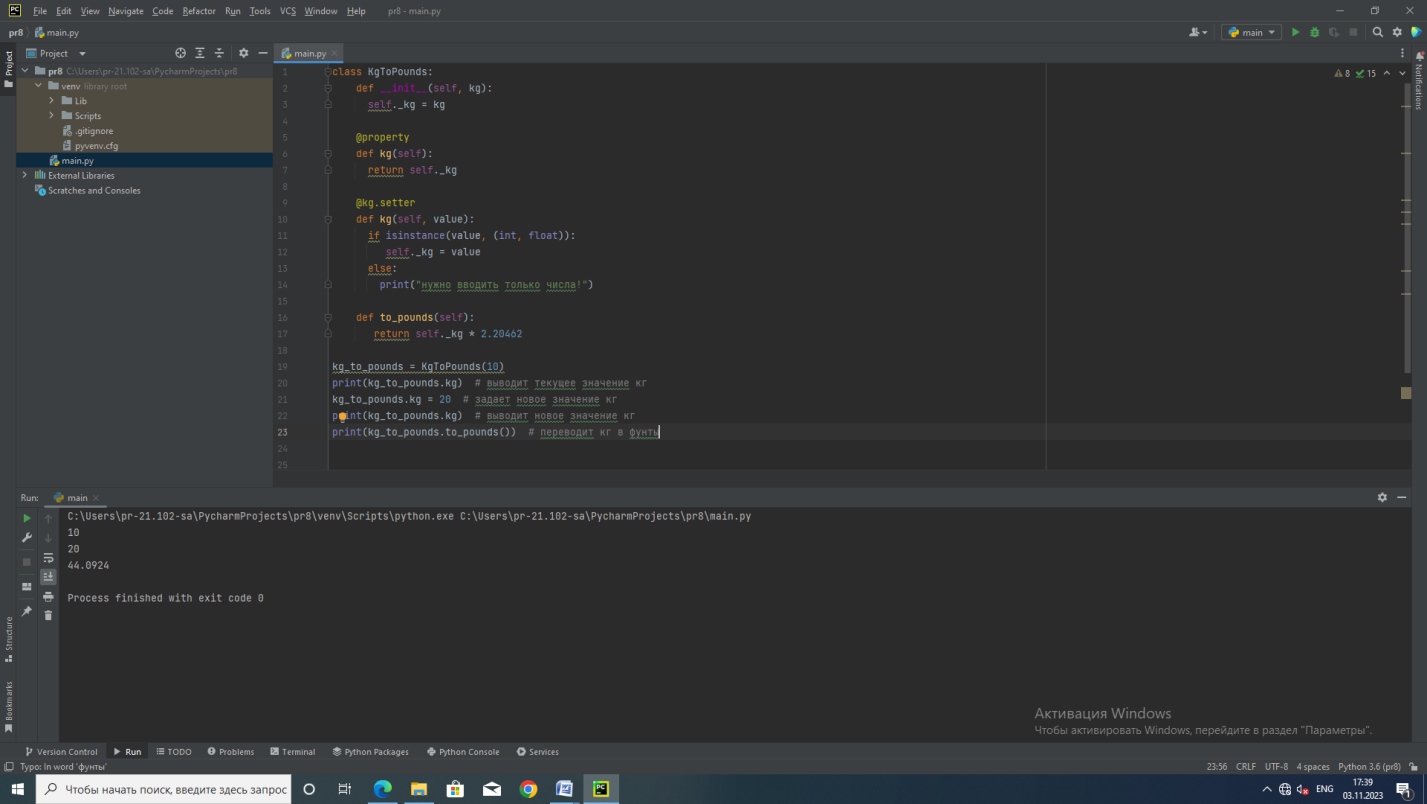
class TriangleChecker:  
 def \_\_init\_\_(self, a, b, c):  
 self.a = a  
 self.b = b  
 self.c = c  
  
 def is\_triangle(self): #этот метод проверяет возможность построения треугольника с заданными сторонами  
 if isinstance(self.a, (int, float)) and isinstance(self.b, (int, float)) and isinstance(self.c, (int, float)):  
 if self.a<= 0 or self.b<= 0 or self.c<= 0:  
 return "с отрицательными числами ничего не выйдет!"  
 elif self.a + self.b>self.c and self.b + self.c>self.a and self.a + self.c>self.b:   
 return "ура, можно построить треугольник!"  
 else:  
 return "жаль, но из этого треугольник не сделать."  
 else:  
 return "нужно вводить только числа!"  
  
  
triangle = TriangleChecker(3, 4, 5)  
print(triangle.is\_triangle())





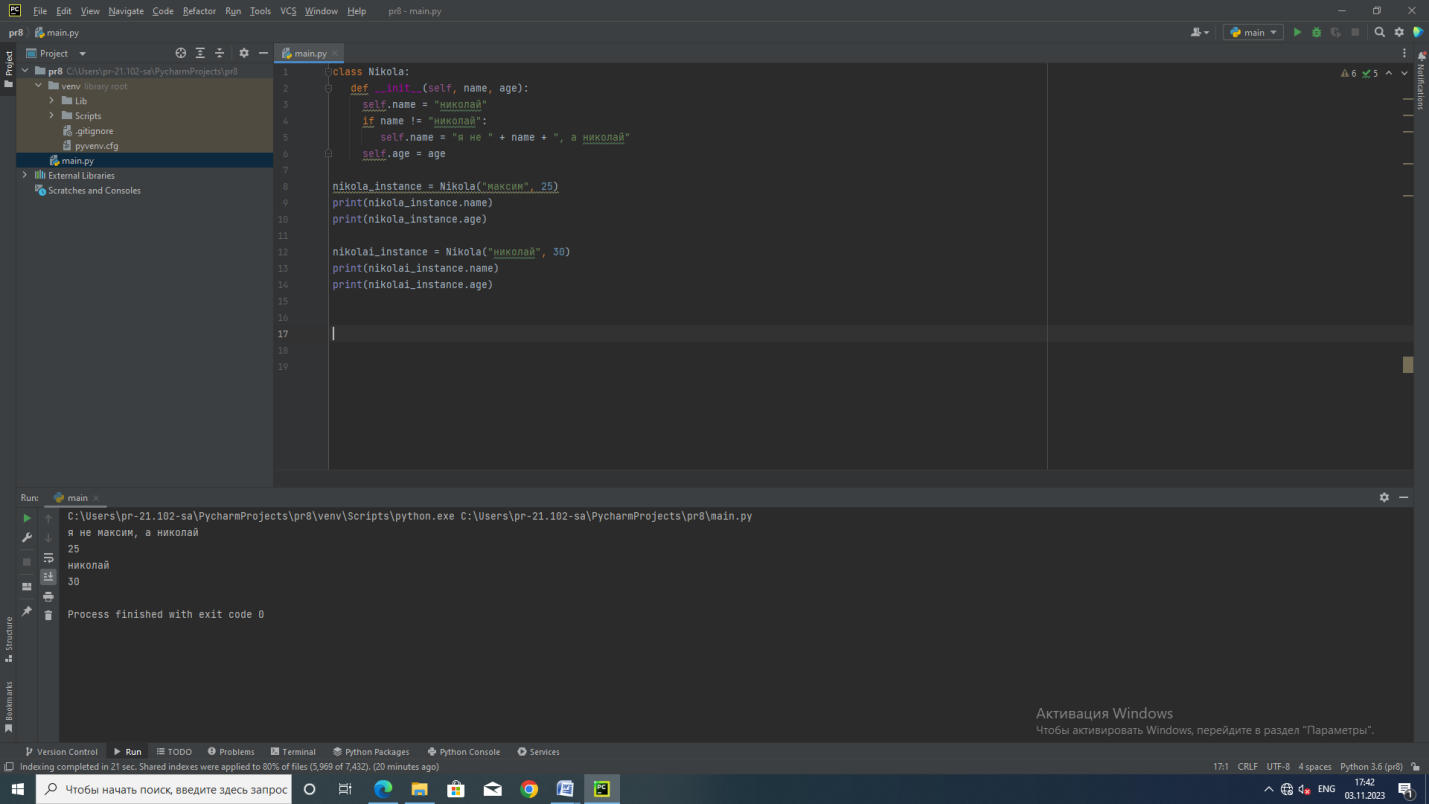
1. Евгения создала класс KgToPounds с параметром kg, куда передается определенное количество килограмм, а с помощью метода to\_pounds() они переводятся в фунты. Чтобы закрыть доступ к переменной “kg” она реализовала методы set\_kg() - для задания нового значения килограммов, get\_kg() - для вывода текущего значения кг. Из-за этого возникло неудобство: нам нужно теперь использовать эти 2 метода для задания и вывода значений. Помогите ей переделать класс с использованием функции property() и свойств-декораторов. Код приведен ниже.

class KgToPounds:  
 def \_\_init\_\_(self, kg):#конструктор класса принимает значение в кг и сохраняет в атрибут \_kg  
 self.\_kg = kg  
  
 @property #декоратор, который превращает метод кг в свойство для чтения  
 def kg(self):  
 return self.\_kg  
  
 @kg.setter #декоратор, который превращает метод кг в свойство для записи  
 def kg(self, value):  
 if isinstance(value, (int, float)):  
 self.\_kg = value  
 else:  
 print("нужно вводить только числа!")  
  
 def to\_pounds(self):#метод, который выполняет конвертацию значения в кг в фунты  
 return self.\_kg \* 2.20462  
  
kg\_to\_pounds = KgToPounds(10)#экземпляр класса  
print(kg\_to\_pounds.kg) # выводит текущее значение кг  
kg\_to\_pounds.kg = 20 # задает новое значение кг  
print(kg\_to\_pounds.kg) # выводит новое значение кг  
print(kg\_to\_pounds.to\_pounds()) # переводит кг в фунты



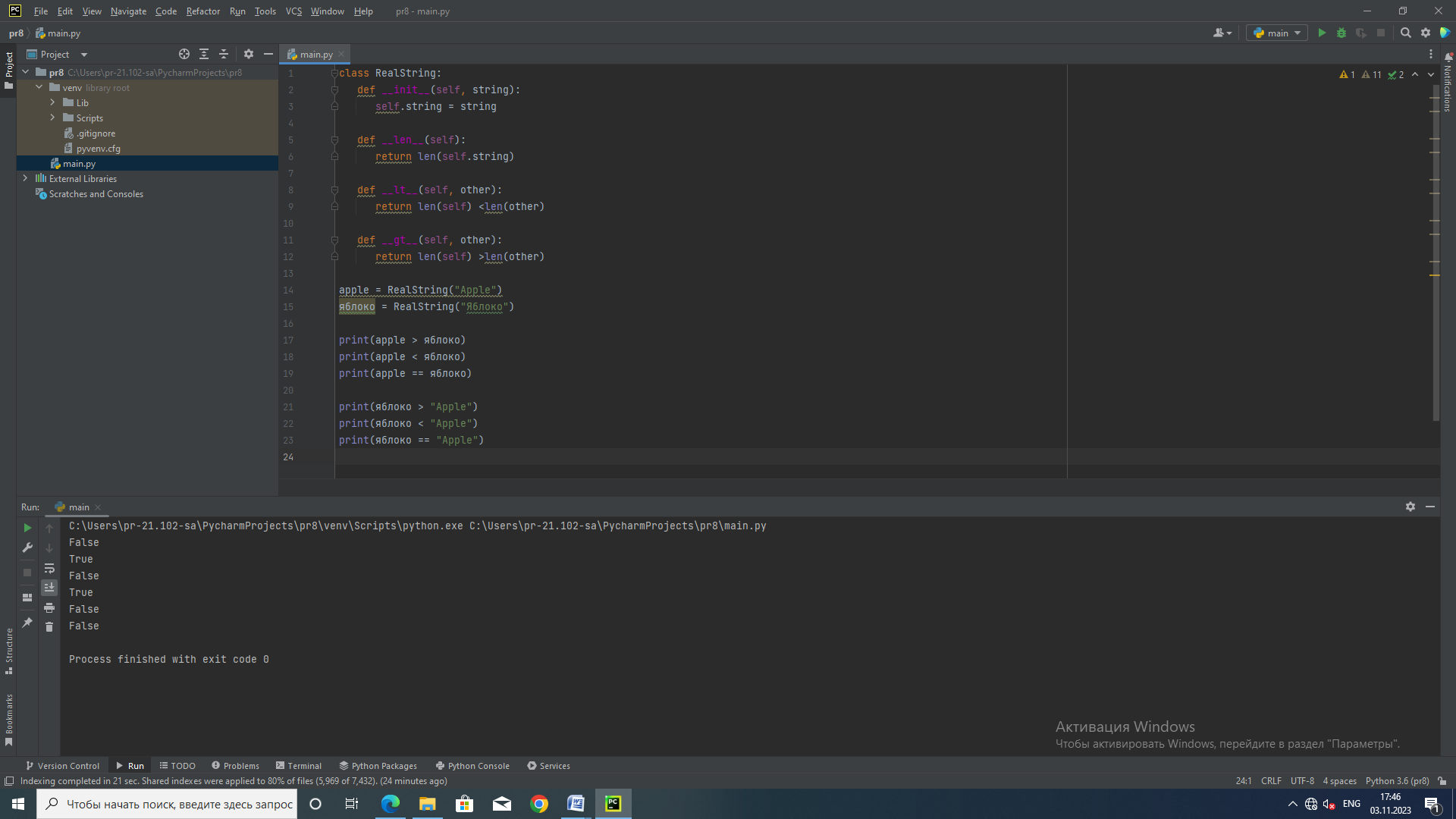
1. Николай – оригинальный человек. Он решил создать класс Nikola, принимающий при инициализации 2 параметра: имя и возраст. Но на этом он не успокоился. Не важно, какое имя передаст пользователь при создании экземпляра, оно всегда будет содержать “Николая”. В частности - если пользователя на самом деле зовут Николаем, то с именем ничего не произойдет, а если его зовут, например, Максим, то оно преобразуется в “Я не Максим, а Николай”. Более того, никаких других атрибутов и методов у экземпляра не может быть добавлено, даже если кто-то и вздумает так поступить (т.е. если некий пользователь решит прибавить к экземпляру свойство «отчество» или метод «приветствие», то ничего у такого хитреца не получится).

class Nikola:  
 def \_\_init\_\_(self, name, age):#конструктор принимает два параметра имя и возраст  
 self.name = "николай"  
 if name != "николай":  
 self.name = "я не " + name + ", а николай"  
 self.age = age  
  
nikola\_instance = Nikola("максим", 25)#экземпляр класса  
print(nikola\_instance.name)   
print(nikola\_instance.age)  
  
nikolai\_instance = Nikola("николай", 30)#экземпляр класса  
print(nikolai\_instance.name)  
print(nikolai\_instance.age)



1. Строки в Питоне сравниваются на основании значений символов. Т.е. если мы захотим выяснить, что больше: «Apple» или «Яблоко», – то «Яблоко» окажется бОльшим. А все потому, что английская буква «A» имеет значение 65 (берется из таблицы кодировки), а русская буква «Я» – 1071 (с помощью функции ord() это можно выяснить). Такое положение дел не устроило Анну. Она считает, что строки нужно сравнивать по количеству входящих в них символов. Для этого девушка создала класс RealString и реализовала озвученный инструментарий. Сравнивать между собой можно как объекты класса, так и обычные строки с экземплярами класса RealString. К слову, Анне понадобилось только 3 метода внутри класса (включая \_\_init\_\_()) для воплощения задуманного.

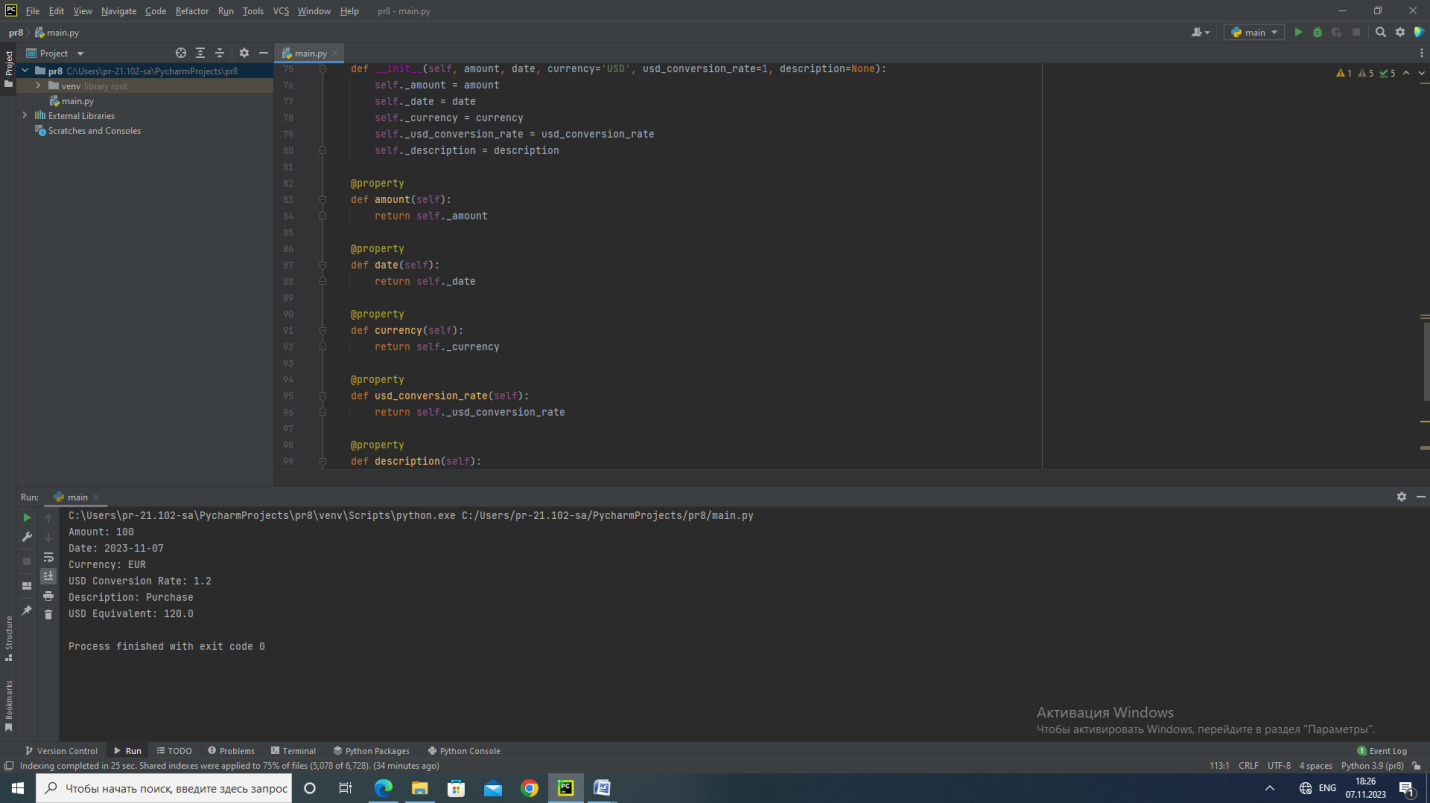
class RealString:  
 def \_\_init\_\_(self, string):#метод, который принимает строку в качестве аргумента  
 self.string = string  
  
 def \_\_len\_\_(self):#возвращает длину строки, которая будет использоваться для сравнения объектов класса  
 return len(self.string)  
  
 def \_\_lt\_\_(self, other):#сравнивает текущий объект с другим объектом и возвращает true, если длина текущей строки меньше длины другого  
 return len(self) <len(other)  
  
 def \_\_gt\_\_(self, other):# сравнивает текущий объект с другим объектом и возвращает true, если длина текущей строки больше длины другого  
 return len(self) >len(other)  
  
apple = RealString("Apple")  
яблоко = RealString("Яблоко")  
  
print(apple > яблоко)  
print(apple < яблоко)  
print(apple == яблоко)  
  
print(яблоко > "Apple")  
print(яблоко < "Apple")  
print(яблоко == "Apple")



Дополнительные задачи

1. Создайте класс Transaction, который хранит сумму, дату, валюту (по умолчанию «USD» – доллар США), курс валюты по отношению к доллару (по умолчанию 1) и описание (по умолчанию None). Все атрибуты данных должны быть частными. Реализуйте следующие свойства, доступные только для чтения: amount, date, currency, usd\_conversion\_rate, description и usd (вычисляется, как amount \* usd\_conversion\_rate).

class Transaction:  
 def \_\_init\_\_(self, amount, date, currency='USD', usd\_conversion\_rate=1, description=None):  
 self.\_amount = amount  
 self.\_date = date  
 self.\_currency = currency  
 self.\_usd\_conversion\_rate = usd\_conversion\_rate  
 self.\_description = description  
  
 @property  
 def amount(self):  
 return self.\_amount  
  
 @property  
 def date(self):  
 return self.\_date  
  
 @property  
 def currency(self):  
 return self.\_currency  
  
 @property  
 def usd\_conversion\_rate(self):  
 return self.\_usd\_conversion\_rate  
  
 @property  
 def description(self):  
 return self.\_description  
  
 @property  
 def usd(self):  
 return self.\_amount \* self.\_usd\_conversion\_rate  
  
transaction = Transaction(100, '2023-11-07', 'EUR', 1.2, 'Purchase')  
print(f"Amount: {transaction.amount}")  
print(f"Date: {transaction.date}")  
print(f"Currency: {transaction.currency}")  
print(f"USD Conversion Rate: {transaction.usd\_conversion\_rate}")  
print(f"Description: {transaction.description}")  
print(f"USD Equivalent: {transaction.usd}")



В этом классе мы определили свойства с использованием декоратора @property для доступа к атрибутам данных только для чтения. Теперь вы можете создавать объекты Transaction и получать доступ к их атрибутам и вычисляемым свойствам.

1. Реализуйте класс Account, который хранил бы номер счета, название счета и список транзакций (объектов класса Transaction). Номер счета должен быть реализован в виде свойства, доступного только для чтения. Название счета должно быть реализовано в виде свойства, доступного для чтения и для записи с проверкой длины названия, которое должно содержать не менее четырех символов. Класс должен поддерживать встроенную функцию len() (возвращая число транзакций) и содержать два вычисляемых свойства, доступных только для чтения: balance, возвращающее баланс счета в долларах США, и all\_usd, возвращающее True, если все транзакции выполнялись в долларах США, или False – в противном случае. Добавьте три дополнительных метода: apply() для добавления транзакции, save() и load(). Методы save() и load() должны сохранять и загружать объекты в двоичном формате, в файле, имя которого совпадает с номером счета и с расширением .acc. Они должны сохранять и загружать номер счета, название счета и все транзакции. Реализацию класса можно уместить в девяносто строк программного кода вместе с несколькими простейшими доктестами, включающими проверку операций сохранения и загрузки с помощью такого программного кода, как name = os.path.join(tempfile.gettempdir(), account\_name), который позволяет получить подходящее имя временного файла. Требуется удалить временные файлы по завершении доктестов.
2. import os  
   import tempfile  
   import pickle  
     
   class Transaction:  
    def \_\_init\_\_(self, amount, currency):  
    self.amount = amount  
    self.currency = currency  
     
   class Account:  
    def \_\_init\_\_(self, account\_number, account\_name):  
    self.\_account\_number = account\_number  
    self.\_account\_name = account\_name  
    self.\_transactions = []  
     
    @property  
    def account\_number(self):  
    return self.\_account\_number  
     
    @property  
    def account\_name(self):  
    return self.\_account\_name  
     
    @account\_name.setter  
    def account\_name(self, value):  
    if len(value) >= 4:  
    self.\_account\_name = value  
    else:  
    raise ValueError("Account name must be at least 4 characters long")  
     
    def apply(self, transaction):  
    self.\_transactions.append(transaction)  
     
    @property  
    def balance(self):  
    total = 0  
    for transaction in self.\_transactions:  
    if transaction.currency == "USD":  
    total += transaction.amount  
    return total  
     
    @property  
    def all\_usd(self):  
    for transaction in self.\_transactions:  
    if transaction.currency != "USD":  
    return False  
    return True  
     
    def save(self):  
    filename = os.path.join(tempfile.gettempdir(), f"{self.\_account\_number}.acc")  
    with open(filename, 'wb') as file:  
    pickle.dump(self, file)  
     
    @classmethod  
    def load(cls, account\_number):  
    filename = os.path.join(tempfile.gettempdir(), f"{account\_number}.acc")  
    with open(filename, 'rb') as file:  
    return pickle.load(file)  
     
   if \_\_name\_\_ == "\_\_main\_\_":  
    account = Account("12345", "Savings")  
    transaction1 = Transaction(100, "USD")  
    transaction2 = Transaction(50, "EUR")  
    account.apply(transaction1)  
    account.apply(transaction2)  
    account.save()  
     
   loaded\_account = Account.load("12345")  
   assert loaded\_account.account\_number == "12345"  
   assert loaded\_account.account\_name == "Savings"  
   assert len(loaded\_account.\_transactions) == 2