Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский экономический университет имени Г.В. Плеханова»

**МОСКОВСКИЙ ПРИБОРОСТРОИТЕЛЬНЫЙ ТЕХНИКУМ**

специальность 09.02.07 «Информационные системы и программирование»

Квалификация: Программист

ПРАКТИЧЕСКАЯ РАБОТА №2

ПО ОП.04 «Основы алгоритмизации и программирования»

|  |  |
| --- | --- |
| Выполнил студент  группы П-2-23  Саргсян Мариам Саргисовна | Проверил преподаватель  \_\_\_\_\_\_\_\_\_Т.Д. Артамонова  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2024 года |

Москва 2024

Цель работы: разработать консольное приложения – управление данными в ювелирном магазине с системой авторизации и основными операциями над данными. Для успешной реализации необходимо разработать структуру приложения, учитывая специфику выбранной области. Реализовать все функции: добавление, удаление, поиск, сортировка, фильтрация.

Ход работы:

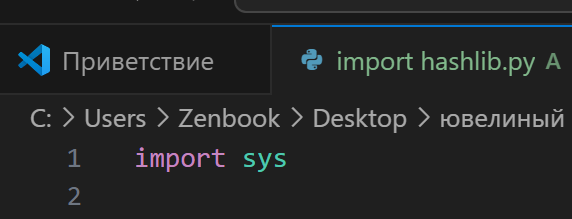


Рисунок 1 - импорты

import sys — это импорт встроенного модуля sys в Python.  Модуль sys предоставляет доступ к некоторым переменным и функциям, используемым или управляемым интерпретатором Python.

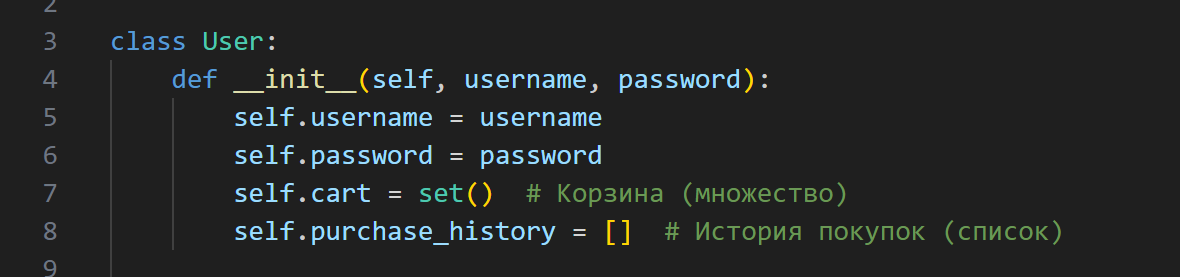


Рисунок 2 - класс User

\* `class User:`: Это ключевое слово, указывающее на определение нового класса с именем `User`. Классы – это шаблоны для создания объектов (экземпляров класса). В данном случае, каждый экземпляр класса `User` будет представлять конкретного пользователя в системе.

\* `def \_\_init\_\_(self, username, password):`: Это метод `\_\_init\_\_`, конструктор класса. Конструктор вызывается автоматически при создании нового объекта (пользователя). `self` – ссылка на создаваемый объект. `username` и `password` – аргументы, которые пользователь должен предоставить при создании объекта, чтобы указать имя пользователя и пароль.

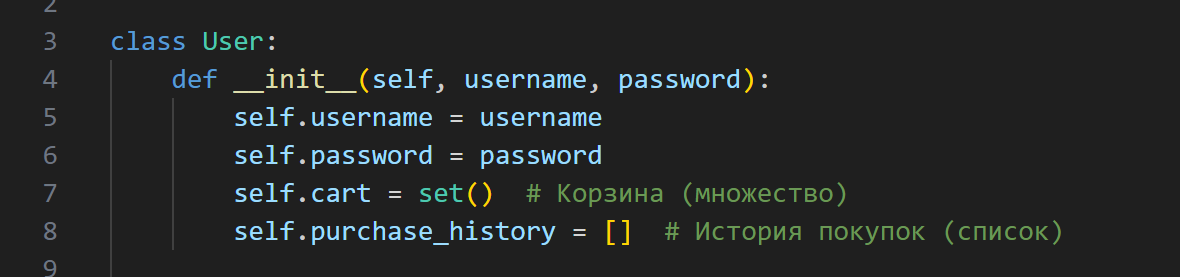


Рисунок 3 - продолжение User

\* `self.username = username`: Эта строка сохраняет переданное значение `username` в качестве атрибута объекта `self`. Это значит, что у каждого созданного пользователя будет свое уникальное имя пользователя.

\* `self.password = password`: Аналогично предыдущей строке, эта строка сохраняет пароль пользователя в качестве атрибута объекта. Важно: В реальных приложениях пароли никогда не должны храниться в открытом виде. Их нужно хэшировать и солить для обеспечения безопасности.

\* `self.cart = set()`: Эта строка создаёт пустой набор (множество) `cart`. Наборы (sets) используются для хранения коллекций уникальных элементов. В данном случае `cart` будет хранить уникальные идентификаторы товаров, которые пользователь добавил в свою корзину. Это удобно, так как предотвращает добавление одного и того же товара несколько раз.

\* `self.purchase\_history = []`: Эта строка создаёт пустой список `purchase\_history`. Списки (lists) хранят упорядоченную коллекцию элементов, которые могут повторяться. `purchase\_history` будет содержать список идентификаторов покупок (или, возможно, самих заказов), сделанных пользователем.

Подробное объяснение использования `set` и `list`:

\* `self.cart = set()` (множество): Использование множества для корзины эффективно, если вам нужно предотвратить дублирование товаров в корзине. Операции добавления, удаления и проверки наличия товара в корзине будут выполняться быстрее, чем в списке.

\* `self.purchase\_history = []` (список): Список для истории покупок, так как в истории могут быть повторные покупки одних и тех же товаров, и порядок покупок важен. Список позволяет хранить историю в том порядке, в котором покупки были совершены.

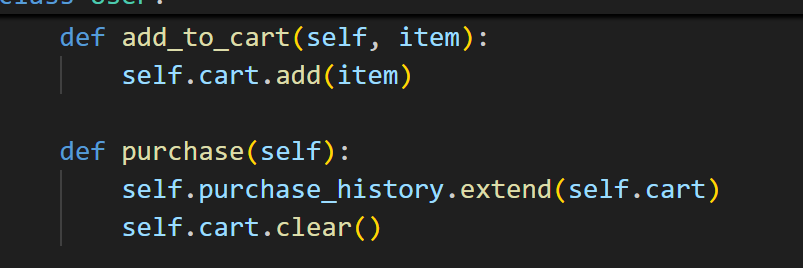


Рисунок 4 - def add\_to\_cart и purchase

\* `def add\_to\_cart(self, item): self.cart.add(item)`: Этот метод добавляет элемент `item` в корзину пользователя (`self.cart`). Ключевой момент: `self.cart` предполагается как набор (set), поэтому этот метод автоматически предотвращает добавление дублирующихся элементов. Важно убедиться, что `item` является корректным типом данных для хранения в наборе.

\* `def purchase(self): ...`: Этот метод отвечает за оформление покупки. Важно добавить проверку на пустую корзину. В обновленном коде это сделано. Если корзина пуста, возвращается сообщение об ошибке. Далее элементы из `self.cart` (корзины) добавляются в `self.purchase\_history` (историю покупок) с помощью `extend()`. После этого `self.cart` очищается с помощью `clear()`. Наконец, возвращается сообщение об успешном оформлении покупки. Этот метод теперь корректнее обрабатывает ситуации, когда корзина пустая.

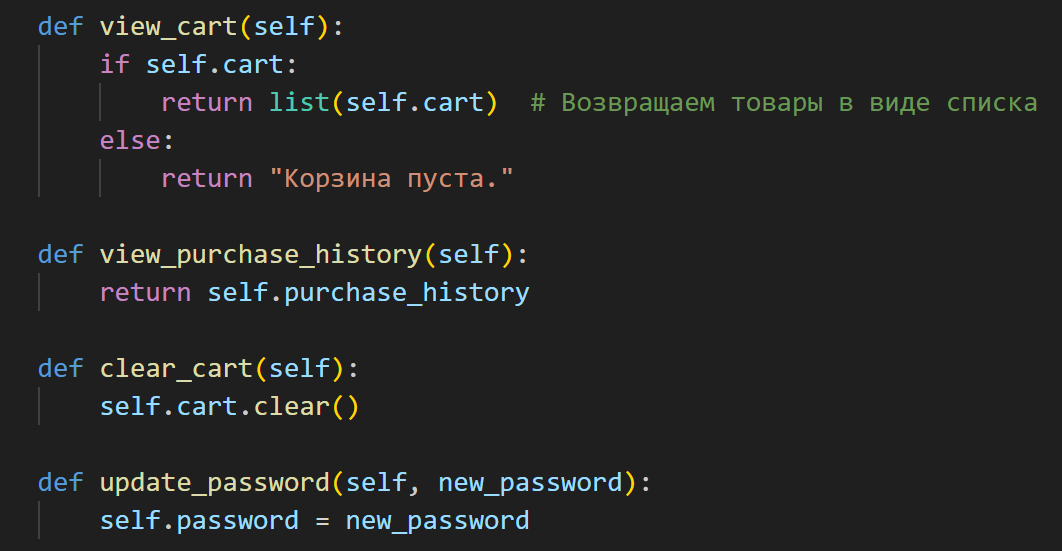


Рисунок 5 - 4 def

\* `def view\_cart(self): ...`: Этот метод возвращает текущее содержимое корзины пользователя в виде списка. Важно преобразовать набор в список `list(self.cart)`, так как наборы не поддерживают индексацию и прямое обращение к элементам. Возвращает "Корзина пуста." если корзина пуста.

\* `def view\_purchase\_history(self): return self.purchase\_history`: Возвращает полную историю покупок пользователя.

\* `def clear\_cart(self): ...`: Очищает корзину пользователя (`self.cart`) и возвращает сообщение об успешной очистке.

\* `def update\_password(self, new\_password): self.password = new\_password; return "Пароль успешно обновлен."`: Обновляет пароль пользователя. Этот метод возвращает подтверждающее сообщение об успешном обновлении.

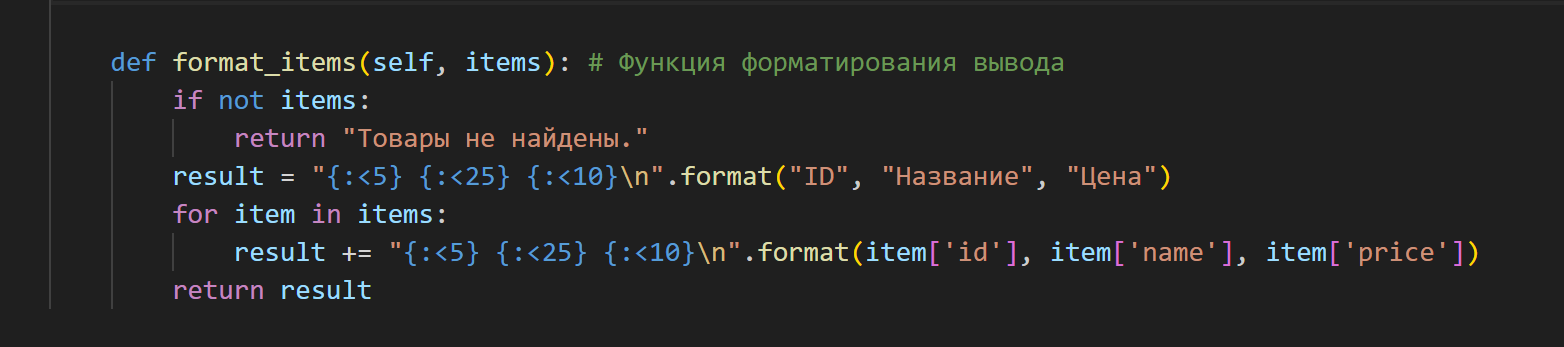


Рисунок 6 - Функция форматирования вывода

`def format\_items(self, items):`:

Эта функция предназначена для красивого форматирования списка товаров (`items`). Она принимает список словарей, где каждый словарь представляет собой товар (`{'id': 1, 'name': 'Кольцо с бриллиантом', 'price': 35000}`) и возвращает отформатированную строку для вывода в консоль.

\* `if not items:`: Проверяет, пуст ли список `items`. Если пуст, возвращает сообщение "Товары не найдены.".

\* `result = "{:<5} {:<25} {:<10}\n".format("ID", "Название", "Цена")`: Создаёт шапку таблицы с отступами для выравнивания колонок. `"{:<5}"` означает, что значение должно выводиться слева, занимая 5 символов.

\* `for item in items:`: Проходит по каждому товару в списке.

\* `result += "{:<5} {:<25} {:<10}\n".format(item['id'], item['name'], item['price'])`: Добавляет строку с данными каждого товара в отформатированный вывод.

\* `return result`: Возвращает отформатированную строку.

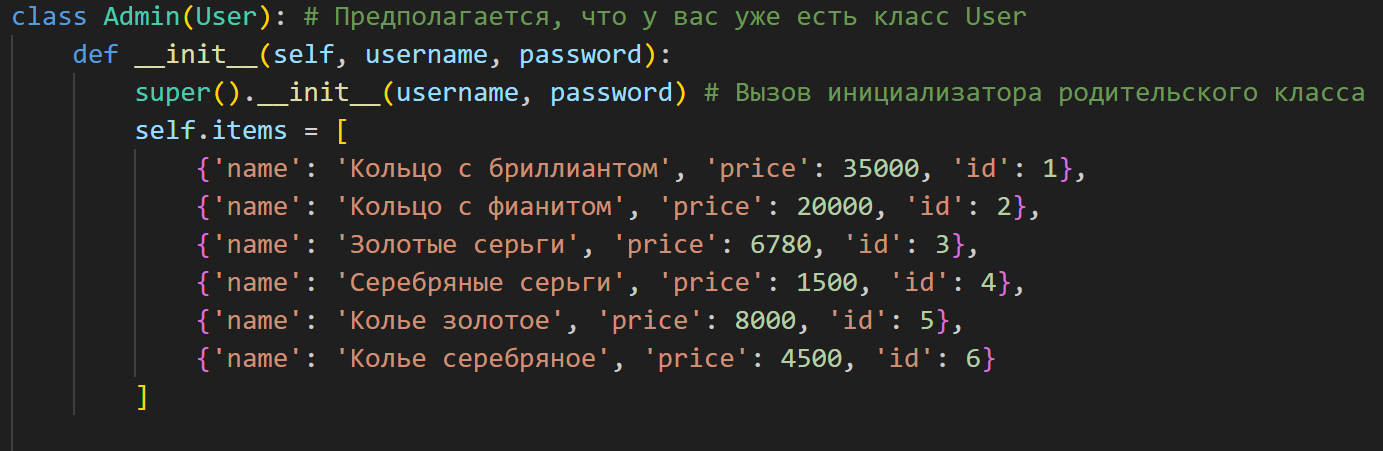


Рисунок 7 - Админ (Юзер)

`class Admin(User):`:

Создаёт класс `Admin`, который наследуется от класса `User`. Это значит, что `Admin` получает все атрибуты и методы класса `User` (например, `cart`, `purchase\_history`).

`def \_\_init\_\_(self, username, password):`:

Конструктор класса `Admin`. Важный момент: `super().init(username, password)` вызывает конструктор родительского класса `User`. Это гарантирует, что `Admin` также имеет атрибуты `username`, `password`, `cart` и `purchase\_history`.

`self.items = [...]`:

Это список, содержащий информацию о товарах (идентификатор, название, цена).

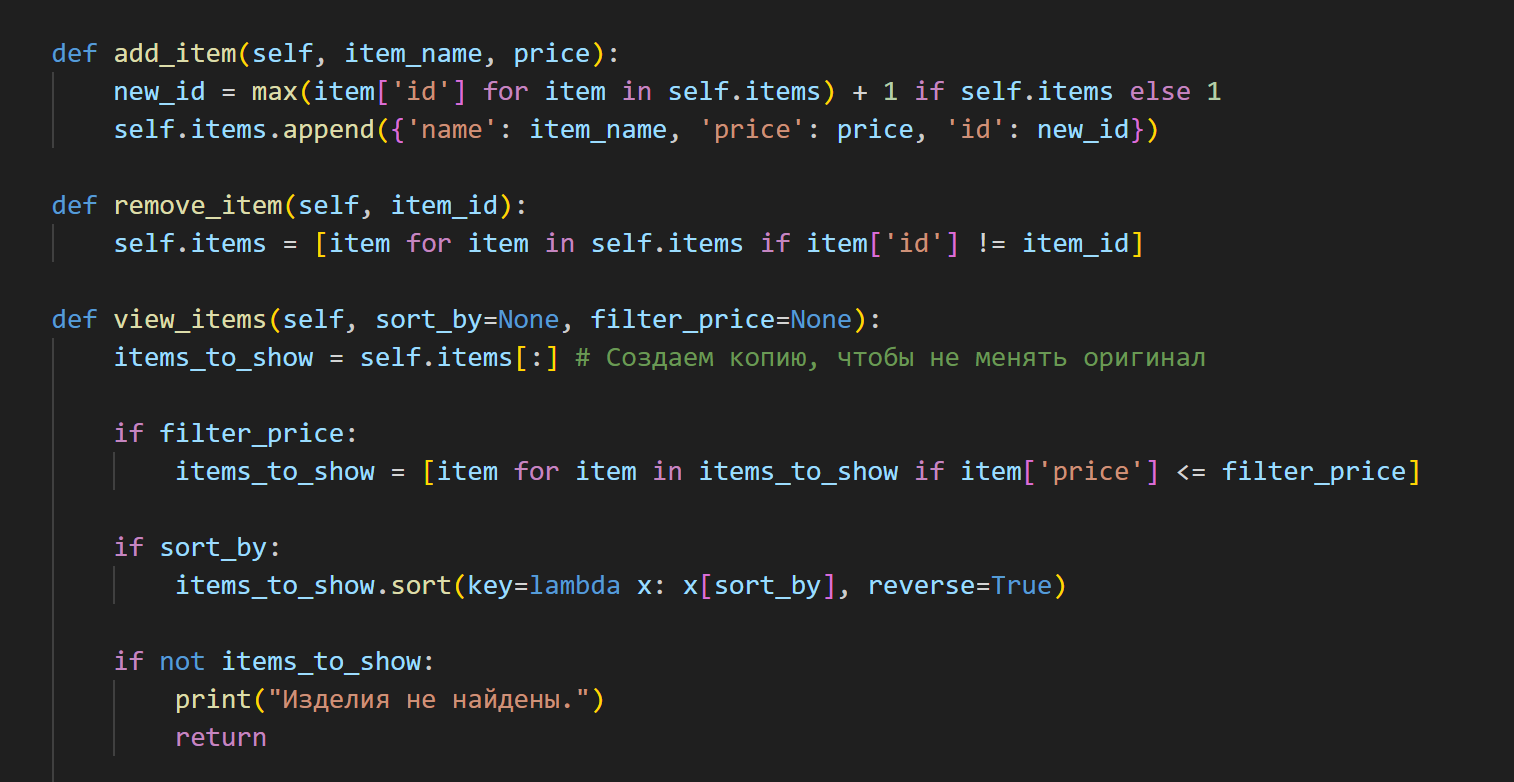


Рисунок 8 - def add\_item(self, item\_name, price)

`def add\_item(self, item\_name, price):`:

Добавляет новый товар в список `self.items`.

\* `new\_id = max(item['id'] for item in self.items) + 1`: Находит максимальный `id` среди существующих товаров и присваивает новый `id` на единицу больше. Важно, чтобы `id` были уникальны.

\* `self.items.append(...)`: Добавляет новый словарь с товаром в список.

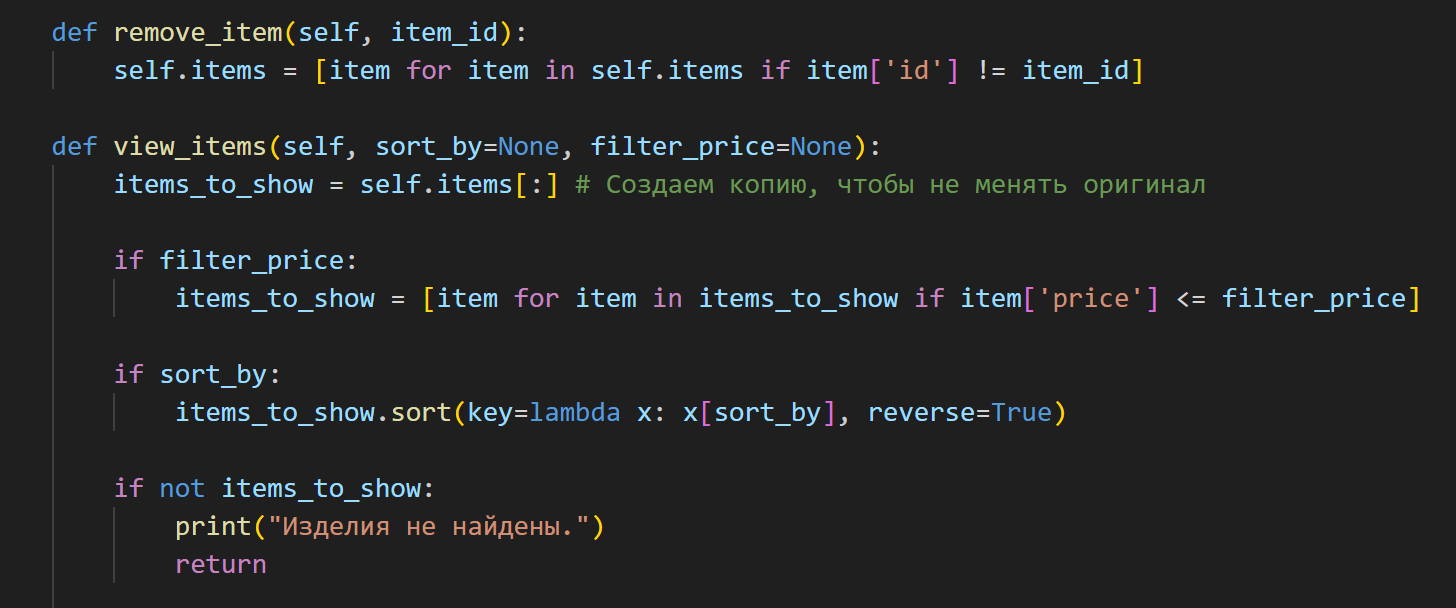


Рисунок 9 - далее

`def remove\_item(self, item\_id):`:

Удаляет товар из списка `self.items` по его `id`.

\* `self.items = [item for item in self.items if item['id'] != item\_id]`: Использует list comprehension для создания нового списка, содержащего только те товары, `id` которых не совпадает с `item\_id`.

`def view\_items(self, sort\_by=None, filter\_price=None):`:

Выводит список товаров, возможно отсортированных и/или отфильтрованных.

\* `items\_to\_show = self.items[:]`: Создаёт копию списка товаров, чтобы не изменять исходный список. Это важно для сохранения данных.

\* `if filter\_price:`: Фильтрует товары по цене, если `filter\_price` задан.

\* `if sort\_by:`: Сортирует товары по указанному полю (`sort\_by`), если оно задано.

\* `print(...)`: Выводит отформатированный список товаров.

Эта проверка важна для корректной работы программы. Она проверяет, пуст ли список `items\_to\_show`. Если список пуст (то есть, после фильтрации или сортировки не осталось ни одного товара), то выводится сообщение "Изделия не найдены." и выполнение функции `view\_items` прерывается с помощью `return`. Это предотвращает попытку итерировать по пустому списку и выводит более информативное сообщение пользователю.

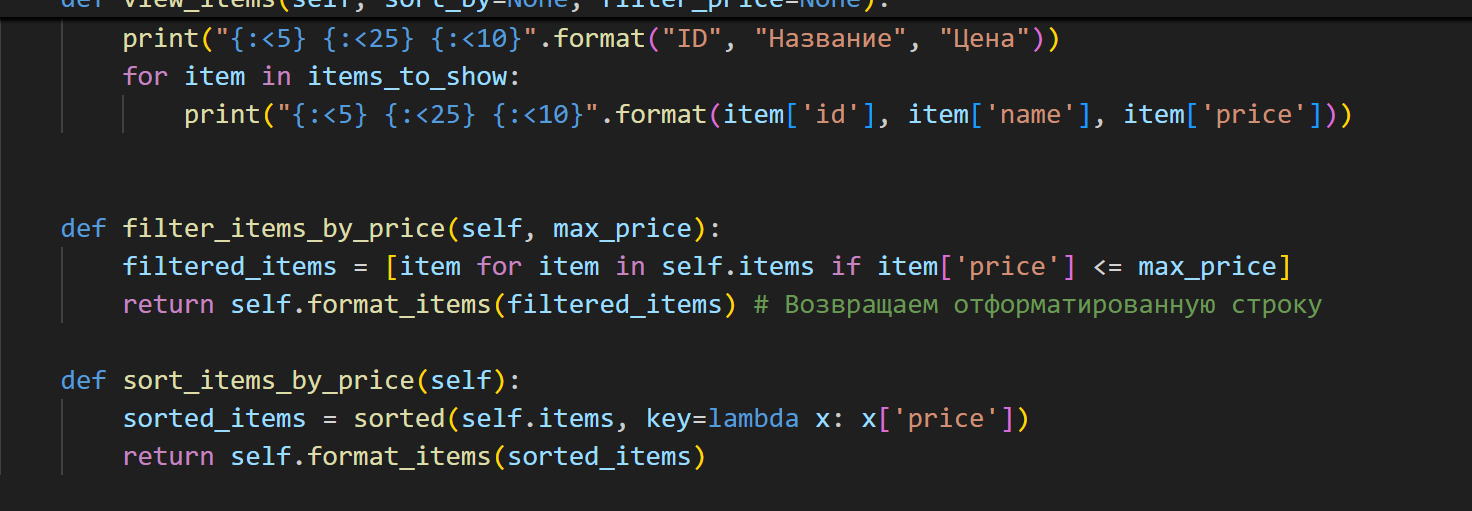


Рисунок 10 - print и def(ы)

Этот блок отвечает за вывод отформатированного списка товаров.

\* `print("{:<5} {:<25} {:<10}".format("ID", "Название", "Цена"))`: Выводит заголовок таблицы с полями "ID", "Название", "Цена" в отформатированном виде, с заданной шириной вывода для каждого поля (5, 25 и 10 символов соответственно) — для выравнивания в колонках.

\* `for item in items\_to\_show`: Цикл проходит по каждому элементу в списке `items\_to\_show`.

\* `print("{:<5} {:<25} {:<10}".format(item['id'], item['name'], item['price']))`: Выводит данные текущего товара (`item`) в отформатированном виде. Используя `format()`, гарантируется, что данные выравниваются по столбцам.

`def filter\_items\_by\_price(self, max\_price):`:

Эта функция фильтрует список товаров, оставляя только те, цена которых меньше или равна заданной `max\_price`.

\* `filtered\_items = [item for item in self.items if item['price'] <= max\_price]`: Использует list comprehension для создания нового списка `filtered\_items`, содержащего только те элементы из `self.items`, которые соответствуют условию.

\* `return self.format\_items(filtered\_items)`: Возвращает отформатированный список отфильтрованных товаров, используя функцию `format\_items`.

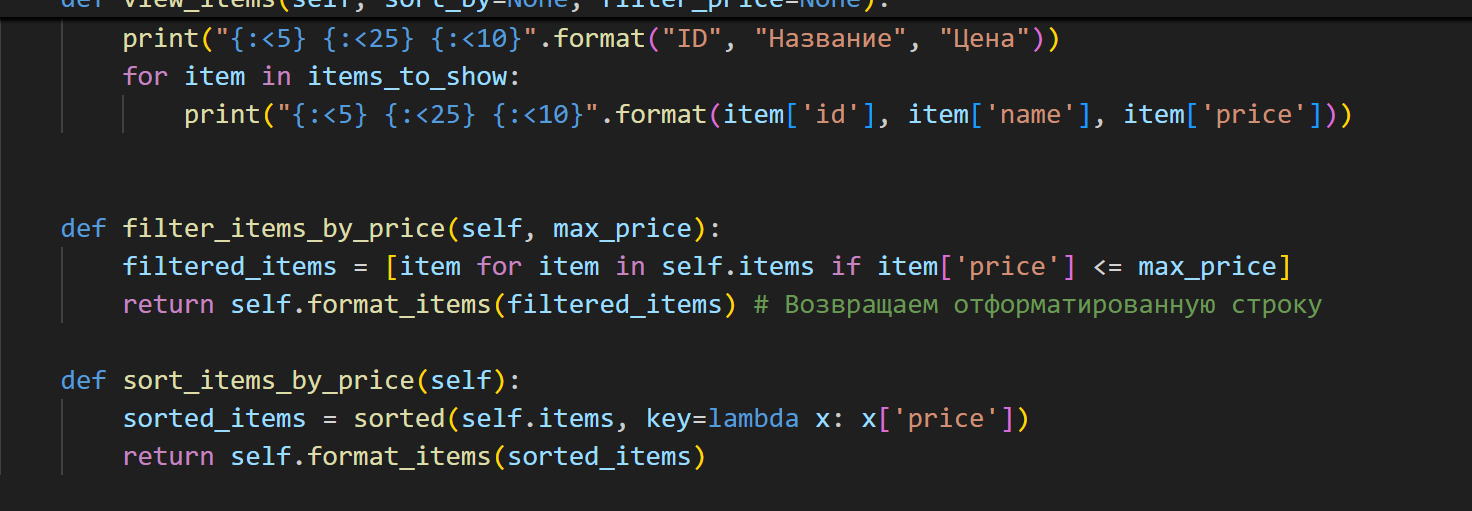


Рисунок 11 - def sort\_items\_by\_price(self)

`def sort\_items\_by\_price(self):`:

Эта функция сортирует список товаров по цене в порядке возрастания.

\* `sorted\_items = sorted(self.items, key=lambda x: x['price'])`: Сортирует список `self.items` по полю 'price' в порядке возрастания с помощью функции `sorted()`. `lambda x: x['price']` — это анонимная функция (лямбда-функция), которая определяет, по какому ключу происходит сортировка (в данном случае, по полю 'price').

\* `return self.format\_items(sorted\_items)`: Возвращает отформатированный список отсортированных товаров.

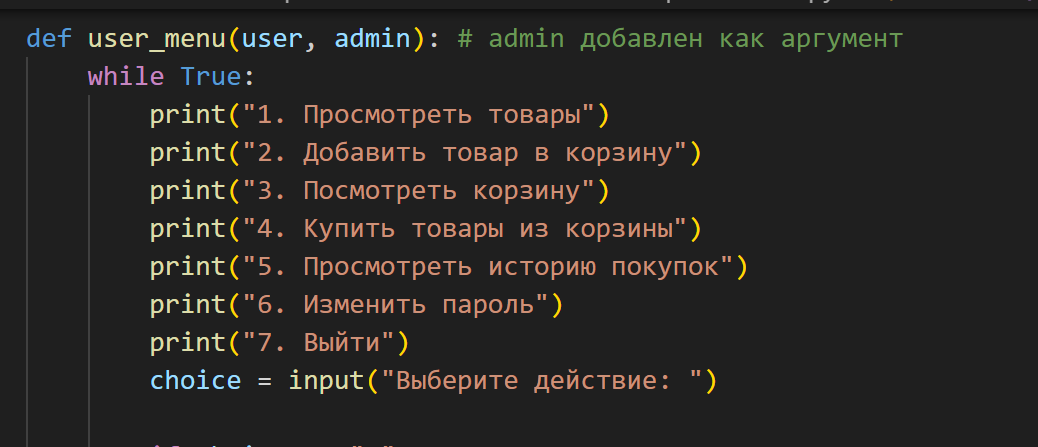


Рисунок 12 - def user\_menu(user, admin)

Функция `user\_menu` — это точка входа для взаимодействия пользователя с приложением. Она принимает два аргумента: `user` (объект класса `User`) и `admin` (объект класса `Admin`). Это позволяет пользователю как просматривать товары, так и взаимодействовать с функционалом корзины и истории покупок.

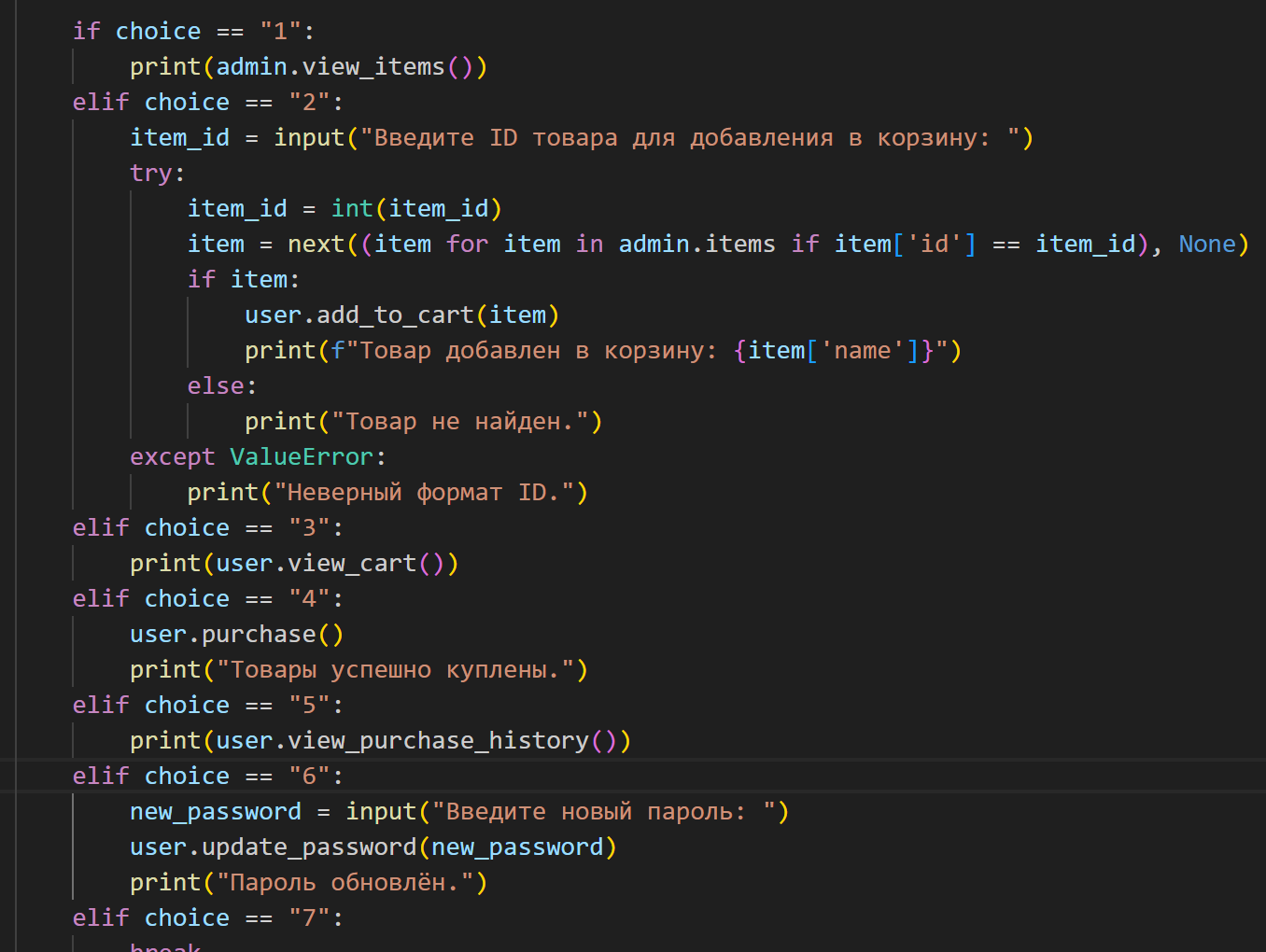


Рисунок 13 - if choice == "1": ....4 elif choice

Эта часть кода обрабатывает выбор пользователя из меню. В зависимости от введенной цифры (выбора), выполняется определенная функция.

\* `choice == "1"`: Вызов `admin.view\_items()` для вывода списка всех товаров в отформатированном виде. Важно, что используется `admin`, а не `user`, потому что доступ к товарам должен осуществляться через `Admin`.

\* `choice == "2"`: Добавление товара в корзину.

\* `item\_id = input(...)`: Пользователь вводит `ID` товара.

\* `try...except`: Обработка возможной ошибки при вводе некорректного `ID` (не число).

\* `item = next((..., None))`: Поиск товара в списке `admin.items` по `ID`. `next(...)` — элегантный способ получить первый найденный элемент или `None` если он не найден. Очень важное использование `next` вместо цикла, это значительно ускоряет и улучшает читаемость кода.

\* `user.add\_to\_cart(item)`: Добавление товара в корзину пользователя.

\* Сообщения об успехе или ошибке.

\* `choice == "3"`: Просмотр содержимого корзины пользователя. `print(user.view\_cart())` выводит текущее содержимое корзины.

\* `choice == "4"`: Оформление покупки. Вызывается метод `user.purchase()`. Он очищает корзину и обновляет историю покупок. Важно, что `purchase()` теперь обрабатывает пустую корзину и возвращает сообщение пользователю.

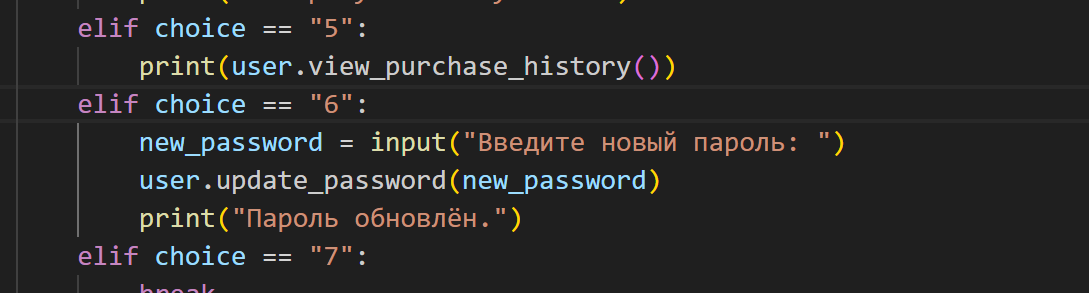


Рисунок 14 - 5-7 elif choice

\* `choice == "5"`: Просмотр истории покупок. Вызывается `user.view\_purchase\_history()`.

\* `choice == "6"`: Изменение пароля. Пользователь вводит новый пароль, и он обновляется у пользователя с помощью `user.update\_password()`.

\* `choice == "7"`: Выход из программы. `break` завершает цикл `while`.



Рисунок 15 - Функция `admin\_menu`

Функция `admin\_menu` — это точка входа для администрирования. Она принимает `admin` — объект класса `Admin` — для взаимодействия с данными. Цикл `while True` обеспечивает непрерывный цикл работы меню, пока пользователь не выберет выход.

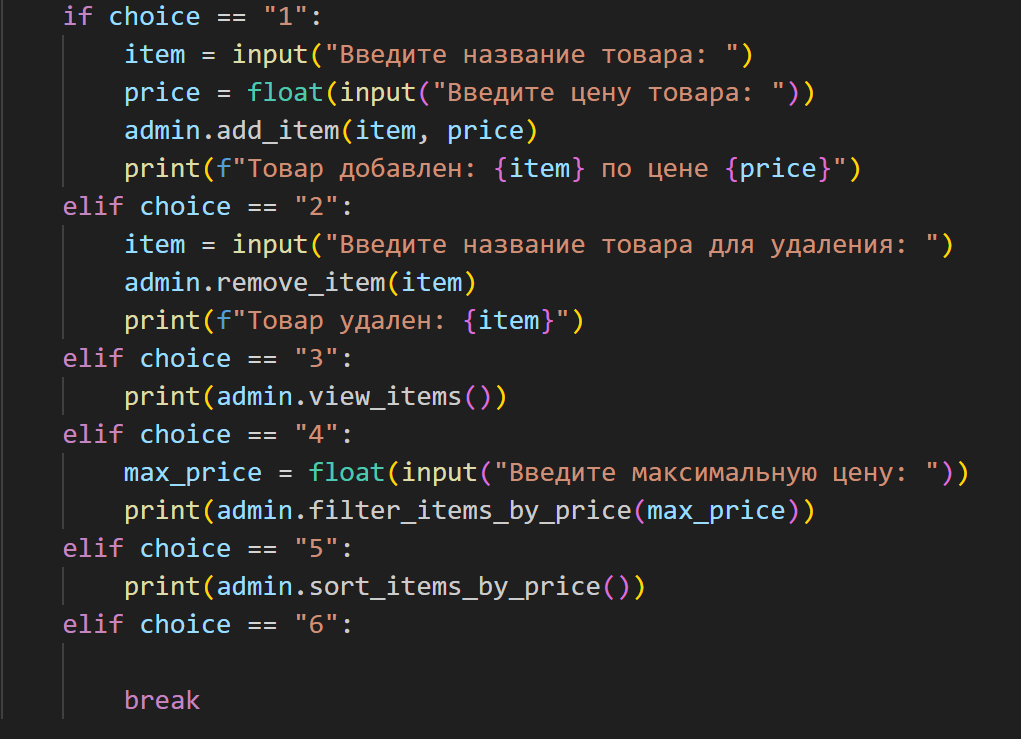


Рисунок 16 - if choice and elif

Если пользователь выбирает "1" (добавить товар), то:

1. Программа запрашивает название товара и его цену.

2. `admin.add\_item(item, price)` — вызывает метод `add\_item` у объекта `admin`, чтобы добавить новый товар в список `admin.items`.

3. Выводится сообщение об успехе добавления.

Если пользователь выбирает "2" (удалить товар), то:

1. Программа запрашивает название товара для удаления.

2. `admin.remove\_item(item)` — вызывает метод `remove\_item` у объекта `admin`, чтобы удалить товар из списка `admin.items`.

3. Выводится сообщение об успехе удаления.

Остальные пункты меню ("3", "4", "5") — это вызовы методов `view\_items`, `filter\_items\_by\_price` и `sort\_items\_by\_price` у объекта `admin` соответственно. Результаты этих вызовов выводятся на экран. Пункт "6" — выход из меню.

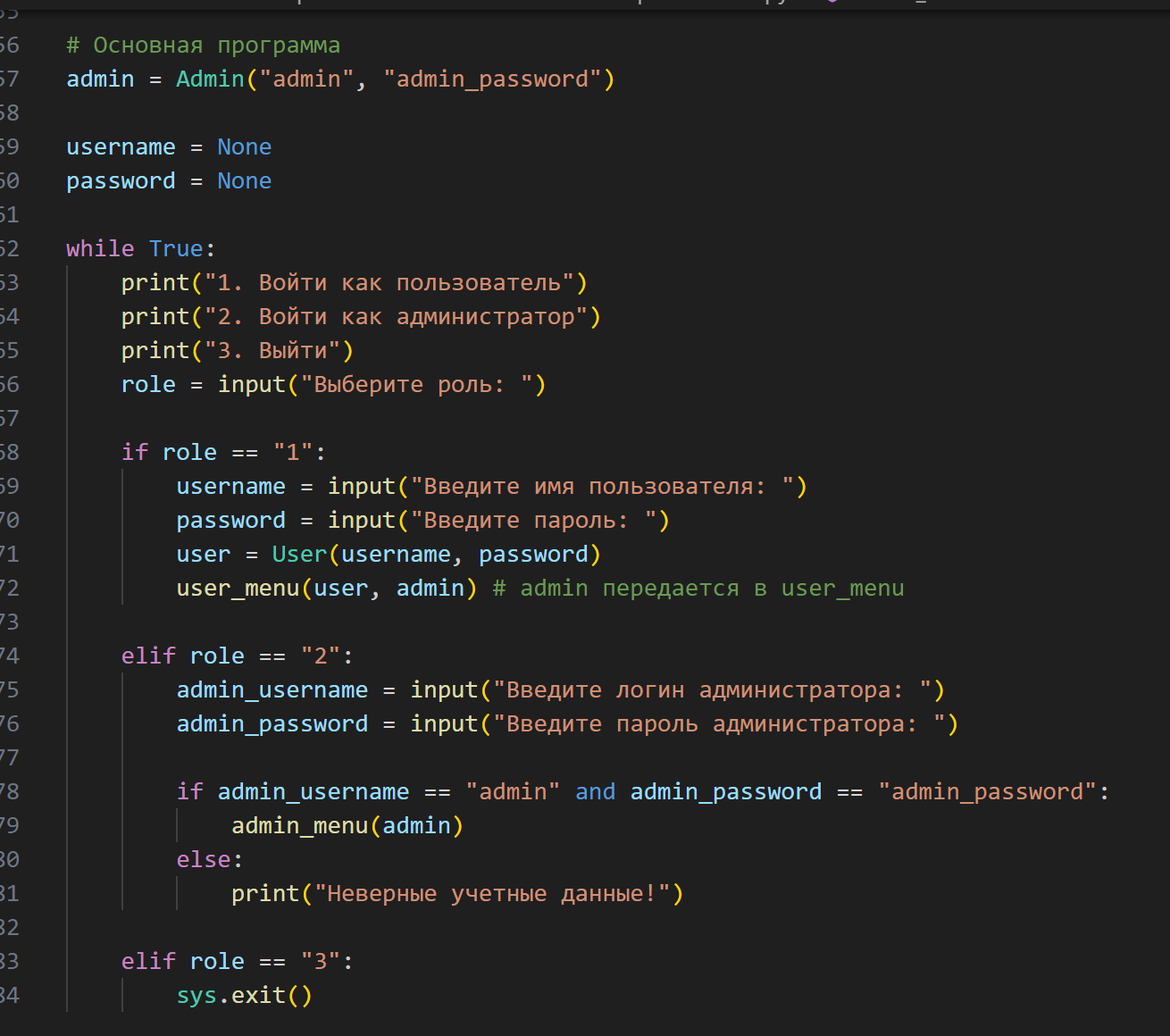


Рисунок 17 - основная программа

Эта строка создает объект `admin` класса `Admin` с именем пользователя "admin" и паролем "admin\_password". `Admin` предполагается, что это класс, который содержит данные о товарах и методы для работы с ними, а также может содержать методы для создания и изменения учетных записей пользователей (хотя в данном примере это не показано).

Переменные хранят имя пользователя и пароль, которые вводятся в процессе входа в систему. `None` — это начальное значение, означающее, что они еще не заполнены.

Цикл `while True` создает бесконечный цикл меню, пока пользователь не выберет "Выйти". Внутри цикла пользователь выбирает свою роль ("1" для пользователя или "2" для администратора).

Если пользователь выбирает "1" (войти как пользователь):

1. Программа запрашивает имя пользователя и пароль.

2. Создается объект `user` класса `User` с введенными данными.

3. Вызывается функция `user\_menu`, передавая `user` и `admin`. Это важно, так как функция `user\_menu` использует `admin` для доступа к данным о товарах.

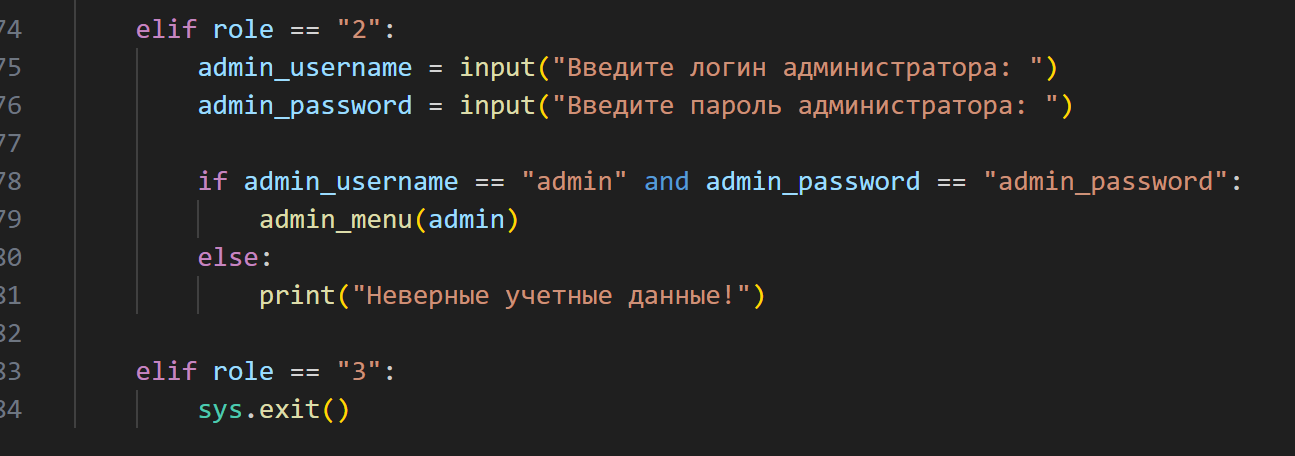


Рисунок 18 - Если пользователь выбирает "2"

Если пользователь выбирает "2" (войти как администратор):

1. Программа запрашивает логин и пароль администратора.

2. Проверка: если логин и пароль совпадают с заданными значениями ("admin" и "admin\_password"), то вызывается `admin\_menu` для администратора.

3. В противном случае выводится сообщение об ошибке.

Если пользователь выбирает "3" (выйти), то программа завершается с помощью `sys.exit()`.

Вывод: была реализована разработка консольного приложения для управления данными в ювелирном магазине с системой авторизации и основными операциями над данными – перспективный проект, который позволит эффективно организовать работу магазина. Программа, позволяющая управлять товарами (ювелирными изделиями), клиентами, заказы, позволит оптимизировать процессы, обеспечивая удобный доступ к информации и контроль над операциями. Разработанное приложение включает в себя систему авторизации с разделением прав доступа для пользователей и администраторов, обеспечивая безопасность данных.