Сборник заданий для семинарских занятий
по курсу
«Объектно-ориентированное программирование на Python»

Содержание

1 Общие сведения

Сборник содержит задания для семинарских занятий по курсу «Объектно-ориентированное программирование на Python» (32 часа).

Задачник находится в процессе наполнения и новые задания появляются перед проведением нового семинара.

Возможна сдача другого кода (например, выполненного в ходе проектной деятельности), еслои они полностью покрывают материал семинара.

2 Задания

2.1 Семинар «Правила формирования класса для программирования в IDE PyCharm. Отработка навыков создания простых классов и объектов класса» (2 очных часа)

В ходе работы создайте 5 классов с соответствующими методами, описанными в индивидуальном задании. Предполагается, что пользователь класса не имеет права обращаться к свойствам напрямую (соблюдая принцип инкапсуляции), а должен использовать методы. Важно: в задании не всегда указаны все необходимые методы и свойства, при необходимости вам надо самостоятельно их добавить. Продемонстрируйте работоспособность всех методов (из задания) посредством создания запускаемых файлов, где осуществляется вызов методов для разных ситуаций (без ручного ввода, но с выводом результатов в консоль). Каждый класс должен сохраняться в отдельном исходном файле. Необходимо соблюдать все стандартные требования к качеству кода (отступы, именования переменных, классов, методов, проверка корректности входных данных). Для каждого класса создайте отдельный запускаемый файл для проверки всех его методов (допускается использование других классов в этих тестах).

Все предлагаемые классы в заданиях упрощенные; для использования в production-окружении они требуют серьезной доработки. Суть задания — в отработке базовых навыков, а не в идеальном моделировании предложенных ситуаций.

Для сдачи работы будьте готовы пояснить или аналогично заданию модифицировать любую часть кода, а также ответить на вопросы:

1. Кратко опишите парадигму объектно-ориентированного программирования (ООП).
2. Что такое класс в парадигме ООП?
3. Что такое объект (экземпляр) в парадигме ООП?
4. Что обозначает свойство инкапсуляции в парадигме ООП?
5. Синтаксис классов в Python (в рамках выполненной работы), создание и работа с объектами в Python.

При выполнении задания предполагается самое простое базовое описание классов, соответствующее следующему примеру (вы можете использовать то, что вы ЗНАЕТЕ дополнительно, но это остается на ваше усмотрение):

Если вы нашли в задачнике ошибки, опечатки и другие недостатки, то вы можете сделать pull-request.

```
class Worker:
    def set_last_name(self, last_name):
        self.last_name = last_name

    def print_last_name(self):
        print (f"Фамилия: {self.last_name}")

    def get_last_name(self):
        return last_name

worker = Worker()
worker.set_last_name(self,"Иванов")
worker.print_last_name()
print(worker.get_last_name())
```

Срок сдачи работы (начала сдачи): следующее занятие после его выдачи. В последующие сроки оценка будет снижаться (при отсутствии оправдывающих документов).

1. **Описание ситуации:** Рассмотрим работу грузовой железнодорожной станции. На станции есть несколько путей, по которым поезда могут прибывать и отправляться. Каждый путь имеет свой номер и может вместить несколько поездов. Поезда формируются из вагонов, каждый из которых может перевозить разные грузы. Работники станции отвечают за диспетчерское управление маневровыми локомотивами, осмотр вагонов, выполнение погрузочно-разгрузочных работ, прием груза к перевозке, ремонт путей, обеспечение безопасности и т.п. Они используют радиостанции для связи друг с другом и для отслеживания положения поездов и передвижения вагонов.

   Создаваемые классы: ‘Путь‘, ‘Поезд‘, ‘Вагон‘, ‘Станция‘, ‘РаботникСтанции‘.

   Для классов реализовать следующие простые методы (ниже приведен не исчерпывающий список методов; для демонстрации работы классов вам потребуются дополнительные методы, позволяющие отследить состояние объектов), используя для хранения данных списки (‘[]‘) Python:

   (a) **Путь:** добавить поезд на путь, убрать поезд с пути, получить список поездов на конкретном пути.

   (b) **Поезд:** прицепить вагон к поезду, отцепить вагон от поезда, получить (распечатать) список вагонов в поезде, вывести информацию о грузе в поезде.

   (c) **Вагон:** добавить номер поезда, в который включался конкретный вагон, удалить номер поезда из истории, отобразить историю поездов для конкретного вагона.

   (d) **РаботникСтанции:** класс, представляющий отдельного работника на станции, имеющий идентификатор, информацию о персональной радиостанции, список закрепленных за ним поездов для осмотра, ФИО, должность.

   (e) **Станция:** добавить станционный путь, добавить поезд на станцию, нанять работника станции, вывести информацию о всех путях, поездах, работниках, удалить путь, удалить поезд, уволить работника.

2. **Описание ситуации:** Рассмотрим работу крупного логистического терминала для обработки грузовых автомобилей. На терминале есть несколько доков (рамп), куда фуры прибывают для проведения погрузочно-разгрузочных работ. Каждый док имеет свой номер и может одновременно обслуживать одну машину. Грузовики перевозят паллеты, каждая из которых содержит определенный товар. Сотрудники терминала отвечают за прием грузовиков, управление погрузочной техникой, проверку сопроводительных документов, приемку и отгрузку товара, а также техническое обслуживание доков. Они используют портативные рации для координации действий и отслеживания статуса обработки автомобилей.

   Создаваемые классы: ‘Док‘, ‘Грузовик‘, ‘Паллета‘, ‘Терминал‘, ‘Сотрудник‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **Док:** занять док конкретным грузовиком, освободить док, получить информацию о грузовике, который сейчас находится на доке.

   (b) **Грузовик:** добавить паллету в грузовик, выгрузить паллету из грузовика, получить (распечатать) список паллет в грузовике, вывести информацию о товарах в грузовике.

   (c) **Паллета:** добавить номер грузовика, в который загружалась конкретная паллета, удалить номер грузовика из истории, отобразить историю перевозок (номера грузовиков) для конкретной паллеты.

(d) **Сотрудник:** класс, представляющий отдельного сотрудника терминала, имеющий идентификатор, номер рации, список доков, за которые он отвечает, ФИО, должность.

(e) **Терминал:** добавить новый док на терминале, зарегистрировать прибытие грузовика, нанять нового сотрудника, вывести список всех доков, грузовиков на территории, сотрудников, удалить док, удалить грузовик, уволить сотрудника.

3. **Описание ситуации:** Рассмотрим работу аэропорта. В аэропорту есть несколько взлетно-посадочных полос (ВПП), которые принимают и отправляют рейсы. Каждая ВПП имеет свой номер, длину и статус доступности. Самолеты перевозят пассажиров и их ручную кладь, размещенную в салоне. Авиадиспетчеры управляют движением самолетов, назначают полосы для взлета и посадки, следят за воздушной обстановкой и координируют действия с помощью радиосвязи.

   Создаваемые классы: ‘ВПП‘, ‘Самолет‘, ‘Пассажир‘, ‘Аэропорт‘, ‘Авиадиспетчер‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **ВПП:** занять полосу для взлета/посадки, освободить полосу, получить список рейсов, использовавших полосу.

   (b) **Самолет:** добавить пассажира на борт (включая вес его ручной клади), высадить пассажира, получить (распечатать) список пассажиров на борту, рассчитать общий вес ручной клади.

   (c) **Пассажир:** добавить рейс в историю перелетов пассажира, удалить рейс из истории (ошибка бронирования), отобразить всю историю перелетов.

   (d) **Авиадиспетчер:** класс, представляющий диспетчера, имеющий идентификатор, рабочую частоту, график работы (список интервалов времени в сутках), ФИО.

   (e) **Аэропорт:** добавить новую ВПП, зарегистрировать прибытие самолета, нанять диспетчера, вывести список всех ВПП, самолетов в аэропорту, диспетчеров, удалить ВПП (на ремонт), списать самолет, уволить диспетчера.

4. **Описание ситуации:** Рассмотрим работу речного порта. В порту есть несколько причалов для швартовки грузовых барж и буксиров. Каждый причал имеет уникальный номер и максимальную глубину, определяющую осадку судов, которые могут к нему подойти. Баржи перевозят контейнеры с различными грузами. Их характеризуют вес судна, максимальная грузоподъемность и осадка (как без груза, так и с максимальным грузом). Портовые рабочие отвечают за швартовку судов, управление портовыми кранами для погрузки/разгрузки контейнеров, оформление документов и поддержание порядка на территории.

   Создаваемые классы: ‘Причал‘, ‘Баржа‘, ‘Контейнер‘, ‘Порт‘, ‘ПортовыйРабочий‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **Причал:** пришвартовать баржу к причалу, отшвартовать баржу, получить список барж, находящихся у причала.

   (b) **Баржа:** загрузить контейнер на баржу (с указанием веса контейнера), разгрузить контейнер с баржи, получить (распечатать) список контейнеров на барже, рассчитать текущую осадку судна (предполагается линейная зависимость осадки от суммарного веса груза и баржи).

(c) **Контейнер:** добавить номер баржи, на которую погрузили контейнер, удалить номер баржи, отобразить историю перемещений контейнера между баржами.

(d) **ПортовыйРабочий:** класс, представляющий рабочего, имеющий идентификатор, допуск к работе с краном, список закрепленных причалов, ФИО, должность.

(e) **Порт:** ввести новый причал в эксплуатацию, принять баржу в акваторию порта, принять на работу рабочего, вывести список причалов, барж в акватории, рабочих, списать причал, отправить баржу, уволить рабочего.

5. **Описание ситуации:** Рассмотрим работу автобусного парка. В парке есть несколько маршрутов, которые обслуживаются автобусами. Каждый маршрут имеет номер и список остановок. Автобусы имеют государственный номер, количество мест и текущий пробег. Водители закреплены за автобусами и маршрутами. Диспетчеры автопарка составляют расписание, следят за выходами автобусов на линию, учетом пробега и техническим состоянием.

   Создаваемые классы: ‘Маршрут‘, ‘Автобус‘, ‘Остановка‘, ‘Автопарк‘, ‘Водитель‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **Маршрут:** добавить остановку в маршрут, удалить остановку из маршрута, получить список всех остановок на маршруте.

   (b) **Автобус:** назначить автобус на маршрут, снять с маршрута, увеличить пробег на заданное значение, получить текущий пробег.

   (c) **Остановка:** добавить маршрут, проходящий через остановку, удалить маршрут, отобразить список всех маршрутов, проходящих через данную остановку.

   (d) **Водитель:** класс, представляющий водителя, имеющий идентификатор, права категории, закрепленный автобус, ФИО, график работы.

   (e) **Автопарк:** добавить новый маршрут, приобрести новый автобус, принять на работу водителя, вывести список маршрутов, автобусов (с указанием их состояния), водителей, списать автобус, уволить водителя.

6. **Описание ситуации:** Рассмотрим работу метрополитена. В метро есть линии, состоящие из станций и тоннелей между ними. Составы из вагонов перемещаются по линиям. Каждая станция имеет название и может быть точкой пересадки на другие линии. Машинисты управляют поездами. Дежурные по станции следят за порядком на платформах и работой оборудования. Управление метрополитеном координирует движение составов.

   Создаваемые классы: ‘ЛинияМетро‘, ‘ПоездМетро‘, ‘Станция‘, ‘УправлениеМетрополитеном‘, ‘Машинист‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **ЛинияМетро:** добавить станцию на линию, получить список станций на линии, получить список поездов на линии.

   (b) **ПоездМетро:** добавить вагон в состав, отцепить вагон, назначить машиниста на поезд.

   (c) **Станция:** добавить линию, проходящую через станцию (для моделирования пересадочных узлов), получить список линий на станции.

(d) **Машинист:** класс, представляющий машиниста, имеющий идентификатор, допуск к управлению, закрепленный поезд, ФИО, стаж.

(e) **УправлениеМетрополитеном:** открыть новую линию, ввести новый поезд в эксплуатацию, принять на работу машиниста, вывести список линий, поездов (в депо и на линиях), машинистов, закрыть линию на техобслуживание, списать поезд, вывести полную схему метро (в текстовом виде).

7. **Описание ситуации:** Рассмотрим работу службы доставки пиццы. В службе есть несколько филиалов. Каждый филиал обслуживает определенный район и имеет курьеров. Заказы формируются из позиций меню. Курьеры используют скутеры для доставки. Менеджеры филиалов принимают заказы, назначают курьеров и следят за выполнением заказов.

   Создаваемые классы: ‘Филиал‘, ‘Заказ‘, ‘Курьер‘, ‘Скутер‘, ‘Менеджер‘.

   Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

   (a) **Филиал:** добавить курьера в филиал, уволить курьера, получить список активных заказов филиала.

   (b) **Заказ:** добавить позицию в заказ (название + цена), удалить позицию, рассчитать стоимость заказа, изменить статус заказа (принят, готовится, в пути, доставлен).

   (c) **Курьер:** назначить заказ курьеру, завершить доставку заказа, получить список доставленных заказов за смену, закрепить скутер за курьером.

   (d) **Менеджер:** класс, представляющий менеджера, имеющий идентификатор, закрепленный филиал, ФИО, смену.

   (e) **Скутер:** отправить на зарядку, вернуть в строй, увеличить пробег, получить текущий пробег.

   Описание ситуации: Рассмотрим работу трамвайного депо. В депо есть несколько маршрутов, обслуживаемых трамвайными вагонами. Каждый трамвайный вагон имеет бортовой номер, вместимость и текущий пробег. Маршруты состоят из остановок и имеют определенный график движения. Водители трамваев закреплены за конкретными вагонами и маршрутами. Диспетчеры управляют выпуском трамваев на линию и ведут учет технического состояния.

   Создаваемые классы: Маршрут, Трамвай, Остановка, Депо, Водитель.

   Для классов реализовать следующие простые методы, используя для хранения данных списки ([]) Python:

   (a) **Маршрут:** добавить остановку в маршрут, удалить остановку из маршрута, получить список всех остановок на маршруте.

   (b) **Трамвай:** назначить трамвай на маршрут, снять с маршрута, увеличить пробег на заданное значение, получить текущий пробег.

   (c) **Остановка:** добавить маршрут, проходящий через остановку, удалить маршрут, отобразить список всех маршрутов, проходящих через данную остановку.

   (d) **Водитель:** класс, представляющий водителя, имеющий идентификатор, права категории, закрепленный трамвай, ФИО, график работы.

(e) **Депо:** добавить новый маршрут, принять новый трамвай в депо, принять на работу водителя, выполнить вывод списка маршрутов, трамваев (с указанием их состояния), водителей, списать трамвай, уволить водителя.

8. **Описание ситуации:** Рассмотрим работу морского порта для приёма пассажирских паромов. В порту есть несколько причалов, каждый из которых обслуживает один паром за раз. Паромы перевозят пассажиров и автомобили. Пассажиры покупают билеты, автомобили записываются в список грузовой палубы. Сотрудники порта координируют погрузку, проверку билетов и безопасность. **Создаваемые классы:** Причал, Паром, Пассажир, Автомобиль, СотрудникПорта.

   (a) **Причал:** пришвартовать паром, освободить причал, получить информацию о пароме у причала.

   (b) **Паром:** добавить пассажира, добавить автомобиль, высадить пассажира, выгрузить автомобиль.

   (c) **Пассажир:** добавить рейс в историю поездок, удалить рейс из истории, вывести историю поездок.

   (d) **Автомобиль:** зарегистрировать номер парома, удалить номер парома, вывести историю перевозок.

   (e) **СотрудникПорта:** идентификатор, должность, ФИО, список закреплённых причалов.

9. **Описание ситуации:** Рассмотрим работу пригородной электрички. В системе есть станции, между которыми курсируют электрички. У каждой электрички есть номер, список вагонов и машинист. Пассажиры покупают билеты и занимают места в вагонах. Диспетчеры контролируют движение электричек. **Создаваемые классы:** Станция, Электричка, Вагон, Пассажир, Диспетчер.

   (a) **Станция:** принять электричку, отправить электричку, вывести список электричек на станции.

   (b) **Электричка:** добавить вагон, отцепить вагон, получить список вагонов.

   (c) **Вагон:** посадить пассажира, высадить пассажира, вывести список пассажиров.

   (d) **Пассажир:** добавить поездку в историю, удалить поездку, показать историю поездок.

   (e) **Диспетчер:** идентификатор, ФИО, рабочая смена, список контролируемых электричек.

10. **Описание ситуации:** Рассмотрим работу таксопарка. В таксопарке есть автомобили, водители и диспетчеры. Автомобиль закрепляется за водителем. Диспетчеры принимают заказы и назначают их водителям. Пассажиры совершают поездки. **Создаваемые классы:** Таксопарк, Автомобиль, Водитель, Заказ, Диспетчер.

    (a) **Таксопарк:** добавить автомобиль, принять водителя, вывести список машин и водителей, уволить водителя.

    (b) **Автомобиль:** назначить водителя, снять водителя, увеличить пробег, получить пробег.

    (c) **Водитель:** назначить заказ, завершить заказ, вывести список выполненных заказов.

(d) **Заказ:** назначить пассажира, завершить поездку, вывести информацию о заказе.

(e) **Диспетчер:** идентификатор, ФИО, список назначенных заказов.

11. **Описание ситуации:** Рассмотрим работу грузового аэропорта. Самолёты перевозят контейнеры. В аэропорту есть ангары для хранения самолётов и площадки для погрузки. Работники аэропорта координируют загрузку и выгрузку контейнеров. **Создаваемые классы:** Самолёт, Контейнер, Ангар, РаботникАэропорта, Аэропорт.

    (a) **Самолёт:** загрузить контейнер, выгрузить контейнер, вывести список контейнеров.

    (b) **Контейнер:** добавить номер самолёта, удалить номер самолёта, вывести историю перевозок.

    (c) **Ангар:** принять самолёт, вывести список самолётов, освободить ангар.

    (d) **РаботникАэропорта:** идентификатор, ФИО, должность, список самолётов в обслуживании.

    (e) **Аэропорт:** принять самолёт, убрать самолёт, принять раотника, уволить работника, вывести список самолётов и работников.

12. **Описание ситуации:** Рассмотрим работу велопроката. В прокате есть велосипеды, станции для их хранения, клиенты и сотрудники. Клиенты арендуют велосипеды и возвращают их на станцию. **Создаваемые классы:** Велосипед, СтанцияПроката, Клиент, Сотрудник, Прокат.

    (a) **Велосипед:** выдать в аренду, вернуть на станцию, получить пробег.

    (b) **СтанцияПроката:** добавить велосипед, убрать велосипед, вывести список велосипедов.

    (c) **Клиент:** арендовать велосипед, вернуть велосипед, вывести историю аренд.

    (d) **Сотрудник:** идентификатор, ФИО, должность, список закреплённых станций.

    (e) **Прокат:** добавить станцию, демонтировать станцию, вывести список станций и велосипедов, уволить сотрудника, нанять сотрудника, вывести список сотрудников.

13. **Описание ситуации:** Рассмотрим работу речных теплоходов. У каждого теплохода есть рейсы и список пассажиров. Пассажиры покупают билеты. Работники пристани обслуживают теплоходы. **Создаваемые классы:** Теплоход, Рейс, Пассажир, Пристань, РаботникПристани.

    (a) **Теплоход:** добавить рейс, убрать рейс, вывести список рейсов.

    (b) **Рейс:** добавить пассажира, удалить пассажира, вывести список пассажиров.

    (c) **Пассажир:** добавить рейс в историю, удалить рейс, вывести историю.

    (d) **Пристань:** принять теплоход, отправить теплоход, вывести список теплоходов.

    (e) **РаботникПристани:** идентификатор, ФИО, должность, закреплённые рейсы.

14. **Описание ситуации:** Рассмотрим работу каршеринга. В системе есть автомобили, клиенты и диспетчеры. Автомобили бронируются клиентами и возвращаются после поездки. Диспетчеры контролируют состояние машин. **Создаваемые классы:** Автомобиль, Клиент, Диспетчер, Заказ, Каршеринг.

(a) **Автомобиль:** выдать клиенту, вернуть, увеличить пробег, вывести пробег.

(b) **Клиент:** арендовать автомобиль, завершить аренду, вывести историю аренд.

(c) **Диспетчер:** идентификатор, ФИО, список автомобилей под контролем.

(d) **Заказ:** назначить автомобиль, завершить поездку, вывести данные заказа.

(e) **Каршеринг:** добавить автомобиль, списать автомобиль, добавить клиента, удалить клиента, добавить диспетчера, удалить диспетчера, вывести список клиентов, диспетчеров и машин.

15. **Описание ситуации:** Рассмотрим работу железнодорожного музея. В музее есть экспонаты (локомотивы и вагоны), экскурсии и экскурсоводы. Посетители записываются на экскурсии. **Создаваемые классы:** Экспонат, Экскурсия, Экскурсовод, Посетитель, Музей.

    (a) **Экспонат:** добавить к экскурсии, убрать, вывести список экскурсий.

    (b) **Экскурсия:** записать посетителя, удалить, вывести список посетителей.

    (c) **Экскурсовод:** идентификатор, ФИО, список экскурсий.

    (d) **Посетитель:** записаться на экскурсию, отменить запись, вывести историю.

    (e) **Музей:** добавить экспонат, списать экспонат, добавить экскурсовода, уволить экскурсовода, провести экскурсию, вывести список всех экскурсий и экскурсоводов.

16. **Описание ситуации:** Рассмотрим работу автозаправочной станции. На станции есть топливо, колонки и операторы. Автомобили приезжают заправляться. **Создаваемые классы:** Колонка, Автомобиль, Оператор, Топливо, АЗС.

    (a) **Колонка:** заправить автомобиль, освободить колонку, вывести статус.

    (b) **Автомобиль:** получить заправку, вывести историю заправок.

    (c) **Оператор:** идентификатор, ФИО, список закреплённых колонок.

    (d) **Топливо:** уменьшить количество, увеличить количество, вывести остаток.

    (e) **АЗС:** добавить колонку, нанять оператора, уволить оператора, демонтировать колонку, вывести список машин, операторов и колонок.

17. **Описание ситуации:** Рассмотрим работу сортировочного центра курьерской службы. В центре есть зоны обработки посылок, конвейерные линии и сотрудники. Каждая посылка имеет трек-номер и проходит через несколько этапов обработки. Сотрудники сканируют посылки, сортируют их по направлениям и загружают в транспортировочные контейнеры. Менеджеры контролируют процесс сортировки и работу оборудования.

    Создаваемые классы: ‘ЗонаОбработки‘, ‘Посылка‘, ‘Конвейер‘, ‘СотрудникЦентра‘, ‘СортировочныйЦентр‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **ЗонаОбработки:** добавить посылку в зону, удалить посылку из зоны, получить список посылок в зоне.

    (b) **Посылка:** добавить статус обработки (принята, сортируется, отправлена), удалить ошибочный статус, отобразить историю статусов обработки.

(c) **Конвейер:** запустить конвейерную ленту, остановить конвейер, добавить посылку на конвейер, снять посылку с конвейера.

(d) **СотрудникЦентра:** класс, представляющий сотрудника, имеющий идентификатор, смену, список закрепленных зон обработки, ФИО, должность.

(e) **СортировочныйЦентр:** добавить новую зону обработки, ввести в эксплуатацию конвейер, нанять сотрудника, вывести список всех зон, конвейеров, сотрудников, удалить зону, вывести из эксплуатации конвейер, уволить сотрудника.

18. **Описание ситуации:** Рассмотрим работу диспетчерской службы городского пассажирского транспорта. Диспетчеры отслеживают движение автобусов, троллейбусов и трамваев на маршрутах, регулируют интервалы движения, фиксируют отклонения от графика. Транспортные средства оснащены GPS-трекерами для передачи местоположения.

    Создаваемые классы: ‘Маршрут‘, ‘ТранспортноеСредство‘, ‘Диспетчер‘, ‘Остановка‘, ‘ДиспетчерскаяСлужба‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **Маршрут:** добавить транспортное средство на маршрут, снять с маршрута, получить список транспорта на маршруте.

    (b) **ТранспортноеСредство:** обновить местоположение (координаты), получить текущее местоположение, добавить информацию о задержке/опережении графика.

    (c) **Диспетчер:** класс, представляющий диспетчера, имеющий идентификатор, смену, список контролируемых маршрутов, ФИО.

    (d) **Остановка:** добавить маршрут, проходящий через остановку, удалить маршрут, получить список маршрутов на остановке.

    (e) **ДиспетчерскаяСлужба:** добавить новый маршрут, зарегистрировать транспортное средство, нанять диспетчера, вывести информацию о всех маршрутах, транспорте, диспетчерах, удалить маршрут, списать транспорт, уволить диспетчера.

19. **Описание ситуации:** Рассмотрим работу центра технического обслуживания городского транспорта. В центре есть ремонтные зоны для разных видов транспорта, запасы запчастей и бригады механиков. Транспортные средства проходят плановое ТО и внеплановый ремонт.

    Создаваемые классы: ‘РемонтнаяЗона‘, ‘ТранспортноеСредство‘, ‘Запчасть‘, ‘Механик‘, ‘ЦентрТехОбслуживания‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **РемонтнаяЗона:** поставить транспорт на ремонт, завершить ремонт, получить список транспорта в ремонте.

    (b) **ТранспортноеСредство:** добавить запись о ремонте (дата, вид работ), удалить ошибочную запись, отобразить историю ремонтов.

    (c) **Запчасть:** уменьшить количество на складе, увеличить количество, получить текущий остаток.

(d) **Механик:** класс, представляющий механика, имеющий идентификатор, квалификацию, список закрепленных ремонтных зон, ФИО.

(e) **ЦентрТехОбслуживания:** добавить ремонтную зону, закупить запчасти, нанять механика, вывести информацию о зонах, запчастях, механиках, удалить зону, уволить механика.

20. **Описание ситуации:** Рассмотрим работу логистического центра междугородных автобусных перевозок. Автобусы совершают рейсы между городами по определенным маршрутам, перевозя пассажиров и их багаж. Диспетчеры формируют расписание, продают билеты и контролируют отправление автобусов.

    Создаваемые классы: ‘Автобус‘, ‘Маршрут‘, ‘Пассажир‘, ‘Диспетчер‘, ‘ЛогистическийЦентр‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **Автобус:** назначить на маршрут, снять с маршрута, добавить пассажира, высадить пассажира, получить список пассажиров.

    (b) **Маршрут:** добавить город в маршрут, удалить город, получить список всех городов на маршруте.

    (c) **Пассажир:** купить билет (добавить маршрут в историю), сдать билет (удалить маршрут), показать историю поездок.

    (d) **Диспетчер:** класс, представляющий диспетчера, имеющий идентификатор, список закрепленных маршрутов, ФИО, график работы.

    (e) **ЛогистическийЦентр:** добавить автобус в парк, добавить маршрут, нанять диспетчера, вывести список автобусов, маршрутов и диспетчеров, списать автобус, уволить диспетчера.

21. **Описание ситуации:** Рассмотрим работу центра управления интеллектуальной транспортной системой города. Система включает в себя управление светофорами, камеры видеонаблюдения, датчики транспортного потока. Операторы следят за дорожной ситуацией и оперативно реагируют на инциденты.

    Создаваемые классы: ‘Перекресток‘, ‘Светофор‘, ‘КамераНаблюдения‘, ‘ОператорИТС‘, ‘ЦентрУправления‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **Перекресток:** добавить светофор к перекрестку, удалить светофор, получить список светофоров на перекрестке.

    (b) **Светофор:** изменить режим работы (красный/желтый/зеленый), получить текущий режим, добавить информацию о неисправности, вывести список неисправностей.

    (c) **КамераНаблюдения:** включить запись, выключить запись, получить статус работы, зафиксировать нарушение ПДД, вывести список нарушений.

    (d) **ОператорИТС:** класс, представляющий оператора, имеющий идентификатор, смену, список контролируемых перекрестков, ФИО.

(e) **ЦентрУправления:** добавить новый перекресток в систему, установить светофор, установить камеру, нанять оператора, вывести информацию о перекрестках, светофорах, камерах, операторах, удалить перекресток, уволить оператора, снять камеру, снять светофор.

22. **Описание ситуации:** Рассмотрим работу службы эвакуации аварийных транспортных средств. Эвакуаторы дежурят на специальных парковках и выезжают по вызову на места ДТП или поломок. Диспетчеры принимают вызовы и направляют ближайший свободный эвакуатор.

    Создаваемые классы: ‘Эвакуатор‘, ‘Вызов‘, ‘ПарковкаЭвакуаторов‘, ‘ДиспетчерЭвакуации‘, ‘СлужбаЭвакуации‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **Эвакуатор:** принять вызов, завершить вызов, получить текущий статус (свободен/занят), обновить местоположение.

    (b) **Вызов:** зафиксировать время принятия, время выполнения, получить статус выполнения.

    (c) **ПарковкаЭвакуаторов:** принять эвакуатор на парковку, выпустить эвакуатор с парковки, получить список эвакуаторов на парковке.

    (d) **ДиспетчерЭвакуации:** класс, представляющий диспетчера, имеющий идентификатор, смену, список обработанных вызовов, ФИО.

    (e) **СлужбаЭвакуации:** добавить эвакуатор в парк, списать эвакуатор, нанять диспетчера, вывести информацию о эвакуаторах, вызовах, диспетчерах, уволить диспетчера.

23. **Описание ситуации:** Рассмотрим работу центра контроля коммерческих грузоперевозок. Система отслеживает движение грузовых автомобилей, контролирует соблюдение маршрутов, норм труда водителей и расход топлива. Менеджеры по логистике планируют маршруты и анализируют отчеты.

    Создаваемые классы: ‘ГрузовойАвтомобиль‘, ‘МаршрутПеревозки‘, ‘Водитель‘, ‘Рейс‘, ‘МенеджерЛогистики‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **ГрузовойАвтомобиль:** начать рейс, завершить рейс, получить текущий статус, зафиксировать расход топлива.

    (b) **МаршрутПеревозки:** добавить точку маршрута (город, склад), удалить точку, получить полный маршрут.

    (c) **Водитель:** класс, представляющий водителя, имеющий идентификатор, права, график работы, ФИО, стаж.

    (d) **Рейс:** закрепить автомобиль за рейсом, закрепить водителя за рейсом, открепить автомобиль, снять водителя, получить информацию о рейсе.

    (e) **МенеджерЛогистики:** класс, представляющий менеджера, имеющий идентификатор, список контролируемых маршрутов, ФИО.

24. **Описание ситуации:** Рассмотрим работу службы парковки аэропорта. На территории аэропорта есть несколько парковочных зон для разных типов транспорта (краткосрочная, долгосрочная, VIP). Операторы контролируют занятость мест, прием оплаты и работу шлагбаумов.

    Создаваемые классы: ‘ПарковочнаяЗона‘, ‘ПарковочноеМесто‘, ‘Автомобиль‘, ‘ОператорПарковки‘, ‘СлужбаПарковки‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **ПарковочнаяЗона:** добавить парковочное место, удалить место, получить список мест в зоне, получить список всех автомобилей. Так же парковочной зоне соответсвует стоимость часа стоянки.

    (b) **ПарковочноеМесто:** занять место автомобилем, освободить место, получить текущий статус (свободно/занято).

    (c) **Автомобиль:** зафиксировать время въезда, время выезда + рассчитать стоимость парковки (с учетом стоимости часа), получить историю.

    (d) **ОператорПарковки:** класс, представляющий оператора, имеющий идентификатор, смену, список контролируемых зон, ФИО.

    (e) **СлужбаПарковки:** добавить новую парковочную зону, нанять оператора, вывести информацию о зонах, местах, операторах, удалить зону, уволить оператора.

25. **Описание ситуации:** Рассмотрим работу центра управления речным судоходством. Диспетчеры следят за движением судов по фарватеру, распределяют шлюзы, контролируют соблюдение графика движения и обеспечивают безопасность судоходства.

    Создаваемые классы: ‘Судно‘, ‘Шлюз‘, ‘Фарватер‘, ‘ДиспетчерСудоходства‘, ‘ЦентрУправления‘.

    Для классов реализовать следующие простые методы, используя для хранения данных списки (‘[]‘) Python:

    (a) **Судно:** начать движение по фарватеру, завершить движение, получить текущее местоположение, зафиксировать прохождение шлюза.

    (b) **Шлюз:** принять судно для шлюзования, завершить шлюзование, получить текущий статус (свободен/занят).

    (c) **Фарватер:** добавить участок фарватера, удалить участок, получить список судов на фарватере.

    (d) **ДиспетчерСудоходства:** класс, представляющий диспетчера, имеющий идентификатор, смену, список контролируемых шлюзов, ФИО.

    (e) **ЦентрУправления:** добавить шлюз в систему, зарегистрировать судно, нанять диспетчера, вывести информацию о шлюзах, фарватерах, судах, диспетчерах, удалить шлюз, уволить диспетчера.

26. **Описание ситуации:** Рассмотрим работу службы технического контроля метрополитена. Инспекторы проверяют состояние путей, тоннелей, подвижного состава и оборудования станций. Дефекты фиксируются в системе для оперативного устранения ремонтными бригадами.

    Создаваемые классы: ‘УчастокПути‘, ‘ПодвижнойСостав‘, ‘Инспектор‘, ‘Дефект‘, ‘СлужбаКонтроля‘.

Для классов реализовать следующие простые методы, использующие для хранения данных списки (‘[]‘) Python:

(a) **УчастокПути:** добавить информацию о дефекте, получить список неустраненных дефектов на участке.

(b) **ПодвижнойСостав:** добавить запись о техническом осмотре, удалить ошибочную запись, отобразить историю осмотров.

(c) **Инспектор:** класс, представляющий инспектора, имеющий идентификатор, квалификацию, список закрепленных участков, ФИО.

(d) **Дефект:** зафиксировать время обнаружения, время устранения, получить статус устранения.

(e) **СлужбаКонтроля:** добавить участок пути в систему, зарегистрировать подвижной состав, нанять инспектора, вывести информацию об участках, составе, инспекторах, дефектах, удалить участок, уволить инспектора, снять с эксплуатации подвижной состав.

27. **Описание ситуации:** Рассмотрим работу центра управления умными светофорами на перекрестках. Умные светофоры адаптивно меняют режим работы в зависимости от транспортного потока, приоритизируя общественный транспорт и спецтранспорт. Система анализирует данные с датчиков и камер, оптимизируя пропускную способность перекрестков.

    Создаваемые классы: УмныйСветофор, Перекресток, ДатчикТранспортногоПотока, ИнженерАТС, ЦентрУправленияСветофорами.

    Для классов реализовать следующие простые методы, используя для хранения данных списки ([]) Python:

    (a) **УмныйСветофор:** изменить длительность фаз (красный/зеленый), перейти в аварийный режим, получить текущий режим работы.

    (b) **Перекресток:** добавить светофор к перекрестку, удалить светофор, получить список всех светофоров перекрестка.

    (c) **ДатчикТранспортногоПотока:** установить текущие данные о интенсивности движения, получить текущие показания, получить историю показаний.

    (d) **ИнженерАТС:** класс, представляющий инженера автоматизированной транспортной системы, имеющий идентификатор, квалификацию, список закрепленных перекрестков, ФИО.

    (e) **ЦентрУправленияСветофорами:** добавить новый перекресток в систему, установить умный светофор, нанять инженера, вывести информацию о перекрестках, светофорах, инженерах, удалить перекресток, уволить инженера, снять умный светофор.

28. **Описание ситуации:** Рассмотрим работу монорельсовой транспортной системы. Монорельс движется по эстакаде, состоящей из станций и перегонов. Составы имеют фиксированное количество вагонов. Операторы управляют движением составов, следят за соблюдением графика и безопасностью пассажиров.

    Создаваемые классы: СтанцияМонорельса, СоставМонорельса, ВагонМонорельса, ОператорСистемы, УправлениеМонорельсом.

    Для классов реализовать следующие простые методы, используя для хранения данных списки ([]) Python:

(a) **СтанцияМонорельса:** принять состав, отправить состав, получить список составов на станции.

(b) **СоставМонорельса:** добавить вагон в состав (при техническом обслуживании), удалить вагон, получить список вагонов.

(c) **ВагонМонорельса:** зафиксировать текущий пробег, провести техническое обслуживание, получить историю обслуживаний.

(d) **ОператорСистемы:** класс, представляющий оператора, имеющий идентификатор, смену, список закрепленных станций, ФИО.

(e) **УправлениеМонорельсом:** добавить новую станцию, ввести состав в эксплуатацию, нанять оператора, вывести информацию о станциях, составах, операторах, закрыть станцию на ремонт, списать состав, уволить оператора.

29. **Описание ситуации:** Рассмотрим работу канатной дороги. Канатная дорога состоит из линий с опорами и кабинок, перемещающихся между станциями. Кабинки имеют ограниченную вместимость. Техники обслуживают механизмы и следят за безопасностью.

    Создаваемые классы: ЛинияКанатнойДороги, Кабинка, СтанцияКанатнойДороги, Техник, УправлениеКанатнойДорогой.

    Для классов реализовать следующие простые методы, используя для хранения данных списки ([]) Python:

    (a) **ЛинияКанатнойДороги:** добавить кабинку на линию, снять кабинку, получить список кабинок на линии.

    (b) **Кабинка:** запустить в движение, остановить для посадки/высадки, получить текущий статус (движется/стоит).

    (c) **СтанцияКанатнойДороги:** принять кабинку, отправить кабинку, получить список кабинок на станции.

    (d) **Техник:** класс, представляющий техника, имеющий идентификатор, квалификацию, список закрепленных линий, ФИО.

    (e) **УправлениеКанатнойДорогой:** добавить новую линию, ввести кабинку в эксплуатацию, нанять техника, вывести информацию о линиях, кабинках, техниках, закрыть линию на обслуживание, списать кабинку, уволить техника.

30. **Описание ситуации:** Рассмотрим работу службы доставки с использованием дронов. Дроны осуществляют доставку небольших грузов между пунктами выдачи. Каждый дрон имеет грузоподъемность и дальность полета. Операторы управляют полетами дронов и обслуживают пункты выдачи.

    Создаваемые классы: ПунктВыдачи, Дрон, Груз, ОператорДронов, СлужбаДоставки.

    Для классов реализовать следующие простые методы, используя для хранения данных списки ([]) Python:

    (a) **ПунктВыдачи:** принять дрон с грузом, отправить дрон, получить список дронов в пункте.

    (b) **Дрон:** загрузить груз, выгрузить груз, начать полет, завершить полет, получить текущий статус (в полете/на земле).

(c) **Груз:** зарегистрировать отправку, зарегистрировать доставку, получить историю перемещений.

(d) **ОператорДронов:** класс, представляющий оператора, имеющий идентификатор, смену, список закрепленных пунктов выдачи, ФИО.

(e) **СлужбаДоставки:** добавить новый пункт выдачи, ввести дрон в эксплуатацию, нанять оператора, вывести информацию о пунктах, дронах, операторах, закрыть пункт, списать дрон, уволить оператора.

2.2 Семинар «Конструкторы, наследование и полиморфизм. 1 часть» (2 часа)

В ходе работы решите 4 задачи. Предполагается, что пользователь класса не имеет права обращаться к свойствам напрямую (соблюдая принцип инкапсуляции), а должен использовать методы.

Продемонстрируйте работоспособность всех методов (из задания) посредством создания запускаемых файлов, где осуществляется вызов методов для разных ситуаций (без ручного ввода, но с выводом результатов в консоль).

Каждый класс должен сохраняться в отдельном исходном файле. Необходимо соблюдать все стандартные требования к качеству кода (отступы, именования переменных, классов, методов, проверка корректности входных данных). Для каждого класса создайте отдельный запускаемый файл для проверки всех его методов (допускается использование других классов в этих тестах).

Все предлагаемые классы в заданиях упрощенные; для использования в production-окружении они требуют серьезной доработки. Суть задания — в отработке базовых навыков, а не в идеальном моделировании предложенных ситуаций.

Для сдачи работы будьте готовы пояснить или аналогично заданию модифицировать любую часть кода, а также ответить на вопросы:

1. Что обозначает свойство наследования в парадигме ООП?
2. Что обозначает свойство полиморфизма в парадигме ООП?
3. Опишите реализацию наследования в Python
4. Как создать конструктор в Python
5. Как реализовать абстрактный класс в Python (и что это значит)
6. Как реализовать абстрактные методы в Python (и что это значит)

Если вы нашли в задачнике ошибки, опечатки и другие недостатки, то вы можете сделать pull-request.

Срок сдачи работы (начала сдачи): через одно занятие после его выдачи. В последующие сроки оценка будет снижаться (при отсутствии оправдывающих документов).

2.2.1 Задача 1

1. Напишите программу, которая создаёт класс `Circle` с методами для вычисления площади и длины окружности (периметра). Программа должна запрашивать у пользователя радиус и выводить вычисленные площадь и длину окружности.

   Инструкции:

   (a) Создайте класс `Circle` с методом `__init__`, который принимает радиус окружности в качестве аргумента и сохраняет его в атрибуте `self.__radius`.

   (b) Создайте метод `calculate_circle_area`, без аргументов, который вычисляет площадь круга по формуле:

   $$\pi \cdot \texttt{\_\_radius}^2$$

(c) Создайте метод `calculate_circle_perimeter` без аргументов, который вычисляет длину окружности по формуле:

$$2 \cdot \pi \cdot \texttt{\_\_radius}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя радиус окружности,
ii. создавать экземпляр класса `Circle` с этим радиусом,
iii. вычислять площадь и длину окружности с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
radius = 3
circle = Circle(radius)
area = circle.calculate_circle_area()
perimeter = circle.calculate_circle_perimeter()
print(f"Площадь окружности равна: {area}")
print(f"Периметр окружности равен: {perimeter}")
```

Вывод:

```
Площадь окружности равна: 28.274333882308138
Периметр окружности равен: 18.84955592153876
```

2. Напишите программу, которая создаёт класс `Square` с методами для вычисления площади и периметра. Программа должна запрашивать у пользователя длину стороны и выводить вычисленные площадь и периметр.

Инструкции:

(a) Создайте класс `Square` с методом `__init__`, который принимает длину стороны квадрата в качестве аргумента и сохраняет её в атрибуте `self.__side`.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь квадрата по формуле:

$$\texttt{\_\_side}^2$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр квадрата по формуле:

$$4 \cdot \texttt{\_\_side}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину стороны квадрата,
ii. создавать экземпляр класса `Square` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
side = 5
square = Square(side)
area = square.calculate_area()
perimeter = square.calculate_perimeter()
print(f"Площадь квадрата равна: {area}")
print(f"Периметр квадрата равен: {perimeter}")
```

Вывод:

```
Площадь квадрата равна: 25
Периметр квадрата равен: 20
```

3. Напишите программу, которая создаёт класс `Rectangle` с методами для вычисления площади и периметра. Программа должна запрашивать у пользователя ширину прямоугольника (при соотношении сторон 1:2) и выводить вычисленные площадь и периметр.

Инструкции:

(a) Создайте класс `Rectangle` с методом `__init__`, который принимает ширину прямоугольника в качестве аргумента и сохраняет её в атрибуте `self.__width`. Высота прямоугольника должна быть в два раза больше ширины.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь прямоугольника по формуле:

$$\texttt{\_\_width} \cdot (2 \cdot \texttt{\_\_width})$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр прямоугольника по формуле:

$$2 \cdot (\texttt{\_\_width} + 2 \cdot \texttt{\_\_width})$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя ширину прямоугольника,
ii. создавать экземпляр класса `Rectangle` с этой шириной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
width = 3
rectangle = Rectangle(width)
area = rectangle.calculate_area()
perimeter = rectangle.calculate_perimeter()
print(f"Площадь прямоугольника равна: {area}")
print(f"Периметр прямоугольника равен: {perimeter}")
```

Вывод:

```
Площадь прямоугольника равна: 18
Периметр прямоугольника равен: 18
```

4. Напишите программу, которая создаёт класс `Triangle` с методами для вычисления площади и периметра. Программа должна запрашивать у пользователя длину стороны равностороннего треугольника и выводить вычисленные площадь и периметр.

Инструкции:

(a) Создайте класс `Triangle` с методом `__init__`, который принимает длину стороны треугольника в качестве аргумента и сохраняет её в атрибуте `self.__side`.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь равностороннего треугольника по формуле:

$$\frac{\sqrt{3}}{4} \cdot \texttt{\_\_side}^2$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр треугольника по формуле:

$$3 \cdot \texttt{\_\_side}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину стороны треугольника,
ii. создавать экземпляр класса `Triangle` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
side = 4
triangle = Triangle(side)
area = triangle.calculate_area()
perimeter = triangle.calculate_perimeter()
print(f"Площадь треугольника равна: {area}")
print(f"Периметр треугольника равен: {perimeter}")
```

Вывод:

```
Площадь треугольника равна: 6.928203230275509
Периметр треугольника равен: 12
```

5. Напишите программу, которая создаёт класс `Sphere` с методами для вычисления площади поверхности и объёма. Программа должна запрашивать у пользователя радиус сферы и выводить вычисленные площадь поверхности и объём.

Инструкции:

(a) Создайте класс `Sphere` с методом `__init__`, который принимает радиус сферы в качестве аргумента и сохраняет его в атрибуте `self.__radius`.

(b) Создайте метод `calculate_surface_area`, без аргументов, который вычисляет площадь поверхности сферы по формуле:

$$4 \cdot \pi \cdot \texttt{\_\_radius}^2$$

(c) Создайте метод `calculate_volume` без аргументов, который вычисляет объём сферы по формуле:

$$\frac{4}{3} \cdot \pi \cdot \texttt{\_\_radius}^3$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя радиус сферы,
ii. создавать экземпляр класса `Sphere` с этим радиусом,
iii. вычислять площадь поверхности и объём с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
radius = 2
sphere = Sphere(radius)
surface_area = sphere.calculate_surface_area()
volume = sphere.calculate_volume()
print(f"Площадь поверхности сферы равна: {surface_area}")
print(f"Объём сферы равен: {volume}")
```

Вывод:

```
Площадь поверхности сферы равна: 50.26548245743669
Объём сферы равен: 33.510321638291124
```

6. Напишите программу, которая создаёт класс `Cylinder` с методами для вычисления объёма и площади боковой поверхности. Программа должна запрашивать у пользователя радиус основания и выводить вычисленные объём и площадь боковой поверхности (высота цилиндра фиксирована и равна 5).

Инструкции:

(a) Создайте класс `Cylinder` с методом `__init__`, который принимает радиус основания цилиндра в качестве аргумента и сохраняет его в атрибуте `self.__radius`. Высота цилиндра фиксирована и равна 5.

(b) Создайте метод `calculate_volume`, без аргументов, который вычисляет объём цилиндра по формуле:

$$\pi \cdot \texttt{\_\_radius}^2 \cdot 5$$

(c) Создайте метод `calculate_lateral_area` без аргументов, который вычисляет площадь боковой поверхности цилиндра по формуле:

$$2 \cdot \pi \cdot \texttt{\_\_radius} \cdot 5$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя радиус основания цилиндра,

ii. создавать экземпляр класса `Cylinder` с этим радиусом,

iii. вычислять объём и площадь боковой поверхности с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
radius = 3
cylinder = Cylinder(radius)
volume = cylinder.calculate_volume()
lateral_area = cylinder.calculate_lateral_area()
print(f"Объём цилиндра равен: {volume}")
print(f"Площадь боковой поверхности равна: {lateral_area}")
```

Вывод:

```
Объём цилиндра равен: 141.3716694115407
Площадь боковой поверхности равна: 94.24777960769379
```

7. Напишите программу, которая создаёт класс `Cone` с методами для вычисления объёма и площади боковой поверхности. Программа должна запрашивать у пользователя радиус основания и выводить вычисленные объём и площадь боковой поверхности (высота конуса фиксирована и равна 10).

Инструкции:

(a) Создайте класс `Cone` с методом `__init__`, который принимает радиус основания конуса в качестве аргумента и сохраняет его в атрибуте `self.__radius`. Высота конуса фиксирована и равна 10.

(b) Создайте метод `calculate_volume`, без аргументов, который вычисляет объём конуса по формуле:

$$\frac{1}{3} \cdot \pi \cdot \texttt{\_\_radius}^2 \cdot 10$$

(c) Создайте метод `calculate_lateral_area` без аргументов, который вычисляет площадь боковой поверхности конуса по формуле:

$$\pi \cdot \texttt{\_\_radius} \cdot \sqrt{\texttt{\_\_radius}^2 + 10^2}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя радиус основания конуса,

ii. создавать экземпляр класса `Cone` с этим радиусом,

iii. вычислять объём и площадь боковой поверхности с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
radius = 3
cone = Cone(radius)
volume = cone.calculate_volume()
lateral_area = cone.calculate_lateral_area()
print(f"Объём конуса равен: {volume}")
print(f"Площадь боковой поверхности равна: {lateral_area}")
```

Вывод:

```
Объём конуса равен: 94.24777960769379
Площадь боковой поверхности равна: 94.86832980505137
```

8. Напишите программу, которая создаёт класс `Cube` с методами для вычисления объёма и площади полной поверхности. Программа должна запрашивать у пользователя длину ребра куба и выводить вычисленные объём и площадь.

Инструкции:

(a) Создайте класс `Cube` с методом `__init__`, который принимает длину ребра куба в качестве аргумента и сохраняет её в атрибуте `self.__side`.

(b) Создайте метод `calculate_volume`, без аргументов, который вычисляет объём куба по формуле:

$$\texttt{\_\_side}^3$$

(c) Создайте метод `calculate_surface_area` без аргументов, который вычисляет площадь полной поверхности куба по формуле:

$$6 \cdot \texttt{\_\_side}^2$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину ребра куба,

ii. создавать экземпляр класса `Cube` с этой длиной,

iii. вычислять объём и площадь полной поверхности с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
side = 4
cube = Cube(side)
volume = cube.calculate_volume()
surface_area = cube.calculate_surface_area()
print(f"Объём куба равен: {volume}")
print(f"Площадь полной поверхности равна: {surface_area}")
```

Вывод:

```
Объём куба равен: 64
Площадь полной поверхности равна: 96
```

9. Напишите программу, которая создаёт класс `Parallelogram` с методами для вычисления площади и периметра. Программа должна запрашивать у пользователя длину основания параллелограмма и выводить вычисленные площадь и периметр (высота параллелограмма фиксирована и равна 8, а боковая сторона равна 6).

Инструкции:

(a) Создайте класс `Parallelogram` с методом `__init__`, который принимает длину основания параллелограмма в качестве аргумента и сохраняет её в атрибуте `self.__base`. Высота параллелограмма фиксирована и равна 8, а боковая сторона равна 6.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь параллелограмма по формуле:

$$\texttt{\_\_base} \cdot 8$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр параллелограмма по формуле:

$$2 \cdot (\texttt{\_\_base} + 6)$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину основания параллелограмма,
ii. создавать экземпляр класса `Parallelogram` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
base = 5
parallelogram = Parallelogram(base)
area = parallelogram.calculate_area()
perimeter = parallelogram.calculate_perimeter()
print(f"Площадь параллелограмма равна: {area}")
print(f"Периметр параллелограмма равен: {perimeter}")
```

Вывод:

```
Площадь параллелограмма равна: 40
Периметр параллелограмма равен: 22
```

10. Напишите программу, которая создаёт класс `Ellipse` с методами для вычисления площади и приближённого значения периметра. Программа должна запрашивать у пользователя длину большой полуоси и выводить вычисленные площадь и периметр (длина малой полуоси фиксирована и равна 3).

Инструкции:

(a) Создайте класс `Ellipse` с методом `__init__`, который принимает длину большой полуоси эллипса в качестве аргумента и сохраняет её в атрибуте `self.__major_axis`. Длина малой полуоси фиксирована и равна 3.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь эллипса по формуле:

$$\pi \cdot \texttt{\_\_major\_axis} \cdot 3$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет приближённое значение периметра эллипса по формуле Рамануджана:

$$\pi \cdot \left(3(\texttt{\_\_major\_axis} + 3) - \sqrt{(3\texttt{\_\_major\_axis} + 3)(\texttt{\_\_major\_axis} + 9)}\right)$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину большой полуоси эллипса,
ii. создавать экземпляр класса `Ellipse` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
major_axis = 5
ellipse = Ellipse(major_axis)
area = ellipse.calculate_area()
perimeter = ellipse.calculate_perimeter()
print(f"Площадь эллипса равна: {area}")
print(f"Периметр эллипса равен: {perimeter}")
```

Вывод:

```
Площадь эллипса равна: 47.12388980384689
Периметр эллипса равен: 25.74488980384689
```

11. Напишите программу, которая создаёт класс `BankAccount` с методами для вычисления начисленных процентов и суммы налога на доход. Программа должна запрашивать у пользователя начальный баланс счёта и выводить вычисленные проценты и налог (процентная ставка фиксирована и равна 5%, налоговая ставка на доход фиксирована и равна 13%).

Инструкции:

(a) Создайте класс `BankAccount` с методом `__init__`, который принимает начальный баланс счёта в качестве аргумента и сохраняет его в атрибуте `self.__balance`.

(b) Создайте метод `calculate_interest`, без аргументов, который вычисляет начисленные проценты по формуле:

$$\texttt{\_\_balance} \cdot 0.05$$

(c) Создайте метод `calculate_tax` без аргументов, который вычисляет сумму налога на полученный доход (проценты) по формуле:

$$(\texttt{\_\_balance} \cdot 0.05) \cdot 0.13$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя начальный баланс счёта,

ii. создавать экземпляр класса `BankAccount` с этим балансом,

iii. вычислять начисленные проценты и сумму налога с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
balance = 1000
account = BankAccount(balance)
interest = account.calculate_interest()
tax = account.calculate_tax()
print(f"Начисленные проценты: {interest}")
print(f"Сумма налога на доход: {tax}")
```

Вывод:

```
Начисленные проценты: 50.0
Сумма налога на доход: 6.5
```

12. Напишите программу, которая создаёт класс `TemperatureConverter` с методами для преобразования температуры из градусов Цельсия в Фаренгейты и Кельвины. Программа должна запрашивать у пользователя температуру в Цельсиях и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `TemperatureConverter` с методом `__init__`, который принимает температуру в градусах Цельсия в качестве аргумента и сохраняет её в атрибуте `self.__celsius`.

(b) Создайте метод `to_fahrenheit`, без аргументов, который преобразует температуру в Фаренгейты по формуле:

$$(\texttt{\_\_celsius} \times \frac{9}{5}) + 32$$

(c) Создайте метод `to_kelvin` без аргументов, который преобразует температуру в Кельвины по формуле:

$$\texttt{\_\_celsius} + 273.15$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя температуру в градусах Цельсия,
ii. создавать экземпляр класса `TemperatureConverter` с этим значением,
iii. вычислять температуру в Фаренгейтах и Кельвинах с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
celsius = 25
converter = TemperatureConverter(celsius)
fahrenheit = converter.to_fahrenheit()
kelvin = converter.to_kelvin()
print(f"Температура в Фаренгейтах: {fahrenheit}")
print(f"Температура в Кельвинах: {kelvin}")
```

Вывод:

```
Температура в Фаренгейтах: 77.0
Температура в Кельвинах: 298.15
```

13. Напишите программу, которая создаёт класс `DistanceConverter` с методами для преобразования расстояния из метров в километры и мили. Программа должна запрашивать у пользователя расстояние в метрах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `DistanceConverter` с методом `__init__`, который принимает расстояние в метрах в качестве аргумента и сохраняет его в атрибуте `self.__meters`.

(b) Создайте метод `to_kilometers`, без аргументов, который преобразует расстояние в километры по формуле:

$$\texttt{\_\_meters} \div 1000$$

(c) Создайте метод `to_miles` без аргументов, который преобразует расстояние в мили по формуле:

$$\texttt{\_\_meters} \div 1609.344$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя расстояние в метрах,

ii. создавать экземпляр класса `DistanceConverter` с этим значением,

iii. вычислять расстояние в километрах и милях с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
meters = 1609.344
converter = DistanceConverter(meters)
kilometers = converter.to_kilometers()
miles = converter.to_miles()
print(f"Расстояние в километрах: {kilometers}")
print(f"Расстояние в милях: {miles}")
```

Вывод:

```
Расстояние в километрах: 1.609344
Расстояние в милях: 1.0
```

14. Напишите программу, которая создаёт класс `WeightConverter` с методами для преобразования массы из килограммов в граммы и фунты. Программа должна запрашивать у пользователя массу в килограммах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `WeightConverter` с методом `__init__`, который принимает массу в килограммах в качестве аргумента и сохраняет её в атрибуте `self.__kg`.

(b) Создайте метод `to_grams`, без аргументов, который преобразует массу в граммы по формуле:

$$\texttt{\_\_kg} \times 1000$$

(c) Создайте метод `to_pounds` без аргументов, который преобразует массу в фунты по формуле:

$$\texttt{\_\_kg} \times 2.20462$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя массу в килограммах,

ii. создавать экземпляр класса `WeightConverter` с этим значением,

iii. вычислять массу в граммах и фунтах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
kg = 2.5
converter = WeightConverter(kg)
grams = converter.to_grams()
pounds = converter.to_pounds()
print(f"Масса в граммах: {grams}")
print(f"Масса в фунтах: {pounds}")
```

Вывод:

```
Масса в граммах: 2500.0
Масса в фунтах: 5.51155
```

15. Напишите программу, которая создаёт класс `TimeConverter` с методами для преобразования времени из секунд в минуты и часы. Программа должна запрашивать у пользователя время в секундах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `TimeConverter` с методом `__init__`, который принимает время в секундах в качестве аргумента и сохраняет его в атрибуте `self.__seconds`.

(b) Создайте метод `to_minutes`, без аргументов, который преобразует время в минуты по формуле:

$$\texttt{\_\_seconds} \div 60$$

(c) Создайте метод `to_hours` без аргументов, который преобразует время в часы по формуле:

$$\texttt{\_\_seconds} \div 3600$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя время в секундах,

ii. создавать экземпляр класса `TimeConverter` с этим значением,

iii. вычислять время в минутах и часах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
seconds = 7200
converter = TimeConverter(seconds)
minutes = converter.to_minutes()
hours = converter.to_hours()
print(f"Время в минутах: {minutes}")
print(f"Время в часах: {hours}")
```

Вывод:

```
Время в минутах: 120.0
Время в часах: 2.0
```

16. Напишите программу, которая создаёт класс `SpeedConverter` с методами для преобразования скорости из километров в час в метры в секунду и мили в час. Программа должна запрашивать у пользователя скорость в км/ч и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `SpeedConverter` с методом `__init__`, который принимает скорость в км/ч в качестве аргумента и сохраняет её в атрибуте `self.__kmh`.

(b) Создайте метод `to_ms`, без аргументов, который преобразует скорость в м/с по формуле:

$$\texttt{\_\_kmh} \times \frac{1000}{3600}$$

(c) Создайте метод `to_mph` без аргументов, который преобразует скорость в мили/ч по формуле:

$$\texttt{\_\_kmh} \div 1.60934$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя скорость в км/ч,
ii. создавать экземпляр класса `SpeedConverter` с этим значением,
iii. вычислять скорость в м/с и милях/ч с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
kmh = 100
converter = SpeedConverter(kmh)
ms = converter.to_ms()
mph = converter.to_mph()
print(f"Скорость в м/с: {ms}")
print(f"Скорость в милях/ч: {mph}")
```

Вывод:

```
Скорость в м/с: 27.77777777777778
Скорость в милях/ч: 62.13727366498068
```

17. Напишите программу, которая создаёт класс `AreaConverter` с методами для преобразования площади из квадратных метров в гектары и акры. Программа должна запрашивать у пользователя площадь в $м^2$ и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `AreaConverter` с методом `__init__`, который принимает площадь в м^2 в качестве аргумента и сохраняет её в атрибуте `self.__sq_meters`.

(b) Создайте метод `to_hectares`, без аргументов, который преобразует площадь в гектары по формуле:

$$\texttt{\_\_sq\_meters} \div 10000$$

(c) Создайте метод `to_acres` без аргументов, который преобразует площадь в акры по формуле:

$$\texttt{\_\_sq\_meters} \div 4046.86$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя площадь в м^2,

ii. создавать экземпляр класса `AreaConverter` с этим значением,

iii. вычислять площадь в гектарах и акрах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
sq_meters = 10000
converter = AreaConverter(sq_meters)
hectares = converter.to_hectares()
acres = converter.to_acres()
print(f"Площадь в гектарах: {hectares}")
print(f"Площадь в акрах: {acres}")
```

Вывод:

```
Площадь в гектары: 1.0
Площадь в акрах: 2.4710514233241505
```

18. Напишите программу, которая создаёт класс `VolumeConverter` с методами для преобразования объёма из литров в галлоны и кубические метры. Программа должна запрашивать у пользователя объём в литрах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `VolumeConverter` с методом `__init__`, который принимает объём в литрах в качестве аргумента и сохраняет его в атрибуте `self.__liters`.

(b) Создайте метод `to_gallons`, без аргументов, который преобразует объём в галлоны по формуле:

$$\texttt{\_\_liters} \div 3.78541$$

(c) Создайте метод `to_cubic_meters` без аргументов, который преобразует объём в кубические метры по формуле:

$$\texttt{\_\_liters} \div 1000$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя объём в литрах,

ii. создавать экземпляр класса `VolumeConverter` с этим значением,

iii. вычислять объём в галлонах и кубических метрах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
liters = 10
converter = VolumeConverter(liters)
gallons = converter.to_gallons()
cubic_meters = converter.to_cubic_meters()
print(f"Объём в галлонах: {gallons}")
print(f"Объём в кубических метрах: {cubic_meters}")
```

Вывод:

```
Объём в галлонах: 2.641720523581484
Объём в кубических метрах: 0.01
```

19. Напишите программу, которая создаёт класс `EnergyConverter` с методами для преобразования энергии из джоулей в калории и киловатт-часы. Программа должна запрашивать у пользователя энергию в джоулях и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `EnergyConverter` с методом `__init__`, который принимает энергию в джоулях в качестве аргумента и сохраняет её в атрибуте `self.__joules`.

(b) Создайте метод `to_calories`, без аргументов, который преобразует энергию в калории по формуле:

$$\texttt{\_\_joules} \div 4.184$$

(c) Создайте метод `to_kwh` без аргументов, который преобразует энергию в киловатт-часы по формуле:

$$\texttt{\_\_joules} \div 3.6 \times 10^6$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя энергию в джоулях,

ii. создавать экземпляр класса `EnergyConverter` с этим значением,

iii. вычислять энергию в калориях и киловатт-часах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
joules = 10000
converter = EnergyConverter(joules)
calories = converter.to_calories()
kwh = converter.to_kwh()
print(f"Энергия в калориях: {calories}")
print(f"Энергия в киловатт-часах: {kwh}")
```

Вывод:

```
Энергия в калориях: 2390.057361376673
Энергия в киловатт-часах: 0.002777777777777778
```

20. Напишите программу, которая создаёт класс `PowerConverter` с методами для преобразования мощности из ватт в лошадиные силы и киловатты. Программа должна запрашивать у пользователя мощность в ваттах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `PowerConverter` с методом `__init__`, который принимает мощность в ваттах в качестве аргумента и сохраняет её в атрибуте `self.__watts`.

(b) Создайте метод `to_horsepower`, без аргументов, который преобразует мощность в лошадиные силы по формуле:

$$\texttt{\_\_watts} \div 745.7$$

(c) Создайте метод `to_kilowatts` без аргументов, который преобразует мощность в киловатты по формуле:

$$\texttt{\_\_watts} \div 1000$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя мощность в ваттах,
ii. создавать экземпляр класса `PowerConverter` с этим значением,
iii. вычислять мощность в л.с. и киловаттах с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
watts = 1000
converter = PowerConverter(watts)
horsepower = converter.to_horsepower()
kilowatts = converter.to_kilowatts()
print(f"Мощность в л.с.: {horsepower}")
print(f"Мощность в киловаттах: {kilowatts}")
```

Вывод:

```
Мощность в л.с.: 1.3410220903956017
Мощность в киловаттах: 1.0
```

21. Напишите программу, которая создаёт класс `PressureConverter` с методами для преобразования давления из паскалей в атмосферы и бары. Программа должна запрашивать у пользователя давление в паскалях и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `PressureConverter` с методом `__init__`, который принимает давление в паскалях в качестве аргумента и сохраняет его в атрибуте `self.__pascals`.

(b) Создайте метод `to_atm`, без аргументов, который преобразует давление в атмосферы по формуле:

$$\texttt{\_\_pascals} \div 101325$$

(c) Создайте метод `to_bar` без аргументов, который преобразует давление в бары по формуле:

$$\texttt{\_\_pascals} \div 100000$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя давление в паскалях,

ii. создавать экземпляр класса `PressureConverter` с этим значением,

iii. вычислять давление в атмосферах и барах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
pascals = 101325
converter = PressureConverter(pascals)
atm = converter.to_atm()
bar = converter.to_bar()
print(f"Давление в атмосферах: {atm}")
print(f"Давление в барах: {bar}")
```

Вывод:

```
Давление в атмосферах: 1.0
Давление в барах: 1.01325
```

22. Напишите программу, которая создаёт класс `ForceConverter` с методами для преобразования силы из ньютонов в дины и фунты-силы. Программа должна запрашивать у пользователя силу в ньютонах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `ForceConverter` с методом `__init__`, который принимает силу в ньютонах в качестве аргумента и сохраняет её в атрибуте `self.__newtons`.

(b) Создайте метод `to_dyne`, без аргументов, который преобразует силу в дины по формуле:

$$\texttt{\_\_newtons} \times 100000$$

(c) Создайте метод `to_pound_force` без аргументов, который преобразует силу в фунты-силы по формуле:

$$\texttt{\_\_newtons} \div 4.44822$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя силу в ньютонах,

ii. создавать экземпляр класса `ForceConverter` с этим значением,

iii. вычислять силу в динах и фунтах-силы с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
newtons = 10
converter = ForceConverter(newtons)
dyne = converter.to_dyne()
pound_force = converter.to_pound_force()
print(f"Сила в динах: {dyne}")
print(f"Сила в фунтах-силы: {pound_force}")
```

Вывод:

```
Сила в динах: 1000000.0
Сила в фунтах-силы: 2.248089430997145
```

23. Задание: Конвертер силы

Напишите программу, которая создаёт класс `ForceConverter` с методами для преобразования силы из ньютонов в дины и фунты-силы. Программа должна запрашивать у пользователя силу в ньютонах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `ForceConverter` с методом `__init__`, который принимает силу в ньютонах в качестве аргумента и сохраняет её в атрибуте `self.__newtons`.

(b) Создайте метод `to_dyne`, без аргументов, который преобразует силу в дины по формуле:

$$\texttt{\_\_newtons} \times 100000$$

(c) Создайте метод `to_pound_force` без аргументов, который преобразует силу в фунты-силы по формуле:

$$\texttt{\_\_newtons} \div 4.44822$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя силу в ньютонах,
ii. создавать экземпляр класса `ForceConverter` с этим значением,
iii. вычислять силу в динах и фунтах-силы с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
newtons = 10
converter = ForceConverter(newtons)
dyne = converter.to_dyne()
pound_force = converter.to_pound_force()
print(f"Сила в динах: {dyne}")
print(f"Сила в фунтах-силы: {pound_force}")
```

Вывод:

```
Сила в динах: 1000000.0
Сила в фунтах-силы: 2.248089430997145
```

24. Напишите программу, которая создаёт класс `ResistanceConverter` с методами для преобразования электрического сопротивления из омов в килоомы и мегаомы. Программа должна запрашивать у пользователя сопротивление в омах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `ResistanceConverter` с методом `__init__`, который принимает сопротивление в омах в качестве аргумента и сохраняет его в атрибуте `self.__ohms`.

(b) Создайте метод `to_kiloohms`, без аргументов, который преобразует сопротивление в килоомы по формуле:

$$\texttt{\_\_ohms} \div 1000$$

(c) Создайте метод `to_megaohms` без аргументов, который преобразует сопротивление в мегаомы по формуле:

$$\texttt{\_\_ohms} \div 1000000$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя сопротивление в омах,
ii. создавать экземпляр класса `ResistanceConverter` с этим значением,
iii. вычислять сопротивление в килоомах и мегаомах с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
ohms = 10000
converter = ResistanceConverter(ohms)
kiloohms = converter.to_kiloohms()
megaohms = converter.to_megaohms()
print(f"Сопротивление в килоомах: {kiloohms}")
print(f"Сопротивление в мегаомах: {megaohms}")
```

Вывод:

```
Сопротивление в килоомах: 10.0
Сопротивление в мегаомах: 0.01
```

25. Дополнительные задания

26. Напишите программу, которая создаёт класс `Pentagon` с методами для вычисления площади и периметра правильного пятиугольника. Программа должна запрашивать у пользователя длину сторону и выводить вычисленные площадь и периметр.

Инструкции:

(a) Создайте класс `Pentagon` с методом `__init__`, который принимает длину стороны пятиугольника в качестве аргумента и сохраняет её в атрибуте `self.__side`.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь правильного пятиугольника по формуле:

$$\frac{1}{4}\sqrt{5(5+2\sqrt{5})}\cdot \texttt{\_\_side}^2$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр пятиугольника по формуле:

$$5\cdot \texttt{\_\_side}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину стороны пятиугольника,
ii. создавать экземпляр класса `Pentagon` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
side = 5
pentagon = Pentagon(side)
area = pentagon.calculate_area()
perimeter = pentagon.calculate_perimeter()
print(f"Площадь пятиугольника: {area}")
print(f"Периметр пятиугольника: {perimeter}")
```

Вывод:

```
Площадь пятиугольника: 43.01193501472417
Периметр пятиугольника: 25
```

27. Напишите программу, которая создаёт класс `Hexagon` с методами для вычисления площади и периметра правильного шестиугольника. Программа должна запрашивать у пользователя длину стороны и выводить вычисленные площадь и периметр.

Инструкции:

(a) Создайте класс `Hexagon` с методом `__init__`, который принимает длину стороны шестиугольника в качестве аргумента и сохраняет её в атрибуте `self.__side`.

(b) Создайте метод `calculate_area`, без аргументов, который вычисляет площадь правильного шестиугольника по формуле:

$$\frac{3\sqrt{3}}{2} \cdot \texttt{\_\_side}^2$$

(c) Создайте метод `calculate_perimeter` без аргументов, который вычисляет периметр шестиугольника по формуле:

$$6 \cdot \texttt{\_\_side}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину стороны шестиугольника,
ii. создавать экземпляр класса `Hexagon` с этой длиной,
iii. вычислять площадь и периметр с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
side = 4
hexagon = Hexagon(side)
area = hexagon.calculate_area()
perimeter = hexagon.calculate_perimeter()
print(f"Площадь шестиугольника: {area}")
print(f"Периметр шестиугольника: {perimeter}")
```

Вывод:

```
Площадь шестиугольника: 41.569219381653056
Периметр шестиугольника: 24
```

28. Напишите программу, которая создаёт класс `AngleConverter` с методами для преобразования углов из градусов в радианы и грады. Программа должна запрашивать у пользователя угол в градусах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `AngleConverter` с методом `__init__`, который принимает угол в градусах в качестве аргумента и сохраняет его в атрибуте `self.__degrees`.

(b) Создайте метод `to_radians`, без аргументов, который преобразует угол в радианы по формуле:

$$\texttt{\_\_degrees} \times \frac{\pi}{180}$$

(c) Создайте метод `to_gradians` без аргументов, который преобразует угол в грады по формуле:

$$\texttt{\_\_degrees} \times \frac{10}{9}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя угол в градусах,

ii. создавать экземпляр класса `AngleConverter` с этим значением,

iii. вычислять угол в радианах и градах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
degrees = 90
converter = AngleConverter(degrees)
radians = converter.to_radians()
gradians = converter.to_gradians()
print(f"Угол в радианах: {radians}")
print(f"Угол в градах: {gradians}")
```

Вывод:

```
Угол в радианах: 1.5707963267948966
Угол в градах: 100.0
```

29. Напишите программу, которая создаёт класс `Tetrahedron` с методами для вычисления объёма и площади поверхности правильного тетраэдра. Программа должна запрашивать у пользователя длину ребра и выводить вычисленные объём и площадь поверхности.

Инструкции:

(a) Создайте класс `Tetrahedron` с методом `__init__`, который принимает длину ребра тетраэдра в качестве аргумента и сохраняет её в атрибуте `self.__edge`.

(b) Создайте метод `calculate_volume`, без аргументов, который вычисляет объём тетраэдра по формуле:

$$\frac{\texttt{\_\_edge}^3}{6\sqrt{2}}$$

(c) Создайте метод `calculate_surface_area` без аргументов, который вычисляет площадь поверхности тетраэдра по формуле:

$$\sqrt{3} \cdot \texttt{\_\_edge}^2$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину ребра тетраэдра,
ii. создавать экземпляр класса `Tetrahedron` с этой длиной,
iii. вычислять объём и площадь поверхности с помощью соответствующих методов,
iv. выводить результаты на экран.

Пример использования:

```
edge = 3
tetrahedron = Tetrahedron(edge)
volume = tetrahedron.calculate_volume()
surface_area = tetrahedron.calculate_surface_area()
print(f"Объём тетраэдра: {volume}")
print(f"Площадь поверхности: {surface_area}")
```

Вывод:

```
Объём тетраэдра: 3.181980515339464
Площадь поверхности: 15.588457268119896
```

30. Напишите программу, которая создаёт класс `CubicMeterConverter` с методами для преобразования объёма из кубических метров в литры и кубические футы. Программа должна запрашивать у пользователя объём в кубометрах и выводить преобразованные значения.

Инструкции:

(a) Создайте класс `CubicMeterConverter` с методом `__init__`, который принимает объём в кубических метрах в качестве аргумента и сохраняет его в атрибуте `self.__cubic_meters`.

(b) Создайте метод `to_liters`, без аргументов, который преобразует объём в литры по формуле:

$$\texttt{\_\_cubic\_meters} \times 1000$$

(c) Создайте метод `__cubic_feet` без аргументов, который преобразует объём в кубические футы по формуле:

$$\texttt{\_\_cubic\_meters} \times 35.3147$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя объём в кубических метрах,

ii. создавать экземпляр класса `CubicMeterConverter` с этим значением,

iii. вычислять объём в литрах и кубических футах с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
cubic_meters = 2
converter = CubicMeterConverter(cubic_meters)
liters = converter.to_liters()
cubic_feet = converter.to_cubic_feet()
print(f"Объём в литрах: {liters}")
print(f"Объём в кубических футах: {cubic_feet}")
```

Вывод:

```
Объём в литрах: 2000.0
Объём в кубических футах: 70.6294
```

31. Напишите программу, которая создаёт класс `RightTriangle` с методами для вычисления гипотенузы и площади прямоугольного треугольника. Программа должна запрашивать у пользователя длину одного катета (второй катет фиксирован и равен 4) и выводить вычисленные гипотенузу и площадь.

Инструкции:

(a) Создайте класс `RightTriangle` с методом `__init__`, который принимает длину первого катета в качестве аргумента и сохраняет его в атрибуте `self.__cathetus`. Второй катет фиксирован и равен 4.

(b) Создайте метод `calculate_hypotenuse`, без аргументов, который вычисляет гипотенузу по формуле:

$$\sqrt{\texttt{\_\_cathetus}^2 + 4^2}$$

(c) Создайте метод `calculate_area` без аргументов, который вычисляет площадь треугольника по формуле:

$$\frac{\texttt{\_\_cathetus} \times 4}{2}$$

(d) Напишите цикл, который повторяется 10 раз. В каждой итерации программа должна:

i. запрашивать у пользователя длину катета,

ii. создавать экземпляр класса `RightTriangle` с этой длиной,

iii. вычислять гипотенузу и площадь с помощью соответствующих методов,

iv. выводить результаты на экран.

Пример использования:

```
cathetus = 3
triangle = RightTriangle(cathetus)
hypotenuse = triangle.calculate_hypotenuse()
area = triangle.calculate_area()
print(f"Гипотенуза: {hypotenuse}")
print(f"Площадь: {area}")
```

Вывод:

```
Гипотенуза: 5.0
Площадь: 6.0
```

2.2.2 Задача 2

1. Написать программу, которая создаёт класс `LeapYearChecker` для определения високосного года. В классе должен быть статический метод `is_leap_year` и возвращать `True`, если год високосный, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого года от 2000 до 2099 и вывода результата на экран.

Инструкции:

(a) Создайте класс `LeapYearChecker`.

(b) Создайте **статический** метод `is_leap_year`, который принимает год в качестве аргумента и проверяет, является ли год високосным. Если год делится на 4 без остатка и не делится на 100 без остатка, или делится на 400 без остатка, то возвращает `True`. В противном случае возвращает `False`.

(c) Используйте цикл для проверки каждого года от 2000 до 2099 (включительно), вызывая статический метод `is_leap_year` и выводя результат на экран.

Пример использования:

```
v = LeapYearChecker.is_leap_year(1999)
```

Вывод (первые и последние строки):

```
2000 True
2001 False
...
2098 False
2099 False
```

2. Написать программу, которая создаёт класс `PrimeChecker` для определения простого числа. В классе должен быть статический метод `is_prime` и возвращать `True`, если число простое, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PrimeChecker`.

(b) Создайте **статический** метод `is_prime`, который принимает число в качестве аргумента и проверяет, является ли число простым. Простое число делится только на 1 и на само себя.

(c) Используйте цикл для проверки каждого числа от 1 до 100 (включительно), вызывая статический метод `is_prime` и выводя результат на экран.

Пример использования:

```
v = PrimeChecker.is_prime(17)
```

Вывод (первые и последние строки):

```
1 False
2 True
3 True
...
98 False
99 False
100 False
```

3. Написать программу, которая создаёт класс `EvenChecker` для определения чётности числа. В классе должен быть статический метод `is_even` и возвращать `True`, если число чётное, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 50 и вывода результата на экран.

Инструкции:

(a) Создайте класс `EvenChecker`.

(b) Создайте **статический** метод `is_even`, который принимает число в качестве аргумента и проверяет, является ли число чётным.

(c) Используйте цикл для проверки каждого числа от 1 до 50 (включительно), вызывая статический метод `is_even` и выводя результат на экран.

Пример использования:

```
v = EvenChecker.is_even(25)
```

Вывод (первые и последние строки):

```
1 False
2 True
3 False
...
48 True
49 False
50 True
```

4. Написать программу, которая создаёт класс `SquareChecker` для определения квадратного числа. В классе должен быть статический метод `is_square` и возвращать `True`, если число является квадратом целого числа, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `SquareChecker`.

(b) Создайте **статический** метод `is_square`, который принимает число в качестве аргумента и проверяет, является ли число квадратом целого числа.

(c) Используйте цикл для проверки каждого числа от 1 до 100 (включительно), вызывая статический метод `is_square` и выводя результат на экран.

Пример использования:

```
v = SquareChecker.is_square(36)
```

Вывод (первые и последние строки):

```
1 True
2 False
3 False
...
99 False
100 True
```

5. Написать программу, которая создаёт класс `FactorialCalculator` для вычисления факториала числа. В классе должен быть статический метод `factorial` и возвращать факториал числа. Программа также должна использовать цикл для вычисления факториала каждого числа от 1 до 10 и вывода результата на экран.

Инструкции:

(a) Создайте класс `FactorialCalculator`.

(b) Создайте **статический** метод `factorial`, который принимает число в качестве аргумента и возвращает его факториал.

(c) Используйте цикл для вычисления факториала каждого числа от 1 до 10 (включительно), вызывая статический метод `factorial` и выводя результат на экран.

Пример использования:

```
v = FactorialCalculator.factorial(5)
```

Вывод (первые и последние строки):

```
1 1
2 2
3 6
...
9 362880
10 3628800
```

6. Написать программу, которая создаёт класс `PalindromeChecker` для определения палиндрома числа. В классе должен быть статический метод `is_palindrome` и возвращать `True`, если число является палиндромом, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 100 до 200 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PalindromeChecker`.

(b) Создайте **статический** метод `is_palindrome`, который принимает число в качестве аргумента и проверяет, является ли число палиндромом (читается одинаково слева направо и справа налево).

(c) Используйте цикл для проверки каждого числа от 100 до 200 (включительно), вызывая статический метод `is_palindrome` и выводя результат на экран.

Пример использования:

```
v = PalindromeChecker.is_palindrome(121)
```

Вывод (первые и последние строки):

```
100 False
101 True
102 False
...
199 False
200 False
```

7. Написать программу, которая создаёт класс `ArmstrongChecker` для определения числа Армстронга. В классе должен быть статический метод `is_armstrong` и возвращать `True`, если число является числом Армстронга, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 100 до 500 и вывода результата на экран.

Инструкции:

(a) Создайте класс `ArmstrongChecker`.

(b) Создайте **статический** метод `is_armstrong`, который принимает число в качестве аргумента и проверяет, является ли число числом Армстронга (сумма цифр в степени, равной количеству цифр, равна самому числу).

(c) Используйте цикл для проверки каждого числа от 100 до 500 (включительно), вызывая статический метод `is_armstrong` и выводя результат на экран.

Пример использования:

```
v = ArmstrongChecker.is_armstrong(153)
```

Вывод (первые и последние строки):

```
100 False
101 False
102 False
...
499 False
500 False
```

8. Написать программу, которая создаёт класс `PerfectNumberChecker` для определения совершенного числа. В классе должен быть статический метод `is_perfect` и возвращать `True`, если число является совершенным, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 1000 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PerfectNumberChecker`.

(b) Создайте **статический** метод `is_perfect`, который принимает число в качестве аргумента и проверяет, является ли число совершенным (сумма делителей равна числу).

(c) Используйте цикл для проверки каждого числа от 1 до 1000 (включительно), вызывая статический метод `is_perfect` и выводя результат на экран.

Пример использования:

```
v = PerfectNumberChecker.is_perfect(28)
```

Вывод (первые и последние строки):

```
1 False
2 False
3 False
...
998 False
999 False
1000 False
```

9. Написать программу, которая создаёт класс `FibonacciChecker` для проверки числа Фибоначчи. В классе должен быть статический метод `is_fibonacci` и возвращать `True`, если число является числом Фибоначчи, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `FibonacciChecker`.

(b) Создайте **статический** метод `is_fibonacci`, который принимает число в качестве аргумента и проверяет, является ли число числом Фибоначчи.

(c) Используйте цикл для проверки каждого числа от 1 до 100 (включительно), вызывая статический метод `is_fibonacci` и выводя результат на экран.

Пример использования:

```
v = FibonacciChecker.is_fibonacci(21)
```

Вывод (первые и последние строки):

```
1 True
2 True
3 True
...
98 False
99 False
100 False
```

10. Написать программу, которая создаёт класс `PowerOfTwoChecker` для проверки степени двойки. В классе должен быть статический метод `is_power_of_two` и возвращать `True`, если число является степенью двойки, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 128 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PowerOfTwoChecker`.

(b) Создайте **статический** метод `is_power_of_two`, который принимает число в качестве аргумента и проверяет, является ли число степенью двойки.

(c) Используйте цикл для проверки каждого числа от 1 до 128 (включительно), вызывая статический метод `is_power_of_two` и выводя результат на экран.

Пример использования:

```
v = PowerOfTwoChecker.is_power_of_two(64)
```

Вывод (первые и последние строки):

```
1 True
2 True
3 False
...
127 False
128 True
```

11. Написать программу, которая создаёт класс `SumOfDigitsCalculator` для вычисления суммы цифр числа. В классе должен быть статический метод `sum_of_digits` и возвращать сумму цифр. Программа также должна использовать цикл для вычисления суммы цифр каждого числа от 1 до 50 и вывода результата на экран.

Инструкции:

(a) Создайте класс `SumOfDigitsCalculator`.

(b) Создайте **статический** метод `sum_of_digits`, который принимает число в качестве аргумента и возвращает сумму его цифр.

(c) Используйте цикл для вычисления суммы цифр каждого числа от 1 до 50 (включительно), вызывая статический метод `sum_of_digits` и выводя результат на экран.

Пример использования:

```
v = SumOfDigitsCalculator.sum_of_digits(123)
```

Вывод (первые и последние строки):

```
1 1
2 2
3 3
...
49 13
50 5
```

12. Написать программу, которая создаёт класс `PrimeSumCalculator` для вычисления суммы простых чисел в диапазоне. В классе должен быть статический метод `sum_of_primes` и возвращать сумму простых чисел в заданном диапазоне. Программа также должна использовать цикл для вычисления суммы простых чисел от 1 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PrimeSumCalculator`.

(b) Создайте **статический** метод `sum_of_primes`, который принимает два аргумента (начало и конец диапазона) и возвращает сумму простых чисел в этом диапазоне.

(c) Используйте метод для вычисления суммы простых чисел от 1 до 100 и выведите результат.

Пример использования:

```
v = PrimeSumCalculator.sum_of_primes(1, 10)
```

Вывод:

```
Сумма простых чисел от 1 до 100: 1060
```

13. Написать программу, которая создаёт класс `DigitCountCalculator` для подсчёта количества цифр в числе. В классе должен быть статический метод `digit_count` и возвращать количество цифр. Программа также должна использовать цикл для подсчёта цифр каждого числа от 1 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `DigitCountCalculator`.

(b) Создайте **статический** метод `digit_count`, который принимает число в качестве аргумента и возвращает количество его цифр.

(c) Используйте цикл для подсчёта цифр каждого числа от 1 до 100 (включительно), вызывая статический метод `digit_count` и выводя результат на экран.

Пример использования:

```
v = DigitCountCalculator.digit_count(12345)
```

Вывод (первые и последние строки):

```
1 1
2 1
3 1
...
99 2
100 3
```

14. Написать программу, которая создаёт класс `BinaryConverter` для преобразования числа в двоичное представление. В классе должен быть статический метод `to_binary` и возвращать строку с двоичным представлением числа. Программа также должна использовать цикл для преобразования каждого числа от 1 до 16 и вывода результата на экран.

Инструкции:

(a) Создайте класс `BinaryConverter`.

(b) Создайте **статический** метод `to_binary`, который принимает число в качестве аргумента и возвращает его двоичное представление в виде строки.

(c) Используйте цикл для преобразования каждого числа от 1 до 16 (включительно), вызывая статический метод `to_binary` и выводя результат на экран.

Пример использования:

```
v = BinaryConverter.to_binary(10)
```

Вывод (первые и последние строки):

```
1 1
2 10
3 11
...
15 1111
16 10000
```

15. Написать программу, которая создаёт класс `HexConverter` для преобразования числа в шестнадцатеричное представление. В классе должен быть статический метод `to_hex` и возвращать строку с шестнадцатеричным представлением числа. Программа также должна использовать цикл для преобразования каждого числа от 1 до 20 и вывода результата на экран.

Инструкции:

(a) Создайте класс `HexConverter`.

(b) Создайте **статический** метод `to_hex`, который принимает число в качестве аргумента и возвращает его шестнадцатеричное представление в виде строки.

(c) Используйте цикл для преобразования каждого числа от 1 до 20 (включительно), вызывая статический метод `to_hex` и выводя результат на экран.

Пример использования:

```
v = HexConverter.to_hex(255)
```

Вывод (первые и последние строки):

```
1 1
2 2
3 3
...
19 13
20 14
```

16. Написать программу, которая создаёт класс `DivisorChecker` для проверки делителей числа. В классе должен быть статический метод `get_divisors` и возвращать список делителей числа. Программа также должна использовать цикл для вывода делителей каждого числа от 1 до 20 и вывода результата на экран.

Инструкции:

(a) Создайте класс `DivisorChecker`.

(b) Создайте **статический** метод `get_divisors`, который принимает число в качестве аргумента и возвращает список его делителей.

(c) Используйте цикл для вывода делителей каждого числа от 1 до 20 (включительно), вызывая статический метод `get_divisors` и выводя результат на экран.

Пример использования:

```
v = DivisorChecker.get_divisors(12)
```

Вывод (первые и последние строки):

```
1 [1]
2 [1, 2]
3 [1, 3]
...
19 [1, 19]
20 [1, 2, 4, 5, 10, 20]
```

17. Написать программу, которая создаёт класс `Multiplier` для создания таблицы умножения. В классе должен быть статический метод `multiply_table` и выводить таблицу умножения для заданного числа. Программа также должна использовать цикл для вывода таблицы умножения для чисел от 1 до 10 и вывода результата на экран.

Инструкции:

(a) Создайте класс `Multiplier`.

(b) Создайте **статический** метод `multiply_table`, который принимает число в качестве аргумента и выводит таблицу умножения для этого числа от 1 до 10.

(c) Используйте цикл для вывода таблицы умножения для чисел от 1 до 10 (включительно), вызывая статический метод `multiply_table` и выводя результат на экран.

Пример использования:

```
Multiplier.multiply_table(5)
```

Вывод (для числа 5):

```
5 * 1 = 5
5 * 2 = 10
...
5 * 10 = 50
```

18. Написать программу, которая создаёт класс `GCDCalculator` для вычисления НОД двух чисел. В классе должен быть статический метод `gcd` и возвращать наибольший общий делитель. Программа также должна использовать цикл для вычисления НОД чисел (1,1), (2,4), (3,9), ..., (10,100) и вывода результата на экран.

Инструкции:

(a) Создайте класс `GCDCalculator`.

(b) Создайте **статический** метод `gcd`, который принимает два числа в качестве аргументов и возвращает их наибольший общий делитель.

(c) Используйте цикл для вычисления НОД пар чисел (1,1), (2,4), (3,9), ..., (10,100), вызывая статический метод `gcd` и выводя результат на экран.

Пример использования:

```
v = GCDCalculator.gcd(48, 18)
```

Вывод:

```
НОД(1, 1) = 1
НОД(2, 4) = 2
НОД(3, 9) = 3
...
НОД(10, 100) = 10
```

19. Написать программу, которая создаёт класс `LCMCalculator` для вычисления НОК двух чисел. В классе должен быть статический метод `lcm` и возвращать наименьшее общее кратное. Программа также должна использовать цикл для вычисления НОК чисел (1,1), (2,3), (3,5), ..., (10,11) и вывода результата на экран.

Инструкции:

(a) Создайте класс `LCMCalculator`.

(b) Создайте **статический** метод `lcm`, который принимает два числа в качестве аргументов и возвращает их наименьшее общее кратное.

(c) Используйте цикл для вычисления НОК пар чисел (1,1), (2,3), (3,5), ..., (10,11), вызывая статический метод `lcm` и выводя результат на экран.

Пример использования:

```
v = LCMCalculator.lcm(4, 6)
```

Вывод:

```
НОК(1, 1) = 1
НОК(2, 3) = 6
НОК(3, 5) = 15
...
НОК(10, 11) = 110
```

20. Написать программу, которая создаёт класс `DigitReverse` для разворота цифр числа. В классе должен быть статический метод `reverse_digits` и возвращать число с обратным порядком цифр. Программа также должна использовать цикл для разворота каждого числа от 10 до 20 и вывода результата на экран.

Инструкции:

(a) Создайте класс `DigitReverse`.

(b) Создайте **статический** метод `reverse_digits`, который принимает число в качестве аргумента и возвращает число с обратным порядком цифр.

(c) Используйте цикл для разворота каждого числа от 10 до 20 (включительно), вызывая статический метод `reverse_digits` и выводя результат на экран.

Пример использования:

```
v = DigitReverse.reverse_digits(123)
```

Вывод:

```
10 1
11 11
12 21
13 31
...
19 91
20 2
```

21. Написать программу, которая создаёт класс `NumberTypeChecker` для определения типа числа (положительное/отрицательное/ноль). В классе должен быть статический метод `check_number_type` и возвращать строку с типом числа. Программа также должна использовать цикл для проверки чисел [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5] и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberTypeChecker`.

(b) Создайте **статический** метод `check_number_type`, который принимает число в качестве аргумента и возвращает строку "positive "negative"или "zero".

(c) Используйте цикл для проверки чисел [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], вызывая статический метод `check_number_type` и выводя результат на экран.

Пример использования:

```
v = NumberTypeChecker.check_number_type(-7)
```

Вывод:

```
-5 negative
-4 negative
-3 negative
-2 negative
-1 negative
0 zero
1 positive
2 positive
3 positive
4 positive
5 positive
```

22. Написать программу, которая создаёт класс `FactorialChecker` для проверки факториала числа. В классе должен быть статический метод `is_factorial` и возвращать `True`, если число является факториалом какого-либо числа, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 120 и вывода результата на экран.

Инструкции:

(a) Создайте класс `FactorialChecker`.

(b) Создайте **статический** метод `is_factorial`, который принимает число в качестве аргумента и проверяет, является ли число факториалом какого-либо числа.

(c) Используйте цикл для проверки каждого числа от 1 до 120 (включительно), вызывая статический метод `is_factorial` и выводя результат на экран.

Пример использования:

```
v = FactorialChecker.is_factorial(24)
```

Вывод (первые и последние строки):

```
1 True
2 True
3 False
...
119 False
120 True
```

23. Написать программу, которая создаёт класс `PowerChecker` для проверки степени числа. В классе должен быть статический метод `is_power` и возвращать `True`, если число является степенью заданного основания, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 1 до 100 относительно основания 3 и вывода результата на экран.

Инструкции:

(a) Создайте класс `PowerChecker`.

(b) Создайте **статический** метод `is_power`, который принимает число и основание в качестве аргументов и проверяет, является ли число степенью основания.

(c) Используйте цикл для проверки каждого числа от 1 до 100 (включительно) относительно основания 3, вызывая статический метод `is_power` и выводя результат на экран.

Пример использования:

```
v = PowerChecker.is_power(81, 3)
```

Вывод (первые и последние строки):

```
1 True
2 False
3 True
...
99 False
100 False
```

24. Написать программу, которая создаёт класс `DigitProductCalculator` для вычисления произведения цифр числа. В классе должен быть статический метод `digit_product` и возвращать произведение цифр. Программа также должна использовать цикл для вычисления произведения цифр каждого числа от 1 до 50 и вывода результата на экран.

Инструкции:

(a) Создайте класс `DigitProductCalculator`.

(b) Создайте **статический** метод `digit_product`, который принимает число в качестве аргумента и возвращает произведение его цифр.

(c) Используйте цикл для вычисления произведения цифр каждого числа от 1 до 50 (включительно), вызывая статический метод `digit_product` и выводя результат на экран.

Пример использования:

```
v = DigitProductCalculator.digit_product(123)
```

Вывод (первые и последние строки):

```
1 1
2 2
3 3
...
49 36
50 0
```

25. Написать программу, которая создаёт класс `NumberLengthChecker` для проверки длины числа. В классе должен быть статический метод `get_length` и возвращать количество цифр в числе. Программа также должна использовать цикл для проверки длины каждого числа от 1 до 1000 с шагом 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberLengthChecker`.

(b) Создайте **статический** метод `get_length`, который принимает число в качестве аргумента и возвращает количество его цифр.

(c) Используйте цикл для проверки длины чисел 1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, вызывая статический метод `get_length` и выводя результат на экран.

Пример использования:

```
v = NumberLengthChecker.get_length(12345)
```

Вывод:

```
1 1
100 3
200 3
300 3
400 3
500 3
600 3
700 3
800 3
900 3
1000 4
```

26. Написать программу, которая создаёт класс `NumberSquareSumCalculator` для вычисления суммы квадратов чисел. В классе должен быть статический метод `square_sum` и возвращать сумму квадратов чисел в диапазоне. Программа также должна использовать метод для вычисления суммы квадратов чисел от 1 до 10 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberSquareSumCalculator`.

(b) Создайте **статический** метод `square_sum`, который принимает два аргумента (начало и конец диапазона) и возвращает сумму квадратов чисел в этом диапазоне.

(c) Используйте метод для вычисления суммы квадратов чисел от 1 до 10 и выведите результат.

Пример использования:

```
v = NumberSquareSumCalculator.square_sum(1, 3)
```

Вывод:

```
Сумма квадратов чисел от 1 до 10: 385
```

27. Написать программу, которая создаёт класс `NumberCubeSumCalculator` для вычисления суммы кубов чисел. В классе должен быть статический метод `cube_sum` и возвращать сумму кубов чисел в диапазоне. Программа также должна использовать метод для вычисления суммы кубов чисел от 1 до 10 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberCubeSumCalculator`.

(b) Создайте **статический** метод `cube_sum`, который принимает два аргумента (начало и конец диапазона) и возвращает сумму кубов чисел в этом диапазоне.

(c) Используйте метод для вычисления суммы кубов чисел от 1 до 10 и выведите результат.

Пример использования:

```
v = NumberCubeSumCalculator.cube_sum(1, 3)
```

Вывод:

```
Сумма кубов чисел от 1 до 10: 3025
```

28. Написать программу, которая создаёт класс `NumberRangeChecker` для проверки числа на принадлежность диапазону. В классе должен быть статический метод `in_range` и возвращать `True`, если число находится в заданном диапазоне, и `False` в противном случае. Программа также должна использовать цикл для проверки чисел от -5 до 5 на принадлежность диапазону [0, 10] и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberRangeChecker`.

(b) Создайте **статический** метод `in_range`, который принимает число, начало и конец диапазона и проверяет, находится ли число в этом диапазоне.

(c) Используйте цикл для проверки чисел от -5 до 5 (включительно) на принадлежность диапазону [0, 10], вызывая статический метод `in_range` и выводя результат на экран.

Пример использования:

```
v = NumberRangeChecker.in_range(5, 0, 10)
```

Вывод:

```
-5 False
-4 False
-3 False
-2 False
-1 False
0 True
1 True
2 True
3 True
4 True
5 True
```

29. Написать программу, которая создаёт класс `NumberSignChecker` для проверки знака числа. В классе должен быть статический метод `get_sign` и возвращать строку с знаком числа (+, - или 0). Программа также должна использовать цикл для проверки чисел [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5] и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberSignChecker`.

(b) Создайте **статический** метод `get_sign`, который принимает число в качестве аргумента и возвращает строку с его знаком (+, - или 0).

(c) Используйте цикл для проверки чисел [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5], вызывая статический метод `get_sign` и выводя результат на экран.

Пример использования:

```
v = NumberSignChecker.get_sign(-7)
```

Вывод:

```
-5 -
-4 -
-3 -
-2 -
-1 -
0 0
1 +
2 +
3 +
4 +
5 +
```

30. Написать программу, которая создаёт класс `NumberPalindromeChecker` для проверки палиндрома числа. В классе должен быть статический метод `is_palindrome` и возвращать `True`, если число является палиндромом, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 100 до 150 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberPalindromeChecker`.

(b) Создайте **статический** метод `is_palindrome`, который принимает число в качестве аргумента и проверяет, является ли число палиндромом.

(c) Используйте цикл для проверки каждого числа от 100 до 150 (включительно), вызывая статический метод `is_palindrome` и выводя результат на экран.

Пример использования:

```
v = NumberPalindromeChecker.is_palindrome(121)
```

Вывод (первые и последние строки):

```
100 False
101 True
102 False
...
149 False
150 False
```

31. Написать программу, которая создаёт класс `NumberAscendingChecker` для проверки, что цифры числа идут в порядке возрастания. В классе должен быть статический метод `is_ascending` и возвращать `True`, если цифры числа идут в порядке возрастания, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 10 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberAscendingChecker`.

(b) Создайте **статический** метод `is_ascending`, который принимает число в качестве аргумента и проверяет, идут ли его цифры в порядке возрастания.

(c) Используйте цикл для проверки каждого числа от 10 до 100 (включительно), вызывая статический метод `is_ascending` и выводя результат на экран.

Пример использования:

```
v = NumberAscendingChecker.is_ascending(123)
```

Вывод (первые и последние строки):

```
10 False
11 False
12 True
13 True
...
98 False
99 False
100 False
```

32. Написать программу, которая создаёт класс `NumberDescendingChecker` для проверки, что цифры числа идут в порядке убывания. В классе должен быть статический метод `is_descending` и возвращать `True`, если цифры числа идут в порядке убывания, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 10 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberDescendingChecker`.

(b) Создайте **статический** метод `is_descending`, который принимает число в качестве аргумента и проверяет, идут ли его цифры в порядке убывания.

(c) Используйте цикл для проверки каждого числа от 10 до 100 (включительно), вызывая статический метод `is_descending` и выводя результат на экран.

Пример использования:

```
v = NumberDescendingChecker.is_descending(321)
```

Вывод (первые и последние строки):

```
10 False
11 False
12 False
13 False
...
98 True
99 True
100 False
```

33. Написать программу, которая создаёт класс `NumberPrimeDigitChecker` для проверки, что все цифры числа простые. В классе должен быть статический метод `all_digits_prime` и возвращать `True`, если все цифры числа простые, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 10 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberPrimeDigitChecker`.

(b) Создайте **статический** метод `all_digits_prime`, который принимает число в качестве аргумента и проверяет, являются ли все его цифры простыми числами.

(c) Используйте цикл для проверки каждого числа от 10 до 100 (включительно), вызывая статический метод `all_digits_prime` и выводя результат на экран.

Пример использования:

```
v = NumberPrimeDigitChecker.all_digits_prime(23)
```

Вывод (первые и последние строки):

```
10 False
11 False
12 False
13 False
...
98 False
99 False
100 False
```

34. Написать программу, которая создаёт класс `NumberEvenDigitChecker` для проверки, что все цифры числа чётные. В классе должен быть статический метод `all_digits_even` и возвравать `True`, если все цифры числа чётные, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 10 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberEvenDigitChecker`.

(b) Создайте **статический** метод `all_digits_even`, который принимает число в качестве аргумента и проверяет, являются ли все его цифры чётными.

(c) Используйте цикл для проверки каждого числа от 10 до 100 (включительно), вызывая статический метод `all_digits_even` и выводя результат на экран.

Пример использования:

```
v = NumberEvenDigitChecker.all_digits_even(24)
```

Вывод (первые и последние строки):

```
10 False
11 False
12 False
13 False
...
98 False
99 False
100 False
```

35. Написать программу, которая создаёт класс `NumberOddDigitChecker` для проверки, что все цифры числа нечётные. В классе должен быть статический метод `all_digits_odd` и возвращать `True`, если все цифры числа нечётные, и `False` в противном случае. Программа также должна использовать цикл для проверки каждого числа от 10 до 100 и вывода результата на экран.

Инструкции:

(a) Создайте класс `NumberOddDigitChecker`.

(b) Создайте **статический** метод `all_digits_odd`, который принимает число в качестве аргумента и проверяет, являются ли все его цифры нечётными.

(c) Используйте цикл для проверки каждого числа от 10 до 100 (включительно), вызывая статический метод `all_digits_odd` и выводя результат на экран.

Пример использования:

```
v = NumberOddDigitChecker.all_digits_odd(135)
```

Вывод (первые и последние строки):

```
10 False
11 True
12 False
13 True
```

```
...
98 False
99 True
100 False
```

2.2.3 Задача 3

1. Написать программу на Python, которая создает класс `Person` для представления сотрудника персонала. Класс должен содержать закрытые атрибуты `__name`, `__country`, `__date_of_birth` и метод `calculate_age`. Доступ к атрибутам только через методы-геттеры. Создать экземпляры и вывести информацию о каждом человеке.

Инструкции:

(a) Создайте класс `Person` с методом `__init__`, который принимает имя, страну и дату рождения.

(b) Создайте методы-геттеры: `get_name()`, `get_country()`, `get_date_of_birth()`.

(c) Создайте метод `calculate_age()` для вычисления возраста.

(d) Создайте несколько экземпляров класса `Person`.

(e) Выведите данные каждого человека через методы класса.

Пример использования:

Листинг 1: Пример кода

```
from datetime import date

person1 = Person("Иванов Иван Иванович", "Россия", date(1946, 8, 15))
person2 = Person("Петров Сергей Александрович", "Белоруссия", date(1982, 10,
    22))

print("Персона 1:")
print("Имя: ", person1.get_name())
print("Страна: ", person1.get_country())
print("Дата рождения: ", person1.get_date_of_birth())
print("Возраст: ", person1.calculate_age())

print("Персона 2:")
print("Имя: ", person2.get_name())
print("Страна: ", person2.get_country())
print("Дата рождения: ", person2.get_date_of_birth())
print("Возраст: ", person2.calculate_age())
```

Вывод:

Листинг 2: Ожидаемый вывод

```
Персона 1:
Имя:  Иванов Иван Иванович
Страна:  Россия
Дата рождения:  1946-08-15
Возраст:  77
```

```
Персона 2:
Имя:  Петров Сергей Александрович
Страна:  Белоруссия
Дата рождения:  1982-10-22
Возраст:  41
```

2. Создайте класс `Student` с закрытыми атрибутами `__full_name`, `__enrollment_date`, `__major`. Реализуйте методы-геттеры и метод `study_duration()` для вычисления количества лет с момента зачисления.

Инструкции:

(a) Создайте класс `Student` с методом `__init__`.

(b) Методы-геттеры: `get_full_name()`, `get_enrollment_date()`, `get_major()`.

(c) Метод `study_duration()` вычисляет количество лет с зачисления.

(d) Создайте несколько экземпляров класса.

(e) Выведите данные каждого студента.

Пример использования:

Листинг 3: Пример кода

```
from datetime import date

student1 = Student("Сидоров Алексей", date(2018, 9, 1), "Математика")
student2 = Student("Иванова Мария", date(2021, 9, 1), "Физика")

print("Студент 1:")
print("Имя: ", student1.get_full_name())
print("Направление: ", student1.get_major())
print("Дата зачисления: ", student1.get_enrollment_date())
print("Стаж учёбы: ", student1.study_duration())

print("Студент 2:")
print("Имя: ", student2.get_full_name())
print("Направление: ", student2.get_major())
print("Дата зачисления: ", student2.get_enrollment_date())
print("Стаж учёбы: ", student2.study_duration())
```

Вывод:

Листинг 4: Ожидаемый вывод

```
Студент 1:
Имя:  Сидоров Алексей
Направление:  Математика
Дата зачисления:  2018-09-01
Стаж учёбы:  5
Студент 2:
Имя:  Иванова Мария
Направление:  Физика
Дата зачисления:  2021-09-01
Стаж учёбы:  2
```

3. Создайте класс `Employee` с закрытыми атрибутами `__name`, `__position`, `__hire_date`. Реализуйте методы-геттеры и метод `work_experience()` для вычисления количества лет работы.

Инструкции:

(a) Создайте класс `Employee` с методом `__init__`.

(b) Методы-геттеры: `get_name()`, `get_position()`, `get_hire_date()`.

(c) Метод `work_experience()` вычисляет стаж в годах.

(d) Создайте несколько экземпляров класса.

(e) Выведите данные каждого сотрудника.

Пример использования:

Листинг 5: Пример кода

```
from datetime import date

emp1 = Employee("Кузнецов Дмитрий", "Инженер", date(2010, 5, 10))
emp2 = Employee("Смирнова Ольга", "Менеджер", date(2015, 8, 1))

print("Сотрудник 1:")
print("Имя: ", emp1.get_name())
print("Должность: ", emp1.get_position())
print("Дата приёма: ", emp1.get_hire_date())
print("Стаж: ", emp1.work_experience())

print("Сотрудник 2:")
print("Имя: ", emp2.get_name())
print("Должность: ", emp2.get_position())
print("Дата приёма: ", emp2.get_hire_date())
print("Стаж: ", emp2.work_experience())
```

Вывод:

Листинг 6: Ожидаемый вывод

```
Сотрудник 1:
Имя:  Кузнецов Дмитрий
Должность:  Инженер
Дата приёма:  2010-05-10
Стаж:  17
Сотрудник 2:
Имя:  Смирнова Ольга
Должность:  Менеджер
Дата приёма:  2015-08-01
Стаж:  8
```

4. Создайте класс `Book` с закрытыми атрибутами `__title`, `__author`, `__publish_date`. Реализуйте геттеры и метод `book_age()` для вычисления возраста книги.

Инструкции:

(a) Создайте класс `Book`.
(b) Методы-геттеры: `get_title()`, `get_author()`, `get_publish_date()`.
(c) Метод `book_age()` вычисляет возраст книги.
(d) Создайте экземпляры класса.
(e) Выведите данные каждой книги.

Пример использования:

Листинг 7: Пример кода

```
from datetime import date

book1 = Book("Программирование на Python", "Иванов И.И.", date(2015, 3, 10))
book2 = Book("Алгебра", "Петров П.П.", date(2000, 9, 1))

print("Книга 1:")
print("Название: ", book1.get_title())
print("Автор: ", book1.get_author())
print("Дата публикации: ", book1.get_publish_date())
print("Возраст книги: ", book1.book_age())

print("Книга 2:")
print("Название: ", book2.get_title())
print("Автор: ", book2.get_author())
print("Дата публикации: ", book2.get_publish_date())
print("Возраст книги: ", book2.book_age())
```

Вывод:

Листинг 8: Ожидаемый вывод

```
Книга 1:
Название:  Программирование на Python
Автор:  Иванов И.И.
Дата публикации:  2015-03-10
Возраст книги:  8
Книга 2:
Название:  Алгебра
Автор:  Петров П.П.
Дата публикации:  2000-09-01
Возраст книги:  23
```

5. Создайте класс `Car` с закрытыми атрибутами `__model`, `__manufacturer`, `__production_date`. Геттеры и метод `car_age()` для вычисления возраста автомобиля.

Инструкции:

(a) Создайте класс `Car`.
(b) Методы-геттеры: `get_model()`, `get_manufacturer()`, `get_production_date()`.
(c) Метод `car_age()` вычисляет возраст автомобиля.
(d) Создайте экземпляры класса.
(e) Выведите данные каждого автомобиля.

Пример использования:

Листинг 9: Пример кода

```
from datetime import date

car1 = Car("Camry", "Toyota", date(2012, 6, 15))
car2 = Car("Focus", "Ford", date(2018, 4, 20))

print("Автомобиль 1:")
print("Модель: ", car1.get_model())
print("Производитель: ", car1.get_manufacturer())
print("Дата выпуска: ", car1.get_production_date())
print("Возраст авто: ", car1.car_age())

print("Автомобиль 2:")
print("Модель: ", car2.get_model())
print("Производитель: ", car2.get_manufacturer())
print("Дата выпуска: ", car2.get_production_date())
print("Возраст авто: ", car2.car_age())
```

Вывод:

Листинг 10: Ожидаемый вывод

```
Автомобиль 1:
Модель:  Camry
Производитель:  Toyota
Дата выпуска:  2012-06-15
Возраст авто:  11
Автомобиль 2:
Модель:  Focus
Производитель:  Ford
Дата выпуска:  2018-04-20
Возраст авто:  5
```

6. Создайте класс `Pet` с закрытыми атрибутами `__name`, `__species`, `__birth_date`. Реализуйте методы-геттеры и метод `pet_age()` для вычисления возраста питомца. Создайте несколько экземпляров и выведите их данные.

Инструкции:

(a) Создайте класс `Pet` с методом `__init__`.

(b) Методы-геттеры: `get_name()`, `get_species()`, `get_birth_date()`.

(c) Метод `pet_age()` вычисляет возраст питомца в годах.

(d) Создайте несколько экземпляров класса.

(e) Выведите данные каждого питомца через методы класса.

Пример использования:

Листинг 11: Пример кода

```
from datetime import date

pet1 = Pet("Барсик", "Кошка", date(2018, 5, 12))
pet2 = Pet("Рекс", "Собака", date(2015, 8, 1))

print("Питомец 1:")
print("Имя: ", pet1.get_name())
print("Вид: ", pet1.get_species())
print("Дата рождения: ", pet1.get_birth_date())
print("Возраст: ", pet1.pet_age())

print("Питомец 2:")
print("Имя: ", pet2.get_name())
print("Вид: ", pet2.get_species())
print("Дата рождения: ", pet2.get_birth_date())
print("Возраст: ", pet2.pet_age())
```

Вывод:

Листинг 12: Ожидаемый вывод

```
Питомец 1:
Имя:  Барсик
Вид:  Кошка
Дата рождения:  2018-05-12
Возраст:  7
Питомец 2:
Имя:  Рекс
Вид:  Собака
Дата рождения:  2015-08-01
Возраст:  10
```

7. Создайте класс `Membership` с закрытыми атрибутами `__member_name`, `__membership_type`, `__join_date`. Реализуйте методы-геттеры и метод `membership_duration()` для вычисления длительности членства в годах.

Инструкции:

(a) Создайте класс `Membership`.

(b) Методы-геттеры: `get_member_name()`, `get_membership_type()`, `get_join_date()`.

(c) Метод `membership_duration()` вычисляет длительность членства в годах.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого участника.

Пример использования:

Листинг 13: Пример кода

```
from datetime import date

member1 = Membership("Иванов Иван", "Золотой", date(2018, 3, 15))
member2 = Membership("Петров Петр", "Серебряный", date(2020, 6, 1))
```

```
print("Член 1:")
print("Имя: ", member1.get_member_name())
print("Тип членства: ", member1.get_membership_type())
print("Дата вступления: ", member1.get_join_date())
print("Длительность членства: ", member1.membership_duration())

print("Член 2:")
print("Имя: ", member2.get_member_name())
print("Тип членства: ", member2.get_membership_type())
print("Дата вступления: ", member2.get_join_date())
print("Длительность членства: ", member2.membership_duration())
```

Вывод:

Листинг 14: Ожидаемый вывод

```
Член 1:
Имя:  Иванов Иван
Тип членства:  Золотой
Дата вступления:  2018-03-15
Длительность членства:  5
Член 2:
Имя:  Петров Петр
Тип членства:  Серебряный
Дата вступления:  2020-06-01
Длительность членства:  3
```

8. Создайте класс `Event` с закрытыми атрибутами `__event_name`, `__location`, `__event_date`. Реализуйте методы-геттеры и метод `days_until_event()` для вычисления количества дней до события.

Инструкции:

(a) Создайте класс `Event`.

(b) Методы-геттеры: `get_event_name()`, `get_location()`, `get_event_date()`.

(c) Метод `days_until_event()` вычисляет дни до события.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого события.

Пример использования:

Листинг 15: Пример кода

```
from datetime import date

event1 = Event("Концерт", "Стадион", date(2025, 12, 1))
event2 = Event("Выставка", "Музей", date(2025, 11, 20))

print("Событие 1:")
print("Название: ", event1.get_event_name())
print("Место: ", event1.get_location())
print("Дата: ", event1.get_event_date())
```

```
print("Дней до события: ", event1.days_until_event())

print("Событие 2:")
print("Название: ", event2.get_event_name())
print("Место: ", event2.get_location())
print("Дата: ", event2.get_event_date())
print("Дней до события: ", event2.days_until_event())
```

Вывод:

Листинг 16: Ожидаемый вывод

```
Событие 1:
Название:  Концерт
Место:  Стадион
Дата:  2025-12-01
Дней до события:  112
Событие 2:
Название:  Выставка
Место:  Музей
Дата:  2025-11-20
Дней до события:  101
```

9. Создайте класс `Course` с закрытыми атрибутами `__course_name`, `__start_date`, `__duration_weeks`. Реализуйте методы-геттеры и метод `weeks_elapsed()` для вычисления прошедших недель с начала курса.

Инструкции:

(a) Создайте класс `Course`.
(b) Методы-геттеры: `get_course_name()`, `get_start_date()`, `get_duration_weeks()`.
(c) Метод `weeks_elapsed()` вычисляет количество прошедших недель.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждого курса.

Пример использования:

Листинг 17: Пример кода

```
from datetime import date

course1 = Course("Python", date(2025, 1, 1), 12)
course2 = Course("Алгебра", date(2025, 2, 1), 10)

print("Курс 1:")
print("Название: ", course1.get_course_name())
print("Дата начала: ", course1.get_start_date())
print("Продолжительность (недель): ", course1.get_duration_weeks())
print("Прошло недель: ", course1.weeks_elapsed())

print("Курс 2:")
print("Название: ", course2.get_course_name())
print("Дата начала: ", course2.get_start_date())
print("Продолжительность (недель): ", course2.get_duration_weeks())
print("Прошло недель: ", course2.weeks_elapsed())
```

Вывод:

Листинг 18: Ожидаемый вывод

```
Курс 1:
Название:  Python
Дата начала:  2025-01-01
Продолжительность (недель):  12
Прошло недель:  36
Курс 2:
Название:  Алгебра
Дата начала:  2025-02-01
Продолжительность (недель):  10
Прошло недель:  31
```

10. Создайте класс `Subscription` с закрытыми атрибутами `__user`, `__plan`, `__start_date`. Реализуйте методы-геттеры и метод `subscription_age()` для вычисления возраста подписки в годах.

Инструкции:

(a) Создайте класс `Subscription`.

(b) Методы-геттеры: `get_user()`, `get_plan()`, `get_start_date()`.

(c) Метод `subscription_age()` вычисляет возраст подписки.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждой подписки.

Пример использования:

Листинг 19: Пример кода

```
from datetime import date

sub1 = Subscription("Иванов И.", "Premium", date(2021, 3, 1))
sub2 = Subscription("Петров П.", "Basic", date(2022, 7, 15))

print("Подписка 1:")
print("Пользователь: ", sub1.get_user())
print("План: ", sub1.get_plan())
print("Дата начала: ", sub1.get_start_date())
print("Возраст подписки: ", sub1.subscription_age())

print("Подписка 2:")
print("Пользователь: ", sub2.get_user())
print("План: ", sub2.get_plan())
print("Дата начала: ", sub2.get_start_date())
print("Возраст подписки: ", sub2.subscription_age())
```

Вывод:

Листинг 20: Ожидаемый вывод

```
Подписка 1:
Пользователь:  Иванов И.
План:  Premium
Дата начала:  2021-03-01
Возраст подписки:  4
Подписка 2:
Пользователь:  Петров П.
План:  Basic
Дата начала:  2022-07-15
Возраст подписки:  3
```

11. Создайте класс `Flight` с закрытыми атрибутами `__flight_number`, `__departure_date`, `__destination`. Реализуйте методы-геттеры и метод `days_until_departure()` для вычисления количества дней до вылета.

Инструкции:

(a) Создайте класс `Flight`.

(b) Методы-геттеры: `get_flight_number()`, `get_departure_date()`, `get_destination()`.

(c) Метод `days_until_departure()` вычисляет количество дней до вылета.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого рейса.

Пример использования:

Листинг 21: Пример кода

```
from datetime import date

flight1 = Flight("SU123", date(2025, 10, 15), "Москва")
flight2 = Flight("AF456", date(2025, 11, 1), "Париж")

print("Рейс 1:")
print("Номер: ", flight1.get_flight_number())
print("Дата вылета: ", flight1.get_departure_date())
print("Пункт назначения: ", flight1.get_destination())
print("Дней до вылета: ", flight1.days_until_departure())

print("Рейс 2:")
print("Номер: ", flight2.get_flight_number())
print("Дата вылета: ", flight2.get_departure_date())
print("Пункт назначения: ", flight2.get_destination())
print("Дней до вылета: ", flight2.days_until_departure())
```

Вывод:

Листинг 22: Ожидаемый вывод

```
Рейс 1:
Номер:  SU123
Дата вылета:  2025-10-15
Пункт назначения:  Москва
```

```
Дней до вылета:  54
Рейс 2:
Номер:  AF456
Дата вылета:  2025-11-01
Пункт назначения:  Париж
Дней до вылета:  71
```

12. Создайте класс `Project` с закрытыми атрибутами `__project_name`, `__start_date`, `__deadline`. Реализуйте методы-геттеры и метод `days_remaining()` для вычисления количества дней до завершения проекта.

Инструкции:

(a) Создайте класс `Project`.

(b) Методы-геттеры: `get_project_name()`, `get_start_date()`, `get_deadline()`.

(c) Метод `days_remaining()` вычисляет дни до дедлайна.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого проекта.

Пример использования:

Листинг 23: Пример кода

```
from datetime import date

project1 = Project("Разработка сайта", date(2025, 9, 1), date(2025, 12, 1))
project2 = Project("Анализ данных", date(2025, 10, 1), date(2025, 11, 30))

print("Проект 1:")
print("Название: ", project1.get_project_name())
print("Дата начала: ", project1.get_start_date())
print("Дедлайн: ", project1.get_deadline())
print("Дней до завершения: ", project1.days_remaining())

print("Проект 2:")
print("Название: ", project2.get_project_name())
print("Дата начала: ", project2.get_start_date())
print("Дедлайн: ", project2.get_deadline())
print("Дней до завершения: ", project2.days_remaining())
```

Вывод:

Листинг 24: Ожидаемый вывод

```
Проект 1:
Название:  Разработка сайта
Дата начала:  2025-09-01
Дедлайн:  2025-12-01
Дней до завершения:  101
Проект 2:
Название:  Анализ данных
Дата начала:  2025-10-01
Дедлайн:  2025-11-30
Дней до завершения:  91
```

13. Создайте класс `Doctor` с закрытыми атрибутами `__full_name`, `__specialty`, `__birth_date`. Реализуйте методы-геттеры и метод `calculate_age()` для вычисления возраста врача.

Инструкции:

(a) Создайте класс `Doctor`.

(b) Методы-геттеры: `get_full_name()`, `get_specialty()`, `get_birth_date()`.

(c) Метод `calculate_age()` вычисляет возраст.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого врача.

Пример использования:

Листинг 25: Пример кода

```
from datetime import date

doc1 = Doctor("Иванов И.И.", "Терапевт", date(1980, 5, 12))
doc2 = Doctor("Петров П.П.", "Хирург", date(1975, 8, 1))

print("Врач 1:")
print("Имя: ", doc1.get_full_name())
print("Специальность: ", doc1.get_specialty())
print("Дата рождения: ", doc1.get_birth_date())
print("Возраст: ", doc1.calculate_age())

print("Врач 2:")
print("Имя: ", doc2.get_full_name())
print("Специальность: ", doc2.get_specialty())
print("Дата рождения: ", doc2.get_birth_date())
print("Возраст: ", doc2.calculate_age())
```

Вывод:

Листинг 26: Ожидаемый вывод

```
Врач 1:
Имя:  Иванов И.И.
Специальность:  Терапевт
Дата рождения:  1980-05-12
Возраст:  45
Врач 2:
Имя:  Петров П.П.
Специальность:  Хирург
Дата рождения:  1975-08-01
Возраст:  50
```

14. Создайте класс `Patient` с закрытыми атрибутами `__full_name`, `__admission_date`, `__diagnosis`. Реализуйте методы-геттеры и метод `hospital_stay()` для вычисления количества дней пребывания в больнице.

Инструкции:

(a) Создайте класс `Patient`.
(b) Методы-геттеры: `get_full_name()`, `get_admission_date()`, `get_diagnosis()`.
(c) Метод `hospital_stay()` вычисляет дни пребывания.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждого пациента.

Пример использования:

Листинг 27: Пример кода

```
from datetime import date

patient1 = Patient("Сидоров С.С.", date(2025, 9, 1), "ОРВИ")
patient2 = Patient("Кузнецов К.К.", date(2025, 8, 28), "Грипп")

print("Пациент 1:")
print("Имя: ", patient1.get_full_name())
print("Дата госпитализации: ", patient1.get_admission_date())
print("Диагноз: ", patient1.get_diagnosis())
print("Дней в больнице: ", patient1.hospital_stay())

print("Пациент 2:")
print("Имя: ", patient2.get_full_name())
print("Дата госпитализации: ", patient2.get_admission_date())
print("Диагноз: ", patient2.get_diagnosis())
print("Дней в больнице: ", patient2.hospital_stay())
```

Вывод:

Листинг 28: Ожидаемый вывод

```
Пациент 1:
Имя:  Сидоров С.С.
Дата госпитализации:  2025-09-01
Диагноз:  ОРВИ
Дней в больнице:  15
Пациент 2:
Имя:  Кузнецов К.К.
Дата госпитализации:  2025-08-28
Диагноз:  Грипп
Дней в больнице:  19
```

15. Создайте класс `Concert` с закрытыми атрибутами `__artist`, `__venue`, `__concert_date`. Реализуйте методы-геттеры и метод `days_until_concert()`.

Инструкции:

(a) Создайте класс `Concert`.
(b) Методы-геттеры: `get_artist()`, `get_venue()`, `get_concert_date()`.
(c) Метод `days_until_concert()` вычисляет дни до концерта.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждого концерта.

Пример использования:

Листинг 29: Пример кода

```
from datetime import date

concert1 = Concert("Imagine Dragons", "Лужники", date(2025, 10, 10))
concert2 = Concert("Coldplay", "O2 Arena", date(2025, 11, 5))

print("Концерт 1:")
print("Исполнитель: ", concert1.get_artist())
print("Место: ", concert1.get_venue())
print("Дата: ", concert1.get_concert_date())
print("Дней до концерта: ", concert1.days_until_concert())

print("Концерт 2:")
print("Исполнитель: ", concert2.get_artist())
print("Место: ", concert2.get_venue())
print("Дата: ", concert2.get_concert_date())
print("Дней до концерта: ", concert2.days_until_concert())
```

Вывод:

Листинг 30: Ожидаемый вывод

```
Концерт 1:
Исполнитель:  Imagine Dragons
Место:  Лужники
Дата:  2025-10-10
Дней до концерта:  49
Концерт 2:
Исполнитель:  Coldplay
Место:  O2 Arena
Дата:  2025-11-05
Дней до концерта:  75
```

16. Создайте класс `Holiday` с закрытыми атрибутами `__name`, `__country`, `__holiday_date`. Реализуйте методы-геттеры и метод `days_until_holiday()`.

Инструкции:

(a) Создайте класс `Holiday`.

(b) Методы-геттеры: `get_name()`, `get_country()`, `get_holiday_date()`.

(c) Метод `days_until_holiday()` вычисляет дни до праздника.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого праздника.

Пример использования:

Листинг 31: Пример кода

```
from datetime import date
```

```
holiday1 = Holiday("Новый Год", "Россия", date(2026, 1, 1))
holiday2 = Holiday("Рождество", "Германия", date(2025, 12, 25))

print("Праздник 1:")
print("Название: ", holiday1.get_name())
print("Страна: ", holiday1.get_country())
print("Дата: ", holiday1.get_holiday_date())
print("Дней до праздника: ", holiday1.days_until_holiday())

print("Праздник 2:")
print("Название: ", holiday2.get_name())
print("Страна: ", holiday2.get_country())
print("Дата: ", holiday2.get_holiday_date())
print("Дней до праздника: ", holiday2.days_until_holiday())
```

Вывод:

Листинг 32: Ожидаемый вывод

```
Праздник 1:
Название:  Новый Год
Страна:  Россия
Дата:  2026-01-01
Дней до праздника:  83
Праздник 2:
Название:  Рождество
Страна:  Германия
Дата:  2025-12-25
Дней до праздника:  67
```

17. Создайте класс `Employee` с закрытыми атрибутами `__full_name`, `__position`, `__hire_date`. Реализуйте методы-геттеры и метод `years_worked()` для вычисления стажа работы в годах.

Инструкции:

(a) Создайте класс `Employee`.

(b) Методы-геттеры: `get_full_name()`, `get_position()`, `get_hire_date()`.

(c) Метод `years_worked()` вычисляет стаж в годах.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого сотрудника.

Пример использования:

Листинг 33: Пример кода

```
from datetime import date

emp1 = Employee("Иванов И.И.", "Менеджер", date(2015, 4, 1))
emp2 = Employee("Петров П.П.", "Разработчик", date(2018, 7, 15))

print("Сотрудник 1:")
print("Имя: ", emp1.get_full_name())
```

```
print("Должность: ", emp1.get_position())
print("Дата приема: ", emp1.get_hire_date())
print("Стаж: ", emp1.years_worked())

print("Сотрудник 2:")
print("Имя: ", emp2.get_full_name())
print("Должность: ", emp2.get_position())
print("Дата приема: ", emp2.get_hire_date())
print("Стаж: ", emp2.years_worked())
```

Вывод:

Листинг 34: Ожидаемый вывод

```
Сотрудник 1:
Имя:  Иванов И.И.
Должность:  Менеджер
Дата приема:  2015-04-01
Стаж:  10
Сотрудник 2:
Имя:  Петров П.П.
Должность:  Разработчик
Дата приема:  2018-07-15
Стаж:  7
```

18. Создайте класс `LibraryBook` с закрытыми атрибутами `__title`, `__author`, `__publication_date`. Реализуйте методы-геттеры и метод `book_age()` для вычисления возраста книги.

Инструкции:

(a) Создайте класс `LibraryBook`.

(b) Методы-геттеры: `get_title()`, `get_author()`, `get_publication_date()`.

(c) Метод `book_age()` вычисляет возраст книги в годах.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждой книги.

Пример использования:

Листинг 35: Пример кода

```
from datetime import date

book1 = LibraryBook("Война и мир", "Толстой", date(1869, 1, 1))
book2 = LibraryBook("Мастер и Маргарита", "Булгаков", date(1967, 5, 1))

print("Книга 1:")
print("Название: ", book1.get_title())
print("Автор: ", book1.get_author())
print("Дата публикации: ", book1.get_publication_date())
print("Возраст книги: ", book1.book_age())

print("Книга 2:")
print("Название: ", book2.get_title())
```

```
print("Автор: ", book2.get_author())
print("Дата публикации: ", book2.get_publication_date())
print("Возраст книги: ", book2.book_age())
```

Вывод:

Листинг 36: Ожидаемый вывод

```
Книга 1:
Название:  Война и мир
Автор:  Толстой
Дата публикации:  1869-01-01
Возраст книги:  156
Книга 2:
Название:  Мастер и Маргарита
Автор:  Булгаков
Дата публикации:  1967-05-01
Возраст книги:  59
```

19. Создайте класс `Vehicle` с закрытыми атрибутами `__brand`, `__model`, `__manufacture_date`. Реализуйте методы-геттеры и метод `vehicle_age()`.

Инструкции:

(a) Создайте класс `Vehicle`.

(b) Методы-геттеры: `get_brand()`, `get_model()`, `get_manufacture_date()`.

(c) Метод `vehicle_age()` вычисляет возраст транспортного средства.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого транспортного средства.

Пример использования:

Листинг 37: Пример кода

```
from datetime import date

vehicle1 = Vehicle("Toyota", "Camry", date(2015, 5, 1))
vehicle2 = Vehicle("BMW", "X5", date(2018, 3, 10))

print("Транспорт 1:")
print("Марка: ", vehicle1.get_brand())
print("Модель: ", vehicle1.get_model())
print("Дата производства: ", vehicle1.get_manufacture_date())
print("Возраст: ", vehicle1.vehicle_age())

print("Транспорт 2:")
print("Марка: ", vehicle2.get_brand())
print("Модель: ", vehicle2.get_model())
print("Дата производства: ", vehicle2.get_manufacture_date())
print("Возраст: ", vehicle2.vehicle_age())
```

Вывод:

Листинг 38: Ожидаемый вывод

```
Транспорт 1:
Марка:  Toyota
Модель:  Camry
Дата производства:  2015-05-01
Возраст:  10
Транспорт 2:
Марка:  BMW
Модель:  X5
Дата производства:  2018-03-10
Возраст:  7
```

20. Создайте класс `Student` с закрытыми атрибутами `__full_name`, `__enrollment_date`, `__major`. Реализуйте методы-геттеры и метод `study_years()`.

Инструкции:

(a) Создайте класс `Student`.

(b) Методы-геттеры: `get_full_name()`, `get_enrollment_date()`, `get_major()`.

(c) Метод `study_years()` вычисляет количество лет учебы.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого студента.

Пример использования:

Листинг 39: Пример кода

```
from datetime import date

student1 = Student("Иванов И.И.", date(2020, 9, 1), "Математика")
student2 = Student("Петров П.П.", date(2021, 9, 1), "Физика")

print("Студент 1:")
print("Имя: ", student1.get_full_name())
print("Дата зачисления: ", student1.get_enrollment_date())
print("Специальность: ", student1.get_major())
print("Лет учебы: ", student1.study_years())

print("Студент 2:")
print("Имя: ", student2.get_full_name())
print("Дата зачисления: ", student2.get_enrollment_date())
print("Специальность: ", student2.get_major())
print("Лет учебы: ", student2.study_years())
```

Вывод:

Листинг 40: Ожидаемый вывод

```
Студент 1:
Имя:  Иванов И.И.
```

```
Дата зачисления:  2020-09-01
Специальность:  Математика
Лет учебы:  5
Студент 2:
Имя:  Петров П.П.
Дата зачисления:  2021-09-01
Специальность:  Физика
Лет учебы:  4
```

21. Создайте класс `Ticket` с закрытыми атрибутами `__ticket_number`, `__issue_date`, `__valid_until`. Реализуйте методы-геттеры и метод `days_valid()`.

Инструкции:

(a) Создайте класс `Ticket`.

(b) Методы-геттеры: `get_ticket_number()`, `get_issue_date()`, `get_valid_until()`.

(c) Метод `days_valid()` вычисляет дни до окончания действия билета.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого билета.

Пример использования:

Листинг 41: Пример кода

```
from datetime import date

ticket1 = Ticket("A123", date(2025, 9, 1), date(2025, 12, 1))
ticket2 = Ticket("B456", date(2025, 10, 1), date(2026, 1, 1))

print("Билет 1:")
print("Номер: ", ticket1.get_ticket_number())
print("Дата выдачи: ", ticket1.get_issue_date())
print("Действителен до: ", ticket1.get_valid_until())
print("Дней до окончания: ", ticket1.days_valid())

print("Билет 2:")
print("Номер: ", ticket2.get_ticket_number())
print("Дата выдачи: ", ticket2.get_issue_date())
print("Действителен до: ", ticket2.get_valid_until())
print("Дней до окончания: ", ticket2.days_valid())
```

Вывод:

Листинг 42: Ожидаемый вывод

```
Билет 1:
Номер:  A123
Дата выдачи:  2025-09-01
Действителен до:  2025-12-01
Дней до окончания:  91
Билет 2:
Номер:  B456
Дата выдачи:  2025-10-01
Действителен до:  2026-01-01
Дней до окончания:  92
```

22. Создайте класс `Appointment` с закрытыми атрибутами `__client`, `__service`, `__appointment_date`. Реализуйте методы-геттеры и метод `days_until_appointment()`.

Инструкции:

(a) Создайте класс `Appointment`.

(b) Методы-геттеры: `get_client()`, `get_service()`, `get_appointment_date()`.

(c) Метод `days_until_appointment()` вычисляет дни до приёма.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого приёма.

Пример использования:

Листинг 43: Пример кода

```
from datetime import date

app1 = Appointment("Иванов И.", "Массаж", date(2025, 10, 5))
app2 = Appointment("Петров П.", "Стрижка", date(2025, 10, 15))

print("Приём 1:")
print("Клиент: ", app1.get_client())
print("Услуга: ", app1.get_service())
print("Дата: ", app1.get_appointment_date())
print("Дней до приёма: ", app1.days_until_appointment())

print("Приём 2:")
print("Клиент: ", app2.get_client())
print("Услуга: ", app2.get_service())
print("Дата: ", app2.get_appointment_date())
print("Дней до приёма: ", app2.days_until_appointment())
```

Вывод:

Листинг 44: Ожидаемый вывод

```
Приём 1:
Клиент:  Иванов И.
Услуга:  Массаж
Дата:  2025-10-05
Дней до приёма:  44
Приём 2:
Клиент:  Петров П.
Услуга:  Стрижка
Дата:  2025-10-15
Дней до приёма:  54
```

23. Создайте класс `Subscription` с закрытыми атрибутами `__subscriber`, `__start_date`, `__end_date`. Реализуйте методы-геттеры и метод `days_remaining()`.

Инструкции:

(a) Создайте класс `Subscription`.
(b) Методы-геттеры: `get_subscriber()`, `get_start_date()`, `get_end_date()`.
(c) Метод `days_remaining()` вычисляет дни до окончания подписки.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждой подписки.

Пример использования:

Листинг 45: Пример кода

```
from datetime import date

sub1 = Subscription("Иванов И.", date(2025, 1, 1), date(2025, 12, 31))
sub2 = Subscription("Петров П.", date(2025, 6, 1), date(2026, 5, 31))

print("Подписка 1:")
print("Абонент: ", sub1.get_subscriber())
print("Дата начала: ", sub1.get_start_date())
print("Дата окончания: ", sub1.get_end_date())
print("Дней до окончания: ", sub1.days_remaining())

print("Подписка 2:")
print("Абонент: ", sub2.get_subscriber())
print("Дата начала: ", sub2.get_start_date())
print("Дата окончания: ", sub2.get_end_date())
print("Дней до окончания: ", sub2.days_remaining())
```

Вывод:

Листинг 46: Ожидаемый вывод

```
Подписка 1:
Абонент:  Иванов И.
Дата начала:  2025-01-01
Дата окончания:  2025-12-31
Дней до окончания:  113
Подписка 2:
Абонент:  Петров П.
Дата начала:  2025-06-01
Дата окончания:  2026-05-31
Дней до окончания:  245
```

24. Создайте класс `MembershipCard` с закрытыми атрибутами `__owner`, `__issue_date`, `__expiry_date`. Реализуйте методы-геттеры и метод `days_until_expiry()`.

Инструкции:

(a) Создайте класс `MembershipCard`.
(b) Методы-геттеры: `get_owner()`, `get_issue_date()`, `get_expiry_date()`.
(c) Метод `days_until_expiry()` вычисляет дни до истечения действия карты.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждой карты.

Пример использования:

Листинг 47: Пример кода

```
from datetime import date

card1 = MembershipCard("Иванов И.", date(2025, 1, 1), date(2026, 1, 1))
card2 = MembershipCard("Петров П.", date(2025, 5, 1), date(2026, 5, 1))

print("Карта 1:")
print("Владелец: ", card1.get_owner())
print("Дата выдачи: ", card1.get_issue_date())
print("Срок действия: ", card1.get_expiry_date())
print("Дней до окончания: ", card1.days_until_expiry())

print("Карта 2:")
print("Владелец: ", card2.get_owner())
print("Дата выдачи: ", card2.get_issue_date())
print("Срок действия: ", card2.get_expiry_date())
print("Дней до окончания: ", card2.days_until_expiry())
```

Вывод:

Листинг 48: Ожидаемый вывод

```
Карта 1:
Владелец:  Иванов И.
Дата выдачи:  2025-01-01
Срок действия:  2026-01-01
Дней до окончания:  113
Карта 2:
Владелец:  Петров П.
Дата выдачи:  2025-05-01
Срок действия:  2026-05-01
Дней до окончания:  204
```

25. Создайте класс `Event` с закрытыми атрибутами `__title`, `__location`, `__event_date`. Реализуйте методы-геттеры и метод `days_until_event()`.

Инструкции:

(a) Создайте класс `Event`.

(b) Методы-геттеры: `get_title()`, `get_location()`, `get_event_date()`.

(c) Метод `days_until_event()` вычисляет дни до события.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого события.

Пример использования:

Листинг 49: Пример кода

```
from datetime import date
```

```
event1 = Event("Фестиваль науки", "Москва", date(2025, 10, 20))
event2 = Event("Конференция IT", "Санкт-Петербург", date(2025, 11, 10))

print("Событие 1:")
print("Название: ", event1.get_title())
print("Место: ", event1.get_location())
print("Дата: ", event1.get_event_date())
print("Дней до события: ", event1.days_until_event())

print("Событие 2:")
print("Название: ", event2.get_title())
print("Место: ", event2.get_location())
print("Дата: ", event2.get_event_date())
print("Дней до события: ", event2.days_until_event())
```

Вывод:

Листинг 50: Ожидаемый вывод

```
Событие 1:
Название:  Фестиваль науки
Место:  Москва
Дата:  2025-10-20
Дней до события:  59
Событие 2:
Название:  Конференция IT
Место:  Санкт-Петербург
Дата:  2025-11-10
Дней до события:  80
```

26. Создайте класс `CarRental` с закрытыми атрибутами `__client`, `__rental_date`, `__return_date`. Реализуйте методы-геттеры и метод `rental_duration()`.

Инструкции:

(a) Создайте класс `CarRental`.

(b) Методы-геттеры: `get_client()`, `get_rental_date()`, `get_return_date()`.

(c) Метод `rental_duration()` вычисляет длительность аренды в днях.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждой аренды.

Пример использования:

Листинг 51: Пример кода

```
from datetime import date

rental1 = CarRental("Иванов И.", date(2025, 10, 1), date(2025, 10, 10))
rental2 = CarRental("Петров П.", date(2025, 11, 1), date(2025, 11, 5))

print("Аренда 1:")
print("Клиент: ", rental1.get_client())
print("Дата аренды: ", rental1.get_rental_date())
```

```
print("Дата возврата: ", rental1.get_return_date())
print("Длительность аренды: ", rental1.rental_duration())

print("Аренда 2:")
print("Клиент: ", rental2.get_client())
print("Дата аренды: ", rental2.get_rental_date())
print("Дата возврата: ", rental2.get_return_date())
print("Длительность аренды: ", rental2.rental_duration())
```

Вывод:

Листинг 52: Ожидаемый вывод

```
Аренда 1:
Клиент:  Иванов И.
Дата аренды:  2025-10-01
Дата возврата:  2025-10-10
Длительность аренды:  9
Аренда 2:
Клиент:  Петров П.
Дата аренды:  2025-11-01
Дата возврата:  2025-11-05
Длительность аренды:  4
```

27. Создайте класс `Visa` с закрытыми атрибутами `__holder`, `__issue_date`, `__expiry_date`. Реализуйте методы-геттеры и метод `days_until_expiry()`.

Инструкции:

(a) Создайте класс `Visa`.
(b) Методы-геттеры: `get_holder()`, `get_issue_date()`, `get_expiry_date()`.
(c) Метод `days_until_expiry()` вычисляет дни до окончания визы.
(d) Создайте несколько экземпляров.
(e) Выведите данные каждой визы.

Пример использования:

Листинг 53: Пример кода

```
from datetime import date

visa1 = Visa("Иванов И.", date(2025, 1, 1), date(2026, 1, 1))
visa2 = Visa("Петров П.", date(2025, 6, 1), date(2026, 6, 1))

print("Виза 1:")
print("Держатель: ", visa1.get_holder())
print("Дата выдачи: ", visa1.get_issue_date())
print("Дата окончания: ", visa1.get_expiry_date())
print("Дней до окончания: ", visa1.days_until_expiry())

print("Виза 2:")
print("Держатель: ", visa2.get_holder())
print("Дата выдачи: ", visa2.get_issue_date())
print("Дата окончания: ", visa2.get_expiry_date())
print("Дней до окончания: ", visa2.days_until_expiry())
```

Вывод:

Листинг 54: Ожидаемый вывод

```
Виза 1:
Держатель:  Иванов И.
Дата выдачи:  2025-01-01
Дата окончания:  2026-01-01
Дней до окончания:  113
Виза 2:
Держатель:  Петров П.
Дата выдачи:  2025-06-01
Дата окончания:  2026-06-01
Дней до окончания:  204
```

28. Создайте класс `Reservation` с закрытыми атрибутами `__guest`, `__checkin_date`, `__checkout_date`. Реализуйте методы-геттеры и метод `stay_duration()`.

Инструкции:

(a) Создайте класс `Reservation`.

(b) Методы-геттеры: `get_guest()`, `get_checkin_date()`, `get_checkout_date()`.

(c) Метод `stay_duration()` вычисляет продолжительность пребывания в днях.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждой брони.

Пример использования:

Листинг 55: Пример кода

```
from datetime import date

res1 = Reservation("Иванов И.", date(2025, 10, 1), date(2025, 10, 7))
res2 = Reservation("Петров П.", date(2025, 11, 5), date(2025, 11, 12))

print("Бронь 1:")
print("Гость: ", res1.get_guest())
print("Дата заезда: ", res1.get_checkin_date())
print("Дата выезда: ", res1.get_checkout_date())
print("Продолжительность пребывания: ", res1.stay_duration())

print("Бронь 2:")
print("Гость: ", res2.get_guest())
print("Дата заезда: ", res2.get_checkin_date())
print("Дата выезда: ", res2.get_checkout_date())
print("Продолжительность пребывания: ", res2.stay_duration())
```

Вывод:

Листинг 56: Ожидаемый вывод

```
Бронь 1:
Гость:  Иванов И.
```

```
Дата заезда:  2025-10-01
Дата выезда:  2025-10-07
Продолжительность пребывания:  6
Бронь 2:
Гость:  Петров П.
Дата заезда:  2025-11-05
Дата выезда:  2025-11-12
Продолжительность пребывания:  7
```

29. Создайте класс `Conference` с закрытыми атрибутами `__name`, `__city`, `__start_date`. Реализуйте методы-геттеры и метод `days_until_start()`.

Инструкции:

(a) Создайте класс `Conference`.

(b) Методы-геттеры: `get_name()`, `get_city()`, `get_start_date()`.

(c) Метод `days_until_start()` вычисляет дни до начала конференции.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждой конференции.

Пример использования:

Листинг 57: Пример кода

```
from datetime import date

conf1 = Conference("PythonConf", "Москва", date(2025, 10, 20))
conf2 = Conference("DataScience Summit", "Санкт-Петербург", date(2025, 11,
    15))

print("Конференция 1:")
print("Название: ", conf1.get_name())
print("Город: ", conf1.get_city())
print("Дата начала: ", conf1.get_start_date())
print("Дней до начала: ", conf1.days_until_start())

print("Конференция 2:")
print("Название: ", conf2.get_name())
print("Город: ", conf2.get_city())
print("Дата начала: ", conf2.get_start_date())
print("Дней до начала: ", conf2.days_until_start())
```

Вывод:

Листинг 58: Ожидаемый вывод

```
Конференция 1:
Название:  PythonConf
Город:  Москва
Дата начала:  2025-10-20
Дней до начала:  59
Конференция 2:
Название:  DataScience Summit
```

```
Город:  Санкт-Петербург
Дата начала:  2025-11-15
Дней до начала:  85
```

30. Создайте класс `Medication` с закрытыми атрибутами `__name`, `__manufacturer`, `__expiry_date`. Реализуйте методы-геттеры и метод `days_until_expiry()`.

Инструкции:

(a) Создайте класс `Medication`.

(b) Методы-геттеры: `get_name()`, `get_manufacturer()`, `get_expiry_date()`.

(c) Метод `days_until_expiry()` вычисляет дни до окончания срока годности.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого лекарства.

Пример использования:

Листинг 59: Пример кода

```
from datetime import date

med1 = Medication("Парацетамол", "Фармком", date(2026, 1, 1))
med2 = Medication("Ибупрофен", "БиоФарм", date(2025, 12, 1))

print("Лекарство 1:")
print("Название: ", med1.get_name())
print("Производитель: ", med1.get_manufacturer())
print("Срок годности: ", med1.get_expiry_date())
print("Дней до окончания: ", med1.days_until_expiry())

print("Лекарство 2:")
print("Название: ", med2.get_name())
print("Производитель: ", med2.get_manufacturer())
print("Срок годности: ", med2.get_expiry_date())
print("Дней до окончания: ", med2.days_until_expiry())
```

Вывод:

Листинг 60: Ожидаемый вывод

```
Лекарство 1:
Название:  Парацетамол
Производитель:  Фармком
Срок годности:  2026-01-01
Дней до окончания:  113
Лекарство 2:
Название:  Ибупрофен
Производитель:  БиоФарм
Срок годности:  2025-12-01
Дней до окончания:  92
```

31. Создайте класс `Project` с закрытыми атрибутами `__title`, `__start_date`, `__deadline`. Реализуйте методы-геттеры и метод `days_until_deadline()`.

Инструкции:

(a) Создайте класс `Project`.

(b) Методы-геттеры: `get_title()`, `get_start_date()`, `get_deadline()`.

(c) Метод `days_until_deadline()` вычисляет дни до дедлайна.

(d) Создайте несколько экземпляров.

(e) Выведите данные каждого проекта.

Пример использования:

Листинг 61: Пример кода

```
from datetime import date

proj1 = Project("Разработка сайта", date(2025, 9, 1), date(2025, 12, 1))
proj2 = Project("Мобильное приложение", date(2025, 10, 1), date(2026, 1, 15))

print("Проект 1:")
print("Название: ", proj1.get_title())
print("Дата начала: ", proj1.get_start_date())
print("Дедлайн: ", proj1.get_deadline())
print("Дней до дедлайна: ", proj1.days_until_deadline())

print("Проект 2:")
print("Название: ", proj2.get_title())
print("Дата начала: ", proj2.get_start_date())
print("Дедлайн: ", proj2.get_deadline())
print("Дней до дедлайна: ", proj2.days_until_deadline())
```

Вывод:

Листинг 62: Ожидаемый вывод

```
Проект 1:
Название:  Разработка сайта
Дата начала:  2025-09-01
Дедлайн:  2025-12-01
Дней до дедлайна:  91
Проект 2:
Название:  Мобильное приложение
Дата начала:  2025-10-01
Дедлайн:  2026-01-15
Дней до дедлайна:  106
```

2.2.4 Задача 4

1. Написать программу на Python, которая создает абстрактный класс `Shape` для представления геометрической фигуры. Класс должен содержать абстрактные методы `calculate_area` и `calculate_perimeter`, которые вычисляют площадь и периметр фигуры соответственно. Программа также должна создавать дочерние классы `Circle`, `Rectangle` и `Triangle`, которые наследуют от класса `Shape` и реализуют специфические для каждого класса методы вычисления площади и периметра.

   Инструкции:

(a) Создайте абстрактный класс `Shape` (с использованием модуля `abc`) с абстрактными методами `calculate_area()` и `calculate_perimeter()`.

(b) Создайте класс `Circle` с конструктором `__init__(self, radius)`, который принимает радиус окружности в качестве аргумента и сохраняет его в приватном атрибуте `__radius`. Добавьте `@property`-геттер `radius` для получения значения радиуса. Реализуйте методы `calculate_area()` и `calculate_perimeter()` для вычисления площади и периметра окружности.

(c) Создайте класс `Rectangle` с конструктором `__init__(self, length, width)`, который принимает длину и ширину прямоугольника в качестве аргументов и сохраняет их в приватных атрибутах `_length` и `_width`. Добавьте `@property`-геттеры `length` и `width` для получения значений атрибутов. Реализуйте методы `calculate_area()` и `calculate_perimeter()` для вычисления площади и периметра прямоугольника.

(d) Создайте класс `Triangle` с конструктором `__init__(self, base, height, side1, side2, side3)`, который принимает основание, высоту и три стороны треугольника в качестве аргументов и сохраняет их в приватных атрибутах `_base`, `_height`, `_side1`, `_side2` и `_side3`. Добавьте `@property`-геттеры `base`, `height`, `side1`, `side2`, `side3`. Реализуйте методы `calculate_area()` и `calculate_perimeter()` для вычисления площади и периметра треугольника.

(e) Создайте экземпляр каждого класса и вызовите методы `calculate_area()` и `calculate_perimeter()` для вычисления площади и периметра фигуры. Выведите результаты на экран, используя геттеры для доступа к атрибутам.

Пример использования:

```
# Вычисление параметров окружности.
r = 7
circle = Circle(r)
print("Радиус окружности:", circle.radius)
print("Площадь окружности:", circle.calculate_area())
print("Периметр окружности:", circle.calculate_perimeter())
```

Примечание: В этом примере используется библиотека `math` для вычисления числа π и квадратного корня.

Вывод:

```
Радиус окружности: 7
Площадь окружности: 153.93804002589985
Периметр окружности: 43.982297150257104
```

Далее вывод для прямоугольника и треугольника.

2. Написать программу на Python, которая создает абстрактный класс `ElectricalComponent` (с использованием модуля `abc`) для представления электрических элементов. Класс должен содержать абстрактные методы `calculate_power()` и `calculate_energy()`. Программа также должна создавать дочерние классы `Resistor`, `Capacitor` и `Inductor`, которые наследуют от класса `ElectricalComponent` и реализуют специфические для каждого класса методы вычисления мощности и энергии.

Подсказка по формулам:

- `Resistor`: $P = U^2/R$, $E = P \cdot t$
- `Capacitor`: $P = V \cdot I$, $E = 0.5 \cdot C \cdot V^2$
- `Inductor`: $P = L \cdot I^2$, $E = 0.5 \cdot L \cdot I^2$

Инструкции:

(a) Создайте абстрактный класс `ElectricalComponent` с методами `calculate_power()` и `calculate_energy()`, используя модуль `abc`.

(b) Создайте класс `Resistor` с конструктором `__init__(self, voltage, resistance, time)`, который сохраняет приватные атрибуты `__voltage`, `__resistance`, `__time`. Добавьте `@property`-геттеры для всех атрибутов. Реализуйте методы вычисления мощности и энергии.

(c) Создайте класс `Capacitor` с конструктором `__init__(self, voltage, current, capacitance)`, приватными атрибутами `__voltage`, `__current`, `__capacitance` и геттерами. Реализуйте методы.

(d) Создайте класс `Inductor` с конструктором `__init__(self, inductance, current)`, приватными атрибутами `__inductance`, `__current` и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы `calculate_power()` и `calculate_energy()`, используя геттеры для доступа к атрибутам. Выведите результаты на экран.

Пример использования:

```
r = Resistor(10, 5, 10)
print("Сопротивление резистора:", r.resistance)
print("Мощность резистора:", r.calculate_power())
print("Энергия резистора:", r.calculate_energy())
```

Вывод:

```
Сопротивление резистора: 5
Мощность резистора: 20
Энергия резистора: 200
```

Далее вывод для конденсатора и катушки индуктивности.

3. Написать программу на Python, которая создает абстрактный класс `MotionObject` (с использованием модуля `abc`) для представления движущихся тел. Класс должен содержать абстрактные методы `calculate_kinetic_energy()` и `calculate_momentum()`. Программа также должна создавать дочерние классы `LinearBody`, `RotatingBody` и `FallingBody`, которые наследуют от класса `MotionObject` и реализуют специфические для каждого класса методы вычисления кинетической энергии и импульса.

Подсказка по формулам:

- `LinearBody`: $KE = 0.5 \cdot m \cdot v^2$, $p = m \cdot v$
- `RotatingBody`: $KE = 0.5 \cdot I \cdot \omega^2$, $p = I \cdot \omega$
- `FallingBody`: $KE = m \cdot g \cdot h$, $p = m \cdot v$

Инструкции:

(a) Создайте абстрактный класс `MotionObject` с методами `calculate_kinetic_energy()` и `calculate_momentum()`, используя модуль `abc`.

(b) Создайте класс `LinearBody` с конструктором `__init__(self, mass, velocity)`, приватными атрибутами `__mass`, `__velocity` и геттерами. Реализуйте методы.

(c) Создайте класс `RotatingBody` с конструктором `__init__(self, moment_of_inertia, angular_velocity)`, приватными атрибутами `__moment_of_inertia`, `__angular_velocity` и геттерами. Реализуйте методы.

(d) Создайте класс `FallingBody` с конструктором `__init__(self, mass, height, velocity)`, приватными атрибутами `__mass`, `__height`, `__velocity` и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы `calculate_kinetic_energy()` и `calculate_momentum()`, используя геттеры для доступа к атрибутам. Выведите результаты на экран.

Пример использования:

```
body = LinearBody(2, 3)
print("Масса тела:", body.mass)
print("Кинетическая энергия:", body.calculate_kinetic_energy())
print("Импульс:", body.calculate_momentum())
```

Вывод:

```
Масса тела: 2
Кинетическая энергия: 6
Импульс: 6
```

Далее вывод для вращающегося тела и падающего тела.

4. Написать программу на Python, которая создает абстрактный класс `Investment` (с использованием модуля `abc`) для финансовых вложений. Класс должен содержать абстрактные методы `calculate_simple_interest()` и `calculate_total_value()`. Программа также должна создавать дочерние классы `ShortTerm`, `LongTerm` и `CompoundInvestment`, которые наследуют от класса `Investment` и реализуют специфические методы вычисления процентов и итоговой суммы.

Подсказка по формулам:

- `ShortTerm`: $SI = P \cdot R \cdot T/100$, $Total = P + SI$
- `LongTerm`: $SI = P \cdot R \cdot T/100 + 50$, $Total = P + SI$
- `CompoundInvestment`: $Total = P \cdot (1 + R/100)^T$, $SI = Total - P$

Инструкции:

(a) Создайте абстрактный класс `Investment` с методами `calculate_simple_interest()` и `calculate_total_value()`, используя модуль `abc`.

(b) Создайте класс `ShortTerm` с конструктором `__init__(self, principal, rate, time)`, приватными атрибутами `__principal`, `__rate`, `__time` и геттерами. Реализуйте методы.

(c) Создайте класс `LongTerm` с конструктором `__init__(self, principal, rate, time)`, приватными атрибутами `__principal`, `__rate`, `__time` и геттерами. Реализуйте методы.

(d) Создайте класс `CompoundInvestment` с конструктором `__init__(self, principal, rate, time)`, приватными атрибутами `__principal`, `__rate`, `__time` и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы `calculate_simple_interest()` и `calculate_total_value()`, используя геттеры для доступа к атрибутам. Выведите результаты на экран.

Пример использования:

```
inv = ShortTerm(1000, 5, 2)
print("Начальная сумма:", inv.principal)
print("Простой процент:", inv.calculate_simple_interest())
print("Итоговая сумма:", inv.calculate_total_value())
```

Вывод:

```
Начальная сумма: 1000
Простой процент: 100
Итоговая сумма: 1100
```

Далее вывод для долгосрочного и сложного вложения.

5. Написать программу на Python, которая создает абстрактный класс `Solid` (с использованием модуля `abc`) для твердого тела. Класс должен содержать абстрактные методы `calculate_volume()` и `calculate_surface_area()`. Программа также должна создавать дочерние классы `Cube`, `RectangularPrism` и `Cylinder`, которые наследуют от класса `Solid` и реализуют специфические методы вычисления объема и площади поверхности.

Подсказка по формулам:

- `Cube`: $V = a^3$, $S = 6 \cdot a^2$
- `RectangularPrism`: $V = l \cdot w \cdot h$, $S = 2(lw + lh + wh)$
- `Cylinder`: $V = \pi r^2 h$, $S = 2\pi r(r + h)$

Инструкции:

(a) Создайте абстрактный класс `Solid` с методами `calculate_volume()` и `calculate_surface_area()`, используя модуль `abc`.

(b) Создайте класс `Cube` с конструктором `__init__(self, side)`, приватным атрибутом `__side` и геттером. Реализуйте методы.

(c) Создайте класс `RectangularPrism` с конструктором `__init__(self, length, width, height)`, приватными атрибутами `__length`, `__width`, `__height` и геттерами. Реализуйте методы.

(d) Создайте класс `Cylinder` с конструктором `__init__(self, radius, height)`, приватными атрибутами `__radius`, `__height` и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы `calculate_volume()` и `calculate_surface_area()`, используя геттеры. Выведите результаты на экран.

Пример использования:

```
cube = Cube(3)
print("Сторона куба:", cube.side)
print("Объем куба:", cube.calculate_volume())
print("Площадь поверхности куба:", cube.calculate_surface_area())
```

Вывод:

```
Сторона куба: 3
Объем куба: 27
Площадь поверхности куба: 54
```

Далее вывод для прямоугольного параллелепипеда и цилиндра.

6. Написать программу на Python, которая создает абстрактный класс `ChemicalSubstance` (с использованием модуля `abc`) для химических веществ. Класс должен содержать абстрактные методы `calculate_molar_mass()` и `calculate_density()`. Программа также должна создавать дочерние классы `Element`, `Compound` и `Mixture`, которые наследуют от класса `ChemicalSubstance` и реализуют специфические методы вычисления молярной массы и плотности.

Подсказка по формулам:

- `Element`: $M = atomic\_mass$, $\rho = mass/volume$
- `Compound`: $M = \sum(fraction \cdot atomic\_mass)$, $\rho = mass/volume$
- `Mixture`: $M = \sum(fraction \cdot molar\_mass)$, $\rho = \sum(fraction \cdot density)$

Инструкции:

(a) Создайте абстрактный класс `ChemicalSubstance` с методами `calculate_molar_mass()` и `calculate_density()`, используя модуль `abc`.

(b) Создайте класс `Element` с конструктором `__init__(self, atomic_mass, mass, volume)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Compound` с конструктором `__init__(self, fractions, atomic_masses, mass, volume)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Mixture` с конструктором `__init__(self, fractions, molar_masses, densities)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
el = Element(12, 24, 2)
print("Атомная масса элемента:", el.atomic_mass)
print("Молярная масса:", el.calculate_molar_mass())
print("Плотность:", el.calculate_density())
```

Вывод:

```
Атомная масса элемента: 12
Молярная масса: 12
Плотность: 12
```

Далее вывод для соединения и смеси.

7. Написать программу на Python, которая создает абстрактный класс `BankAccount` (с использованием модуля `abc`) для банковских счетов. Класс должен содержать абстрактные методы `calculate_interest()` и `calculate_balance()`. Программа также должна создавать дочерние классы `Savings`, `Checking` и `FixedDeposit`, которые наследуют от класса `BankAccount` и реализуют специфические методы вычисления процентов и баланса.

   Подсказка по формулам:

   - `Savings`: $Interest = balance \cdot rate \cdot time/100$, $Balance = balance + Interest$
   - `Checking`: $Interest = balance \cdot rate \cdot time/100 - fee$, $Balance = balance + Interest$
   - `FixedDeposit`: $Balance = principal \cdot (1 + rate/100)^time$, $Interest = Balance - principal$

   Инструкции:

   (a) Создайте абстрактный класс `BankAccount` с методами `calculate_interest()` и `calculate_balance()`, используя модуль `abc`.

   (b) Создайте класс `Savings` с конструктором `__init__(self, balance, rate, time)`, приватными атрибутами и геттерами. Реализуйте методы.

   (c) Создайте класс `Checking` с конструктором `__init__(self, balance, rate, time, fee)`, приватными атрибутами и геттерами. Реализуйте методы.

   (d) Создайте класс `FixedDeposit` с конструктором `__init__(self, principal, rate, time)`, приватными атрибутами и геттерами. Реализуйте методы.

   (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

   Пример использования:

```
s = Savings(1000, 5, 2)
print("Баланс на сберегательном счете:", s.balance)
print("Проценты:", s.calculate_interest())
print("Итоговый баланс:", s.calculate_balance())
```

   Вывод:

```
Баланс на сберегательном счете: 1000
Проценты: 100
Итоговый баланс: 1100
```

   Далее вывод для расчетного счета и срочного депозита.

8. Написать программу на Python, которая создает абстрактный класс `Shape3D` (с использованием модуля `abc`) для трехмерных фигур. Класс должен содержать абстрактные методы `calculate_volume()` и `calculate_surface_area()`. Программа также должна создавать дочерние классы `Sphere`, `Cone` и `Pyramid`, которые наследуют от класса `Shape3D` и реализуют специфические методы вычисления объема и площади поверхности.

   Подсказка по формулам:

   - `Sphere`: $V = 4/3 \cdot \pi r^3$, $S = 4 \cdot \pi r^2$
   - `Cone`: $V = 1/3 \cdot \pi r^2 h$, $S = \pi r(r + \sqrt{r^2 + h^2})$
   - `Pyramid`: $V = 1/3 \cdot base\_area \cdot height$, $S = base\_area + lateral\_area$

   Инструкции:

   (a) Создайте абстрактный класс `Shape3D` с методами `calculate_volume()` и `calculate_surface_area()`, используя модуль `abc`.

   (b) Создайте класс `Sphere` с конструктором `__init__(self, radius)`, приватным атрибутом и геттером. Реализуйте методы.

   (c) Создайте класс `Cone` с конструктором `__init__(self, radius, height)`, приватными атрибутами и геттерами. Реализуйте методы.

   (d) Создайте класс `Pyramid` с конструктором `__init__(self, base_area, lateral_area, height)`, приватными атрибутами и геттерами. Реализуйте методы.

   (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

   Пример использования:

```
s = Sphere(3)
print("Радиус сферы:", s.radius)
print("Объем сферы:", s.calculate_volume())
print("Площадь поверхности сферы:", s.calculate_surface_area())
```

   Вывод:

```
Радиус сферы: 3
Объем сферы: 113.097
Площадь поверхности сферы: 113.097
```

   Далее вывод для конуса и пирамиды.

9. Написать программу на Python, которая создает абстрактный класс `Vehicle` (с использованием модуля `abc`) для транспортных средств. Класс должен содержать абстрактные методы `calculate_fuel_consumption()` и `calculate_range()`. Программа также должна создавать дочерние классы `Car`, `Truck` и `Motorcycle`, которые наследуют от класса `Vehicle` и реализуют специфические методы вычисления расхода топлива и запаса хода.

   Подсказка по формулам:

   - `Car`: $fuel = distance/efficiency$, $range = tank\_capacity \cdot efficiency$

- `Truck`: $fuel = (distance/efficiency) \cdot load\_factor$, $range = tank\_capacity \cdot efficiency / load\_factor$
- `Motorcycle`: $fuel = distance/efficiency \cdot 0.8$, $range = tank\_capacity \cdot efficiency \cdot 1.2$

Инструкции:

(a) Создайте абстрактный класс `Vehicle` с методами `calculate_fuel_consumption()` и `calculate_range()`, используя модуль `abc`.

(b) Создайте класс `Car` с конструктором `__init__(self, efficiency, distance, tank_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Truck` с конструктором `__init__(self, efficiency, distance, tank_capacity, load_factor)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Motorcycle` с конструктором `__init__(self, efficiency, distance, tank_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
car = Car(15, 150, 50)
print("Эффективность автомобиля:", car.efficiency)
print("Расход топлива:", car.calculate_fuel_consumption())
print("Запас хода:", car.calculate_range())
```

Вывод:

```
Эффективность автомобиля: 15
Расход топлива: 10
Запас хода: 750
```

Далее вывод для грузовика и мотоцикла.

10. Написать программу на Python, которая создает абстрактный класс `Plant` (с использованием модуля `abc`) для растений. Класс должен содержать абстрактные методы `calculate_growth()` и `calculate_water_needs()`. Программа также должна создавать дочерние классы `Tree`, `Flower` и `Shrub`, которые наследуют от класса `Plant` и реализуют специфические методы вычисления роста и потребности в воде.

Подсказка по формулам:

- `Tree`: $growth = height\_rate \cdot time$, $water = area \cdot water\_rate$
- `Flower`: $growth = height\_rate \cdot time \cdot 0.5$, $water = area \cdot water\_rate \cdot 0.3$
- `Shrub`: $growth = height\_rate \cdot time \cdot 0.8$, $water = area \cdot water\_rate \cdot 0.6$

Инструкции:

(a) Создайте абстрактный класс `Plant` с методами `calculate_growth()` и `calculate_water_needs()`, используя модуль `abc`.

(b) Создайте класс `Tree` с конструктором `__init__(self, height_rate, time, area, water_rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Flower` с конструктором `__init__(self, height_rate, time, area, water_rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Shrub` с конструктором `__init__(self, height_rate, time, area, water_rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
tree = Tree(2, 5, 10, 3)
print("Скорость роста дерева:", tree.height_rate)
print("Рост:", tree.calculate_growth())
print("Потребность в воде:", tree.calculate_water_needs())
```

Вывод:

```
Скорость роста дерева: 2
Рост: 10
Потребность в воде: 30
```

Далее вывод для цветка и кустарника.

11. Написать программу на Python, которая создает абстрактный класс `Sensor` (с использованием модуля `abc`) для измерительных датчиков. Класс должен содержать абстрактные методы `calculate_signal()` и `calculate_accuracy()`. Программа также должна создавать дочерние классы `TemperatureSensor`, `PressureSensor` и `LightSensor`, которые наследуют от класса `Sensor` и реализуют специфические методы вычисления сигнала и точности.

Подсказка по формулам:

- `TemperatureSensor`: $signal = voltage \cdot sensitivity$, $accuracy = tolerance$
- `PressureSensor`: $signal = pressure \cdot sensitivity$, $accuracy = tolerance \cdot 1.1$
- `LightSensor`: $signal = intensity \cdot sensitivity$, $accuracy = tolerance \cdot 0.9$

Инструкции:

(a) Создайте абстрактный класс `Sensor` с методами `calculate_signal()` и `calculate_accuracy()`, используя модуль `abc`.

(b) Создайте класс `TemperatureSensor` с конструктором `__init__(self, voltage, sensitivity, tolerance)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `PressureSensor` с конструктором `__init__(self, pressure, sensitivity, tolerance)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `LightSensor` с конструктором `__init__(self, intensity, sensitivity, tolerance)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
temp_sensor = TemperatureSensor(5, 2, 0.1)
print("Напряжение:", temp_sensor.voltage)
print("Сигнал:", temp_sensor.calculate_signal())
print("Точность:", temp_sensor.calculate_accuracy())
```

Вывод:

```
Напряжение: 5
Сигнал: 10
Точность: 0.1
```

Далее вывод для датчиков давления и света.

12. Написать программу на Python, которая создает абстрактный класс `CookingIngredient` (с использованием модуля `abc`) для ингредиентов. Класс должен содержать абстрактные методы `calculate_calories()` и `calculate_mass()`. Программа также должна создавать дочерние классы `Vegetable`, `Meat` и `Grain`, которые наследуют от класса `CookingIngredient` и реализуют специфические методы вычисления калорий и массы.

Подсказка по формулам:

- `Vegetable`: $calories = weight \cdot cal\_per\_100g/100$, $mass = weight$
- `Meat`: $calories = weight \cdot cal\_per\_100g/100 \cdot 1.2$, $mass = weight$
- `Grain`: $calories = weight \cdot cal\_per\_100g/100 \cdot 1.1$, $mass = weight$

Инструкции:

(a) Создайте абстрактный класс `CookingIngredient` с методами `calculate_calories()` и `calculate_mass()`, используя модуль `abc`.

(b) Создайте класс `Vegetable` с конструктором `__init__(self, weight, cal_per_100g)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Meat` с конструктором `__init__(self, weight, cal_per_100g)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Grain` с конструктором `__init__(self, weight, cal_per_100g)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
veg = Vegetable(200, 30)
print("Вес овоща:", veg.weight)
print("Калории:", veg.calculate_calories())
print("Масса:", veg.calculate_mass())
```

Вывод:

```
Вес овоща: 200
Калории: 60
Масса: 200
```

Далее вывод для мяса и зерна.

13. Написать программу на Python, которая создает абстрактный класс `ElectronicDevice` (с использованием модуля `abc`) для электронных устройств. Класс должен содержать абстрактные методы `calculate_power()` и `calculate_efficiency()`. Программа также должна создавать дочерние классы `Laptop`, `Smartphone` и `Tablet`, которые наследуют от класса `ElectronicDevice` и реализуют специфические методы вычисления мощности и эффективности.

    Подсказка по формулам:

    - `Laptop`: $power = voltage \cdot current$, $efficiency = useful\_power/power$
    - `Smartphone`: $power = voltage \cdot current \cdot 0.8$, $efficiency = useful\_power/power$
    - `Tablet`: $power = voltage \cdot current \cdot 0.9$, $efficiency = useful\_power/power$

    Инструкции:

    (a) Создайте абстрактный класс `ElectronicDevice` с методами `calculate_power()` и `calculate_efficiency()`, используя модуль `abc`.

    (b) Создайте класс `Laptop` с конструктором `__init__(self, voltage, current, useful_power)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `Smartphone` с конструктором `__init__(self, voltage, current, useful_power)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `Tablet` с конструктором `__init__(self, voltage, current, useful_power)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

    ```
    laptop = Laptop(19, 3, 50)
    print("Напряжение ноутбука:", laptop.voltage)
    print("Мощность:", laptop.calculate_power())
    print("Эффективность:", laptop.calculate_efficiency())
    ```

    Вывод:

    ```
    Напряжение ноутбука: 19
    Мощность: 57
    Эффективность: 0.877
    ```

    Далее вывод для смартфона и планшета.

14. Написать программу на Python, которая создает абстрактный класс `MusicalInstrument` (с использованием модуля `abc`) для музыкальных инструментов. Класс должен содержать абстрактные методы `calculate_sound_level()` и `calculate_frequency()`. Программа также должна создавать дочерние классы `Piano`, `Guitar` и `Flute`, которые наследуют от класса `MusicalInstrument` и реализуют специфические методы вычисления уровня звука и частоты.

    Подсказка по формулам:

    - `Piano`: $sound\_level = keys \cdot intensity$, $frequency = 440 \cdot 2^{(note-49)/12}$
    - `Guitar`: $sound\_level = strings \cdot intensity \cdot 0.8$, $frequency = 440 \cdot 2^{(note-49)/12}$
    - `Flute`: $sound\_level = holes \cdot intensity \cdot 0.9$, $frequency = 440 \cdot 2^{(note-49)/12}$

    Инструкции:

    (a) Создайте абстрактный класс `MusicalInstrument` с методами `calculate_sound_level()` и `calculate_frequency()`, используя модуль `abc`.

    (b) Создайте класс `Piano` с конструктором `__init__(self, keys, intensity, note)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `Guitar` с конструктором `__init__(self, strings, intensity, note)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `Flute` с конструктором `__init__(self, holes, intensity, note)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

```
piano = Piano(88, 5, 49)
print("Клавиши:", piano.keys)
print("Уровень звука:", piano.calculate_sound_level())
print("Частота:", piano.calculate_frequency())
```

    Вывод:

```
Клавиши: 88
Уровень звука: 440
Частота: 440
```

    Далее вывод для гитары и флейты.

15. Написать программу на Python, которая создает абстрактный класс `Workout` (с использованием модуля `abc`) для физических упражнений. Класс должен содержать абстрактные методы `calculate_calories_burned()` и `calculate_duration()`. Программа также должна создавать дочерние классы `Cardio`, `Strength` и `Flexibility`, которые наследуют от класса `Workout` и реализуют специфические методы вычисления сожженных калорий и длительности тренировки.

    Подсказка по формулам:

    - `Cardio`: $calories = weight \cdot time \cdot 0.1$, $duration = time$

- `Strength`: $calories = weight \cdot time \cdot 0.08$, $duration = time$
- `Flexibility`: $calories = weight \cdot time \cdot 0.05$, $duration = time$

Инструкции:

(a) Создайте абстрактный класс `Workout` с методами `calculate_calories_burned()` и `calculate_duration()`, используя модуль `abc`.

(b) Создайте класс `Cardio` с конструктором `__init__(self, weight, time)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Strength` с конструктором `__init__(self, weight, time)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Flexibility` с конструктором `__init__(self, weight, time)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
cardio = Cardio(70, 30)
print("Вес:", cardio.weight)
print("Сожженные калории:", cardio.calculate_calories_burned())
print("Длительность:", cardio.calculate_duration())
```

Вывод:

```
Вес: 70
Сожженные калории: 210
Длительность: 30
```

Далее вывод для силовой и растяжки.

16. Написать программу на Python, которая создает абстрактный класс `ComputerComponent` (с использованием модуля `abc`) для компонентов компьютера. Класс должен содержать абстрактные методы `calculate_power_consumption()` и `calculate_cost()`. Программа также должна создавать дочерние классы `CPU`, `GPU` и `RAM`, которые наследуют от класса `ComputerComponent` и реализуют специфические методы вычисления энергопотребления и стоимости.

Подсказка по формулам:

- `CPU`: $power = cores \cdot frequency \cdot 10$, $cost = cores \cdot 50$
- `GPU`: $power = cores \cdot frequency \cdot 12$, $cost = cores \cdot 80$
- `RAM`: $power = size \cdot 3$, $cost = size \cdot 20$

Инструкции:

(a) Создайте абстрактный класс `ComputerComponent` с методами `calculate_power_consumption()` и `calculate_cost()`, используя модуль `abc`.

(b) Создайте класс `CPU` с конструктором `__init__(self, cores, frequency)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `GPU` с конструктором `__init__(self, cores, frequency)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `RAM` с конструктором `__init__(self, size)`, приватным атрибутом и геттером. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
cpu = CPU(4, 3.5)
print("Ядра CPU:", cpu.cores)
print("Энергопотребление:", cpu.calculate_power_consumption())
print("Стоимость:", cpu.calculate_cost())
```

Вывод:

```
Ядра CPU: 4
Энергопотребление: 140
Стоимость: 200
```

Далее вывод для GPU и RAM.

17. Написать программу на Python, которая создает абстрактный класс `Building` (с использованием модуля `abc`) для зданий. Класс должен содержать абстрактные методы `calculate_volume()` и `calculate_floor_area()`. Программа также должна создавать дочерние классы `House`, `Office` и `Warehouse`, которые наследуют от класса `Building` и реализуют специфические методы вычисления объема и площади.

Подсказка по формулам:

- `House`: $volume = length \cdot width \cdot height$, $floor\_area = length \cdot width$
- `Office`: $volume = length \cdot width \cdot height \cdot 1.2$, $floor\_area = length \cdot width \cdot 1.1$
- `Warehouse`: $volume = length \cdot width \cdot height \cdot 1.5$, $floor\_area = length \cdot width \cdot 1.3$

Инструкции:

(a) Создайте абстрактный класс `Building` с методами `calculate_volume()` и `calculate_floor_area()`, используя модуль `abc`.

(b) Создайте класс `House` с конструктором `__init__(self, length, width, height)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Office` с конструктором `__init__(self, length, width, height)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Warehouse` с конструктором `__init__(self, length, width, height)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
house = House(10, 8, 3)
print("Длина дома:", house.length)
print("Объем:", house.calculate_volume())
print("Площадь пола:", house.calculate_floor_area())
```

Вывод:

```
Длина дома: 10
Объем: 240
Площадь пола: 80
```

Далее вывод для офиса и склада.

18. Написать программу на Python, которая создает абстрактный класс `Vehicle` (с использованием модуля `abc`) для транспортных средств. Класс должен содержать абстрактные методы `calculate_max_speed()` и `calculate_range()`. Программа также должна создавать дочерние классы `Car`, `Motorcycle` и `Bicycle`, которые наследуют от класса `Vehicle` и реализуют специфические методы вычисления максимальной скорости и дальности.

    Подсказка по формулам:

    - `Car`: $max\_speed = engine\_power \cdot 2$, $range = fuel\_capacity \cdot 10$
    - `Motorcycle`: $max\_speed = engine\_power \cdot 2.5$, $range = fuel\_capacity \cdot 8$
    - `Bicycle`: $max\_speed = pedaling\_power \cdot 3$, $range = stamina \cdot 5$

    Инструкции:

    (a) Создайте абстрактный класс `Vehicle` с методами `calculate_max_speed()` и `calculate_range()`, используя модуль `abc`.

    (b) Создайте класс `Car` с конструктором `__init__(self, engine_power, fuel_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `Motorcycle` с конструктором `__init__(self, engine_power, fuel_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `Bicycle` с конструктором `__init__(self, pedaling_power, stamina)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

```
car = Car(150, 50)
print("Мощность двигателя автомобиля:", car.engine_power)
print("Максимальная скорость:", car.calculate_max_speed())
print("Дальность:", car.calculate_range())
```

    Вывод:

```
Мощность двигателя автомобиля: 150
Максимальная скорость: 300
Дальность: 500
```

Далее вывод для мотоцикла и велосипеда.

19. Написать программу на Python, которая создает абстрактный класс `BankAccount` (с использованием модуля `abc`) для банковских счетов. Класс должен содержать абстрактные методы `calculate_interest()` и `calculate_fees()`. Программа также должна создавать дочерние классы `SavingsAccount`, `CheckingAccount` и `InvestmentAccount`, которые наследуют от класса `BankAccount` и реализуют специфические методы вычисления процентов и комиссий.

    Подсказка по формулам:

    - `SavingsAccount`: $interest = balance \cdot 0.03,\ fees = 5$
    - `CheckingAccount`: $interest = balance \cdot 0.01,\ fees = 2$
    - `InvestmentAccount`: $interest = balance \cdot 0.05,\ fees = 10$

    Инструкции:

    (a) Создайте абстрактный класс `BankAccount` с методами `calculate_interest()` и `calculate_fees()`, используя модуль `abc`.

    (b) Создайте класс `SavingsAccount` с конструктором `__init__(self, balance)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `CheckingAccount` с конструктором `__init__(self, balance)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `InvestmentAccount` с конструктором `__init__(self, balance)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

```
savings = SavingsAccount(1000)
print("Баланс сберегательного счета:", savings.balance)
print("Проценты:", savings.calculate_interest())
print("Комиссии:", savings.calculate_fees())
```

    Вывод:

```
Баланс сберегательного счета: 1000
Проценты: 30.0
Комиссии: 5
```

    Далее вывод для расчетного и инвестиционного счета.

20. Написать программу на Python, которая создает абстрактный класс `Appliance` (с использованием модуля `abc`) для бытовой техники. Класс должен содержать абстрактные методы `calculate_energy_usage()` и `calculate_operating_cost()`. Программа также должна создавать дочерние классы `Refrigerator`, `WashingMachine` и `Microwave`, которые наследуют от класса `Appliance` и реализуют специфические методы вычисления энергопотребления и стоимости эксплуатации.

    Подсказка по формулам:

- `Refrigerator`: $energy = power \cdot hours$, $cost = energy \cdot 0.12$
- `WashingMachine`: $energy = power \cdot hours \cdot 1.1$, $cost = energy \cdot 0.12$
- `Microwave`: $energy = power \cdot hours \cdot 0.8$, $cost = energy \cdot 0.12$

Инструкции:

(a) Создайте абстрактный класс `Appliance` с методами `calculate_energy_usage()` и `calculate_operating_cost()`, используя модуль `abc`.

(b) Создайте класс `Refrigerator` с конструктором `__init__(self, power, hours)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `WashingMachine` с конструктором `__init__(self, power, hours)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Microwave` с конструктором `__init__(self, power, hours)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
fridge = Refrigerator(150, 24)
print("Мощность холодильника:", fridge.power)
print("Энергопотребление:", fridge.calculate_energy_usage())
print("Стоимость эксплуатации:", fridge.calculate_operating_cost())
```

Вывод:

```
Мощность холодильника: 150
Энергопотребление: 3600
Стоимость эксплуатации: 432.0
```

Далее вывод для стиральной машины и микроволновки.

21. Написать программу на Python, которая создает абстрактный класс `Planet` (с использованием модуля `abc`) для планет. Класс должен содержать абстрактные методы `calculate_surface_area()` и `calculate_gravity()`. Программа также должна создавать дочерние классы `Earth`, `Mars` и `Jupiter`, которые наследуют от класса `Planet` и реализуют специфические методы вычисления площади поверхности и силы гравитации.

Подсказка по формулам:

- `Earth`: $surface\_area = 4 \cdot \pi \cdot radius^2$, $gravity = G \cdot mass/radius^2$
- `Mars`: $surface\_area = 4 \cdot \pi \cdot radius^2 \cdot 0.95$, $gravity = G \cdot mass/radius^2 \cdot 0.38$
- `Jupiter`: $surface\_area = 4 \cdot \pi \cdot radius^2 \cdot 11.2$, $gravity = G \cdot mass/radius^2 \cdot 2.5$

Инструкции:

(a) Создайте абстрактный класс `Planet` с методами `calculate_surface_area()` и `calculate_gravity()`, используя модуль `abc`.

(b) Создайте класс `Earth` с конструктором `__init__(self, radius, mass)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Mars` с конструктором `__init__(self, radius, mass)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Jupiter` с конструктором `__init__(self, radius, mass)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
earth = Earth(6371, 5.97e24)
print("Радиус Земли:", earth.radius)
print("Площадь поверхности:", earth.calculate_surface_area())
print("Сила гравитации:", earth.calculate_gravity())
```

Вывод:

```
Радиус Земли: 6371
Площадь поверхности: 510064471
Сила гравитации: 9.8
```

Далее вывод для Марса и Юпитера.

22. Написать программу на Python, которая создает абстрактный класс `FoodItem` (с использованием модуля `abc`) для пищевых продуктов. Класс должен содержать абстрактные методы `calculate_calories()` и `calculate_price()`. Программа также должна создавать дочерние классы `Fruit`, `Vegetable` и `Meat`, которые наследуют от класса `FoodItem` и реализуют специфические методы вычисления калорийности и стоимости.

Подсказка по формулам:

- `Fruit`: $calories = weight \cdot 0.52$, $price = weight \cdot 3$
- `Vegetable`: $calories = weight \cdot 0.3$, $price = weight \cdot 2$
- `Meat`: $calories = weight \cdot 2.5$, $price = weight \cdot 10$

Инструкции:

(a) Создайте абстрактный класс `FoodItem` с методами `calculate_calories()` и `calculate_price()`, используя модуль `abc`.

(b) Создайте класс `Fruit` с конструктором `__init__(self, weight)`, приватным атрибутом и геттером. Реализуйте методы.

(c) Создайте класс `Vegetable` с конструктором `__init__(self, weight)`, приватным атрибутом и геттером. Реализуйте методы.

(d) Создайте класс `Meat` с конструктором `__init__(self, weight)`, приватным атрибутом и геттером. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
apple = Fruit(150)
print("Вес фрукта:", apple.weight)
print("Калории:", apple.calculate_calories())
print("Стоимость:", apple.calculate_price())
```

Вывод:

```
Вес фрукта: 150
Калории: 78.0
Стоимость: 450
```

Далее вывод для овощей и мяса.

23. Написать программу на Python, которая создает абстрактный класс `Tool` (с использованием модуля `abc`) для инструментов. Класс должен содержать абстрактные методы `calculate_efficiency()` и `calculate_durability()`. Программа также должна создавать дочерние классы `Hammer`, `Screwdriver` и `Wrench`, которые наследуют от класса `Tool` и реализуют специфические методы вычисления эффективности и прочности.

    Подсказка по формулам:

    - `Hammer`: $efficiency = weight \cdot swing\_speed$, $durability = material\_hardness \cdot 10$
    - `Screwdriver`: $efficiency = length \cdot torque$, $durability = material\_hardness \cdot 8$
    - `Wrench`: $efficiency = size \cdot torque$, $durability = material\_hardness \cdot 12$

    Инструкции:

    (a) Создайте абстрактный класс `Tool` с методами `calculate_efficiency()` и `calculate_durability()`, используя модуль `abc`.

    (b) Создайте класс `Hammer` с конструктором `__init__(self, weight, swing_speed, material_hardness)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `Screwdriver` с конструктором `__init__(self, length, torque, material_hardness)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `Wrench` с конструктором `__init__(self, size, torque, material_hardness)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

    ```
    hammer = Hammer(2, 5, 7)
    print("Вес молотка:", hammer.weight)
    print("Эффективность:", hammer.calculate_efficiency())
    print("Прочность:", hammer.calculate_durability())
    ```

    Вывод:

```
Вес молотка: 2
Эффективность: 10
Прочность: 70
```

Далее вывод для отвертки и ключа.

24. Написать программу на Python, которая создает абстрактный класс `Book` (с использованием модуля `abc`) для книг. Класс должен содержать абстрактные методы `calculate_reading_time()` и `calculate_cost()`. Программа также должна создавать дочерние классы `Fiction`, `NonFiction` и `Comics`, которые наследуют от класса `Book` и реализуют специфические методы вычисления времени чтения и стоимости.

    Подсказка по формулам:

    - `Fiction`: $reading\_time = pages \cdot 2$, $cost = pages \cdot 1.5$
    - `NonFiction`: $reading\_time = pages \cdot 2.5$, $cost = pages \cdot 2$
    - `Comics`: $reading\_time = pages \cdot 1$, $cost = pages \cdot 1$

    Инструкции:

    (a) Создайте абстрактный класс `Book` с методами `calculate_reading_time()` и `calculate_cost()`, используя модуль `abc`.

    (b) Создайте класс `Fiction` с конструктором `__init__(self, pages)`, приватным атрибутом и геттером. Реализуйте методы.

    (c) Создайте класс `NonFiction` с конструктором `__init__(self, pages)`, приватным атрибутом и геттером. Реализуйте методы.

    (d) Создайте класс `Comics` с конструктором `__init__(self, pages)`, приватным атрибутом и геттером. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

    ```
    novel = Fiction(300)
    print("Количество страниц:", novel.pages)
    print("Время чтения:", novel.calculate_reading_time())
    print("Стоимость:", novel.calculate_cost())
    ```

    Вывод:

    ```
    Количество страниц: 300
    Время чтения: 600
    Стоимость: 450.0
    ```

    Далее вывод для научной литературы и комиксов.

25. Написать программу на Python, которая создает абстрактный класс `ElectronicDevice` (с использованием модуля `abc`) для электронных устройств. Класс должен содержать абстрактные методы `calculate_power_consumption()` и `calculate_battery_life()`. Программа также должна создавать дочерние классы `Smartphone`, `Laptop` и `Tablet`, которые наследуют от класса `ElectronicDevice` и реализуют специфические методы вычисления потребляемой мощности и времени работы от батареи.

    Подсказка по формулам:

- `Smartphone`: $power = voltage \cdot current \cdot hours$, $battery\_life = battery\_capacity / current$
- `Laptop`: $power = voltage \cdot current \cdot hours \cdot 1.5$, $battery\_life = battery\_capacity / (current \cdot 1.5)$
- `Tablet`: $power = voltage \cdot current \cdot hours \cdot 1.2$, $battery\_life = battery\_capacity / (current \cdot 1.2)$

Инструкции:

(a) Создайте абстрактный класс `ElectronicDevice` с методами `calculate_power_consumption()` и `calculate_battery_life()`, используя модуль `abc`.

(b) Создайте класс `Smartphone` с конструктором `__init__(self, voltage, current, hours, battery_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Laptop` с конструктором `__init__(self, voltage, current, hours, battery_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Tablet` с конструктором `__init__(self, voltage, current, hours, battery_capacity)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
phone = Smartphone(5, 1, 10, 5000)
print("Напряжение смартфона:", phone.voltage)
print("Потребляемая мощность:", phone.calculate_power_consumption())
print("Время работы от батареи:", phone.calculate_battery_life())
```

Вывод:

```
Напряжение смартфона: 5
Потребляемая мощность: 50
Время работы от батареи: 5000.0
```

Далее вывод для ноутбука и планшета.

26. Написать программу на Python, которая создает абстрактный класс `MusicalInstrument` (с использованием модуля `abc`) для музыкальных инструментов. Класс должен содержать абстрактные методы `calculate_sound_volume()` и `calculate_weight()`. Программа также должна создавать дочерние классы `Piano`, `Guitar` и `Drum`, которые наследуют от класса `MusicalInstrument` и реализуют специфические методы вычисления громкости и веса.

Подсказка по формулам:

- `Piano`: $volume = keys \cdot 2$, $weight = base\_weight \cdot 3$
- `Guitar`: $volume = strings \cdot 3$, $weight = base\_weight \cdot 1.5$
- `Drum`: $volume = diameter \cdot 4$, $weight = base\_weight \cdot 2$

Инструкции:

(a) Создайте абстрактный класс `MusicalInstrument` с методами `calculate_sound_volume()` и `calculate_weight()`, используя модуль `abc`.

(b) Создайте класс `Piano` с конструктором `__init__(self, keys, base_weight)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Guitar` с конструктором `__init__(self, strings, base_weight)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Drum` с конструктором `__init__(self, diameter, base_weight)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
piano = Piano(88, 200)
print("Количество клавиш:", piano.keys)
print("Громкость:", piano.calculate_sound_volume())
print("Вес:", piano.calculate_weight())
```

Вывод:

```
Количество клавиш: 88
Громкость: 176
Вес: 600
```

Далее вывод для гитары и барабана.

27. Написать программу на Python, которая создает абстрактный класс `VehiclePart` (с использованием модуля `abc`) для частей транспортного средства. Класс должен содержать абстрактные методы `calculate_durability()` и `calculate_maintenance_cost()`. Программа также должна создавать дочерние классы `Engine`, `Wheel` и `Brake`, которые наследуют от класса `VehiclePart` и реализуют специфические методы вычисления долговечности и стоимости обслуживания.

Подсказка по формулам:

- `Engine`: $durability = hours\_run \cdot 1.2$, $maintenance = base\_cost \cdot 5$
- `Wheel`: $durability = rotation\_count \cdot 0.8$, $maintenance = base\_cost \cdot 2$
- `Brake`: $durability = pressure\_applied \cdot 0.5$, $maintenance = base\_cost \cdot 3$

Инструкции:

(a) Создайте абстрактный класс `VehiclePart` с методами `calculate_durability()` и `calculate_maintenance_cost()`, используя модуль `abc`.

(b) Создайте класс `Engine` с конструктором `__init__(self, hours_run, base_cost)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Wheel` с конструктором `__init__(self, rotation_count, base_cost)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Brake` с конструктором `__init__(self, pressure_applied, base_cost)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
engine = Engine(1000, 200)
print("Наработка двигателя:", engine.hours_run)
print("Долговечность:", engine.calculate_durability())
print("Стоимость обслуживания:", engine.calculate_maintenance_cost())
```

Вывод:

```
Наработка двигателя: 1000
Долговечность: 1200.0
Стоимость обслуживания: 1000
```

Далее вывод для колес и тормозов.

28. Написать программу на Python, которая создает абстрактный класс `Appliance` (с использованием модуля `abc`) для бытовых приборов. Класс должен содержать абстрактные методы `calculate_energy_consumption()` и `calculate_cost()`. Программа также должна создавать дочерние классы `Refrigerator`, `WashingMachine` и `Microwave`, которые наследуют от класса `Appliance` и реализуют специфические методы вычисления потребляемой энергии и стоимости эксплуатации.

Подсказка по формулам:

- `Refrigerator`: $energy = power \cdot hours \cdot 30$, $cost = energy \cdot rate$
- `WashingMachine`: $energy = power \cdot hours \cdot 1.5$, $cost = energy \cdot rate$
- `Microwave`: $energy = power \cdot hours \cdot 0.8$, $cost = energy \cdot rate$

Инструкции:

(a) Создайте абстрактный класс `Appliance` с методами `calculate_energy_consumption()` и `calculate_cost()`, используя модуль `abc`.

(b) Создайте класс `Refrigerator` с конструктором `__init__(self, power, hours, rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `WashingMachine` с конструктором `__init__(self, power, hours, rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Microwave` с конструктором `__init__(self, power, hours, rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
fridge = Refrigerator(150, 24, 0.1)
print("Мощность холодильника:", fridge.power)
print("Энергопотребление:", fridge.calculate_energy_consumption())
print("Стоимость эксплуатации:", fridge.calculate_cost())
```

Вывод:

```
Мощность холодильника: 150
Энергопотребление: 108000
Стоимость эксплуатации: 10800.0
```

Далее вывод для стиральной машины и микроволновки.

29. Написать программу на Python, которая создает абстрактный класс `SportActivity` (с использованием модуля `abc`) для спортивных занятий. Класс должен содержать абстрактные методы `calculate_calories_burned()` и `calculate_duration()`. Программа также должна создавать дочерние классы `Running`, `Swimming` и `Cycling`, которые наследуют от класса `SportActivity` и реализуют специфические методы вычисления сожженных калорий и продолжительности.

Подсказка по формулам:

- `Running`: $calories = weight \cdot distance \cdot 1.036$, $duration = distance/speed$
- `Swimming`: $calories = weight \cdot distance \cdot 1.5$, $duration = distance/speed$
- `Cycling`: $calories = weight \cdot distance \cdot 0.8$, $duration = distance/speed$

Инструкции:

(a) Создайте абстрактный класс `SportActivity` с методами `calculate_calories_burned()` и `calculate_duration()`, используя модуль `abc`.

(b) Создайте класс `Running` с конструктором `__init__(self, weight, distance, speed)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Swimming` с конструктором `__init__(self, weight, distance, speed)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `Cycling` с конструктором `__init__(self, weight, distance, speed)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
run = Running(70, 5, 10)
print("Вес бегуна:", run.weight)
print("Сожженные калории:", run.calculate_calories_burned())
print("Продолжительность:", run.calculate_duration())
```

Вывод:

```
Вес бегуна: 70
Сожженные калории: 362.6
Продолжительность: 0.5
```

Далее вывод для плавания и езды на велосипеде.

30. Написать программу на Python, которая создает абстрактный класс `BuildingMaterial` (с использованием модуля `abc`) для строительных материалов. Класс должен содержать абстрактные методы `calculate_strength()` и `calculate_cost()`. Программа также должна создавать дочерние классы `Concrete`, `Wood`, `Steel`, которые наследуют от класса `BuildingMaterial` и реализуют специфические методы вычисления прочности и стоимости.

    Подсказка по формулам:

    - `Concrete`: $strength = density \cdot compressive\_factor$, $cost = volume \cdot price\_per\_m3$
    - `Wood`: $strength = density \cdot elastic\_factor$, $cost = volume \cdot price\_per\_m3$
    - `Steel`: $strength = density \cdot tensile\_factor$, $cost = volume \cdot price\_per\_m3$

    Инструкции:

    (a) Создайте абстрактный класс `BuildingMaterial` с методами `calculate_strength()` и `calculate_cost()`, используя модуль `abc`.

    (b) Создайте класс `Concrete` с конструктором `__init__(self, density, compressive_factor, volume, price_per_m3)`, приватными атрибутами и геттерами. Реализуйте методы.

    (c) Создайте класс `Wood` с конструктором `__init__(self, density, elastic_factor, volume, price_per_m3)`, приватными атрибутами и геттерами. Реализуйте методы.

    (d) Создайте класс `Steel` с конструктором `__init__(self, density, tensile_factor, volume, price_per_m3)`, приватными атрибутами и геттерами. Реализуйте методы.

    (e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

    Пример использования:

```
concrete = Concrete(2400, 30, 2, 100)
print("Плотность бетона:", concrete.density)
print("Прочность:", concrete.calculate_strength())
print("Стоимость:", concrete.calculate_cost())
```

    Вывод:

```
Плотность бетона: 2400
Прочность: 72000
Стоимость: 200
```

    Далее вывод для древесины и стали.

31. Написать программу на Python, которая создает абстрактный класс `TransportVehicle` (с использованием модуля `abc`) для транспортных средств. Класс должен содержать абстрактные методы `calculate_range()` и `calculate_fuel_cost()`. Программа также должна создавать дочерние классы `Car`, `Motorcycle` и `ElectricScooter`, которые наследуют от класса `TransportVehicle` и реализуют специфические методы вычисления дальности хода и стоимости топлива/энергии.

    Подсказка по формулам:

- `Car`: $range = tank\_capacity / consumption \cdot 100$, $fuel\_cost = tank\_capacity \cdot fuel\_price$
- `Motorcycle`: $range = tank\_capacity / consumption \cdot 120$, $fuel\_cost = tank\_capacity \cdot fuel\_price$
- `ElectricScooter`: $range = battery\_capacity / consumption \cdot 100$, $fuel\_cost = battery\_capacity \cdot electricity\_rate$

Инструкции:

(a) Создайте абстрактный класс `TransportVehicle` с методами `calculate_range()` и `calculate_fuel_cost()`, используя модуль `abc`.

(b) Создайте класс `Car` с конструктором `__init__(self, tank_capacity, consumption, fuel_price)`, приватными атрибутами и геттерами. Реализуйте методы.

(c) Создайте класс `Motorcycle` с конструктором `__init__(self, tank_capacity, consumption, fuel_price)`, приватными атрибутами и геттерами. Реализуйте методы.

(d) Создайте класс `ElectricScooter` с конструктором `__init__(self, battery_capacity, consumption, electricity_rate)`, приватными атрибутами и геттерами. Реализуйте методы.

(e) Создайте экземпляр каждого класса и вызовите методы, используя геттеры, и выведите результаты.

Пример использования:

```
car = Car(50, 8, 1.5)
print("Ёмкость бака автомобиля:", car.tank_capacity)
print("Дальность хода:", car.calculate_range())
print("Стоимость топлива:", car.calculate_fuel_cost())
```

Вывод:

```
Ёмкость бака автомобиля: 50
Дальность хода: 625.0
Стоимость топлива: 75.0
```

Далее вывод для Motorcycle и ElectricScooter.

2.3 Семинар «Структуры данных в ООП-реализации» (2 часа)

В ходе работы решите 4 задачи. Предполагается, что пользователь класса не имеет права обращаться к свойствам напрямую (соблюдая принцип инкапсуляции), а должен использовать методы.

Продемонстрируйте работоспособность всех методов (из задания) посредством создания запускаемых файлов, где осуществляется вызов методов для разных ситуаций (без ручного ввода, но с выводом результатов в консоль).

Каждый класс должен сохраняться в отдельном исходном файле. Необходимо соблюдать все стандартные требования к качеству кода (отступы, именования переменных, классов, методов, проверка корректности входных данных).

Задания этого семинара предназначены для освоения не только ООП, но и структур данных, поэтому требуется структуры формировать вручную без использования библиотечных вариантов.

Для сдачи работы будьте готовы пояснить или аналогично заданию модифицировать любую часть кода, а также ответить на вопросы:

1. Что обозначает свойство наследования в парадигме ООП?
2. Что обозначает свойство полиморфизма в парадигме ООП?
3. Опишите реализацию наследования в Python
4. Как создать конструктор в Python
5. Как реализовать абстрактный класс в Python (и что это значит)
6. Как реализовать абстрактные методы в Python (и что это значит)
7. Опишите бинарное дерево
8. Как вставить элемент в бинарное дерево
9. Как найи элемент в бинарном дереве
10. Опишите, что такое стек
11. Опишите, что такое очередь
12. Опишите двусвязный список
13. Сравните стек, очередь и двусвязный список

Если вы нашли в задачнике ошибки, опечатки и другие недостатки, то вы можете сделать pull-request.

Срок сдачи работы (начала сдачи): через одно занятие после его выдачи. В последующие сроки оценка будет снижаться (при отсутствии оправдывающих документов).

2.3.1 Задача 1 (дерево)

1. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией внутренней структуры. Программа должна создавать экземпляры класса TreeNode, которые представляют узлы дерева, и класса SearchTree, который представляет дерево поиска. Класс SearchTree должен содержать методы для добавления,

поиска и удаления элементов из дерева, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс TreeNode с методом __init__, который принимает значение в качестве аргумента и сохраняет его в атрибуте self.data. Атрибуты left и right должны быть инициализированы как None.

(b) Создайте класс SearchTree с методом __init__, который инициализирует корневой узел дерева как None.

(c) Создайте публичный метод add в классе SearchTree, который добавляет значение в дерево. Если корневой узел отсутствует, создайте новый узел с добавляемым значением. В противном случае, вызовите приватный метод _add_helper, передав ему корень и значение.

(d) Создайте приватный метод _add_helper в классе SearchTree, который рекурсивно добавляет значение в дерево. Если значение меньше или равно значению текущего узла, добавьте его в левое поддерево. Если значение строго больше значения текущего узла, добавьте его в правое поддерево.

(e) Создайте публичный метод locate в классе SearchTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_helper, передав ему корень и искомое значение.

(f) Создайте приватный метод _locate_helper в классе SearchTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_helper для поиска значения в левом поддереве (если искомое значение меньше или равно значению текущего узла) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса SearchTree и вставьте в него 15 случайных чисел от 1 до 30.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
tree = SearchTree()
for i in range(15):
    tree.add(random.randint(1, 30))

print("Поиск элементов:")
print(tree.locate(7))   # Обнаружено, возвращен узел (7)
print(tree.locate(25))  # Не обнаружено, возвращено None
print(tree.locate(15))  # Обнаружено, возвращен узел (15)
```

2. Написать программу на Python, которая реализует бинарное дерево поиска с соблюдением принципов инкапсуляции. Программа должна создавать экземпляры класса Vertex, которые представляют узлы дерева, и класса BinaryTree, который представляет дерево поиска. Класс BinaryTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные вспомогательные методы должны быть скрыты от внешнего доступа. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

   Инструкции:

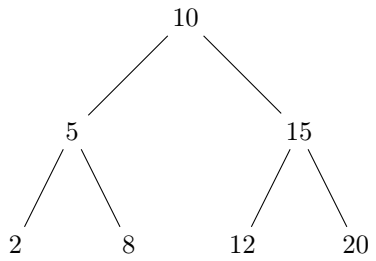


Рис. 1: Пример бинарного дерева поиска

(a) Создайте класс Vertex с методом __init__, который принимает значение value и сохраняет его в атрибуте self.key. Атрибуты self.left_child и self.right_child должны быть инициализированы как None.

(b) Создайте класс BinaryTree с методом __init__, который инициализирует атрибут self.top как None.

(c) Создайте публичный метод put в классе BinaryTree, который вставляет значение в дерево. Если self.top отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _put_recursively, передав ему self.top и значение.

(d) Создайте приватный метод _put_recursively в классе BinaryTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find в классе BinaryTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_recursively, передав ему self.top и искомое значение.

(f) Создайте приватный метод _find_recursively в классе BinaryTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_recursively для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса BinaryTree и вставьте в него 18 случайных чисел от 5 до 35.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
bt = BinaryTree()
for i in range(18):
    bt.put(random.randint(5, 35))

print("Поиск элементов:")
print(bt.find(10))  # Обнаружено, возвращен узел (10)
print(bt.find(40))  # Не обнаружено, возвращено None
print(bt.find(22))  # Обнаружено, возвращен узел (22)
```

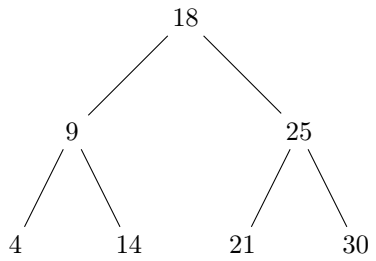


Рис. 2: Пример бинарного дерева поиска

3. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией внутренней логики. Программа должна создавать экземпляры класса BNode, которые представляют узлы дерева, и класса BSTree, который представляет дерево поиска. Класс BSTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные функции должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

   Инструкции:

   (a) Создайте класс BNode с методом __init__, который принимает параметр item и сохраняет его в атрибуте self.element. Атрибуты self.left_branch и self.right_branch должны быть инициализированы как None.

   (b) Создайте класс BSTree с методом __init__, который инициализирует атрибут self.root_node как None.

   (c) Создайте публичный метод insert_value в классе BSTree, который вставляет значение в дерево. Если self.root_node отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _recursive_insert, передав ему self.root_node и значение.

   (d) Создайте приватный метод _recursive_insert в классе BSTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

   (e) Создайте публичный метод retrieve в классе BSTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _recursive_retrieve, передав ему self.root_node и искомое значение.

   (f) Создайте приватный метод _recursive_retrieve в классе BSTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _recursive_retrieve для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

   (g) Создайте экземпляр класса BSTree и вставьте в него 20 случайных чисел от 1 до 40.

   (h) Выполните поиск элементов в дереве и выведите результаты на экран.

   Пример использования:

```
bst = BSTree()
for i in range(20):
    bst.insert_value(random.randint(1, 40))

print("Поиск элементов:")
print(bst.retrieve(12))  # Обнаружено, возвращен узел (12)
print(bst.retrieve(50))  # Не обнаружено, возвращено None
print(bst.retrieve(33))  # Обнаружено, возвращен узел (33)
```

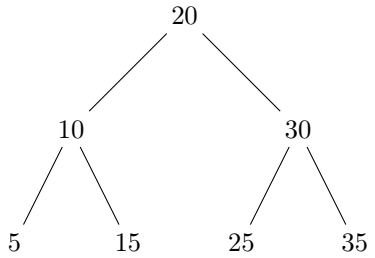


Рис. 3: Пример бинарного дерева поиска

4. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса ElementNode, которые представляют узлы дерева, и класса OrderedTree, который представляет дерево поиска. Класс OrderedTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

   Инструкции:

   (a) Создайте класс ElementNode с методом __init__, который принимает параметр content и сохраняет его в атрибуте self.payload. Атрибуты self.left и self.right должны быть инициализированы как None.

   (b) Создайте класс OrderedTree с методом __init__, который инициализирует атрибут self.head как None.

   (c) Создайте публичный метод store в классе OrderedTree, который вставляет значение в дерево. Если self.head отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _store_recursive, передав ему self.head и значение.

   (d) Создайте приватный метод _store_recursive в классе OrderedTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

   (e) Создайте публичный метод query в классе OrderedTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _query_recursive, передав ему self.head и искомое значение.

   (f) Создайте приватный метод _query_recursive в классе OrderedTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном

случае, рекурсивно вызывайте метод _query_recursive для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса OrderedTree и вставьте в него 17 случайных чисел от 3 до 33.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ot = OrderedTree()
for i in range(17):
    ot.store(random.randint(3, 33))

print("Поиск элементов:")
print(ot.query(8))   # Обнаружено, возвращен узел (8)
print(ot.query(45))  # Не обнаружено, возвращено None
print(ot.query(27))  # Обнаружено, возвращен узел (27)
```

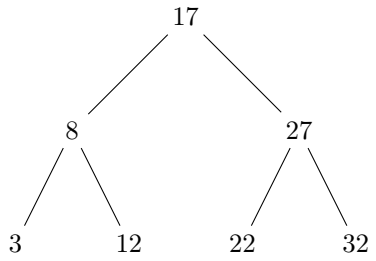


Рис. 4: Пример бинарного дерева поиска

5. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией внутренней структуры. Программа должна создавать экземпляры класса DataNode, которые представляют узлы дерева, и класса SortedTree, который представляет дерево поиска. Класс SortedTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть скрыты. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

   Инструкции:

   (a) Создайте класс DataNode с методом __init__, который принимает параметр val и сохраняет его в атрибуте self.entry. Атрибуты self.left и self.right должны быть инициализированы как None.

   (b) Создайте класс SortedTree с методом __init__, который инициализирует атрибут self.first_node как None.

   (c) Создайте публичный метод enqueue в классе SortedTree, который вставляет значение в дерево. Если self.first_node отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _enqueue_helper, передав ему self.first_node и значение.

   (d) Создайте приватный метод _enqueue_helper в классе SortedTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению

текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод lookup в классе SortedTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _lookup_helper, передав ему self.first_node и искомое значение.

(f) Создайте приватный метод _lookup_helper в классе SortedTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _lookup_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса SortedTree и вставьте в него 16 случайных чисел от 2 до 28.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
st = SortedTree()
for i in range(16):
    st.enqueue(random.randint(2, 28))

print("Поиск элементов:")
print(st.lookup(6))   # Обнаружено, возвращен узел (6)
print(st.lookup(35))  # Не обнаружено, возвращено None
print(st.lookup(19))  # Обнаружено, возвращен узел (19)
```

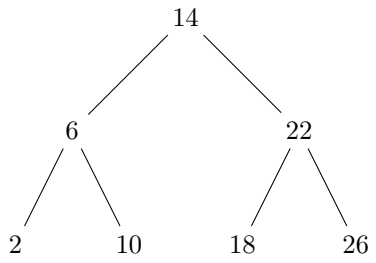


Рис. 5: Пример бинарного дерева поиска

6. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса BinNode, которые представляют узлы дерева, и класса LookupTree, который представляет дерево поиска. Класс LookupTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс BinNode с методом __init__, который принимает параметр num и сохраняет его в атрибуте self.number. Атрибуты self.left и self.right должны быть инициализированы как None.

(b) Создайте класс LookupTree с методом __init__, который инициализирует атрибут self.initial_node как None.

(c) Создайте публичный метод add_entry в классе LookupTree, который вставляет значение в дерево. Если self.initial_node отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_entry_rec, передав ему self.initial_node и значение.

(d) Создайте приватный метод _add_entry_rec в классе LookupTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод fetch в классе LookupTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _fetch_rec, передав ему self.initial_node и искомое значение.

(f) Создайте приватный метод _fetch_rec в классе LookupTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _fetch_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса LookupTree и вставьте в него 19 случайных чисел от 4 до 34.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
lt = LookupTree()
for i in range(19):
    lt.add_entry(random.randint(4, 34))

print("Поиск элементов:")
print(lt.fetch(9))   # Обнаружено, возвращен узел (9)
print(lt.fetch(40))  # Не обнаружено, возвращено None
print(lt.fetch(24))  # Обнаружено, возвращен узел (24)
```

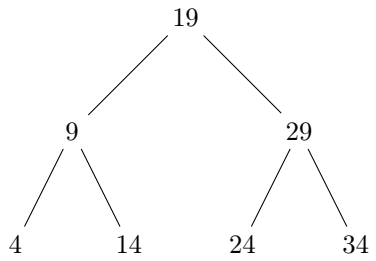


Рис. 6: Пример бинарного дерева поиска

7. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса NodeItem, которые представляют узлы дерева, и класса BinaryTreeSearch, который представляет дерево поиска. Класс BinaryTreeSearch должен содержать методы для вставки, поиска и удаления

элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс NodeItem с методом __init__, который принимает параметр item_value и сохраняет его в атрибуте self.val. Атрибуты self.left и self.right должны быть инициализированы как None.

(b) Создайте класс BinaryTreeSearch с методом __init__, который инициализирует атрибут self.start_node как None.

(c) Создайте публичный метод insert_item в классе BinaryTreeSearch, который вставляет значение в дерево. Если self.start_node отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_item_helper, передав ему self.start_node и значение.

(d) Создайте приватный метод _insert_item_helper в классе BinaryTreeSearch, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_item в классе BinaryTreeSearch, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_item_helper, передав ему self.start_node и искомое значение.

(f) Создайте приватный метод _find_item_helper в классе BinaryTreeSearch, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_item_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса BinaryTreeSearch и вставьте в него 21 случайное число от 1 до 38.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
bts = BinaryTreeSearch()
for i in range(21):
    bts.insert_item(random.randint(1, 38))

print("Поиск элементов:")
print(bts.find_item(11))  # Обнаружено, возвращен узел (11)
print(bts.find_item(50))  # Не обнаружено, возвращено None
print(bts.find_item(29))  # Обнаружено, возвращен узел (29)
```

8. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса TreeVertex, которые представляют узлы дерева, и класса SearchBinTree, который представляет дерево поиска. Класс SearchBinTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

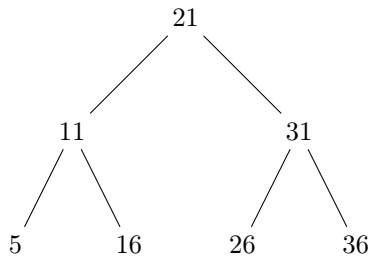


Рис. 7: Пример бинарного дерева поиска

Инструкции:

(a) Создайте класс TreeVertex с методом __init__, который принимает параметр vertex_data и сохраняет его в атрибуте self.info. Атрибуты self.left и self.right должны быть инициализированы как None.

(b) Создайте класс SearchBinTree с методом __init__, который инициализирует атрибут self.root_vertex как None.

(c) Создайте публичный метод insert_data в классе SearchBinTree, который вставляет значение в дерево. Если self.root_vertex отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_data_rec, передав ему self.root_vertex и значение.

(d) Создайте приватный метод _insert_data_rec в классе SearchBinTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_data в классе SearchBinTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_data_rec, передав ему self.root_vertex и искомое значение.

(f) Создайте приватный метод _search_data_rec в классе SearchBinTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_data_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса SearchBinTree и вставьте в него 14 случайных чисел от 6 до 36.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
sbt = SearchBinTree()
for i in range(14):
    sbt.insert_data(random.randint(6, 36))

print("Поиск элементов:")
print(sbt.search_data(13))  # Обнаружено, возвращен узел (13)
print(sbt.search_data(42))  # Не обнаружено, возвращено None
print(sbt.search_data(28))  # Обнаружено, возвращен узел (28)
```

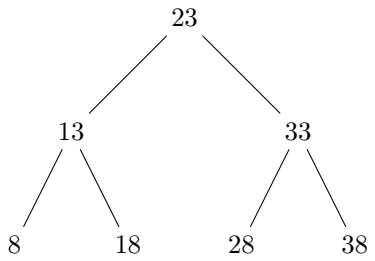


Рис. 8: Пример бинарного дерева поиска

9. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса BranchNode, которые представляют узлы дерева, и класса BinaryTreeLookup, который представляет дерево поиска. Класс BinaryTreeLookup должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

   Инструкции:

   (a) Создайте класс BranchNode с методом __init__, который принимает параметр node_val и сохраняет его в атрибуте self.data_point. Атрибуты self.left_link и self.right_link должны быть инициализированы как None.

   (b) Создайте класс BinaryTreeLookup с методом __init__, который инициализирует атрибут self.base_node как None.

   (c) Создайте публичный метод add_point в классе BinaryTreeLookup, который вставляет значение в дерево. Если self.base_node отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_point_recursive, передав ему self.base_node и значение.

   (d) Создайте приватный метод _add_point_recursive в классе BinaryTreeLookup, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

   (e) Создайте публичный метод locate_point в классе BinaryTreeLookup, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_point_recursive, передав ему self.base_node и искомое значение.

   (f) Создайте приватный метод _locate_point_recursive в классе BinaryTreeLookup, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_point_recursive для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

   (g) Создайте экземпляр класса BinaryTreeLookup и вставьте в него 13 случайных чисел от 7 до 37.

   (h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
btl = BinaryTreeLookup()
for i in range(13):
    btl.add_point(random.randint(7, 37))

print("Поиск элементов:")
print(btl.locate_point(14))  # Обнаружено, возвращен узел (14)
print(btl.locate_point(45))  # Не обнаружено, возвращено None
print(btl.locate_point(27))  # Обнаружено, возвращен узел (27)
```

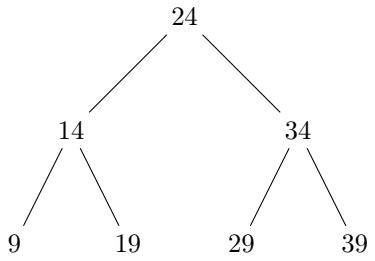


Рис. 9: Пример бинарного дерева поиска

10. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса TreeNodeStruct, которые представляют узлы дерева, и класса BinSearchStructure, который представляет дерево поиска. Класс BinSearchStructure должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс TreeNodeStruct с методом __init__, который принимает параметр struct_value и сохраняет его в атрибуте self.node_value. Атрибуты self.left_sub и self.right_sub должны быть инициализированы как None.

    (b) Создайте класс BinSearchStructure с методом __init__, который инициализирует атрибут self.top_element как None.

    (c) Создайте публичный метод insert_struct в классе BinSearchStructure, который вставляет значение в дерево. Если self.top_element отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_struct_helper, передав ему self.top_element и значение.

    (d) Создайте приватный метод _insert_struct_helper в классе BinSearchStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод find_struct в классе BinSearchStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_struct_helper, передав ему self.top_element и искомое значение.

(f) Создайте приватный метод _find_struct_helper в классе BinSearchStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_struct_helper для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса BinSearchStructure и вставьте в него 22 случайных числа от 2 до 42.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
bss = BinSearchStructure()
for i in range(22):
    bss.insert_struct(random.randint(2, 42))

print("Поиск элементов:")
print(bss.find_struct(15))  # Обнаружено, возвращен узел (15)
print(bss.find_struct(55))  # Не обнаружено, возвращено None
print(bss.find_struct(35))  # Обнаружено, возвращен узел (35)
```

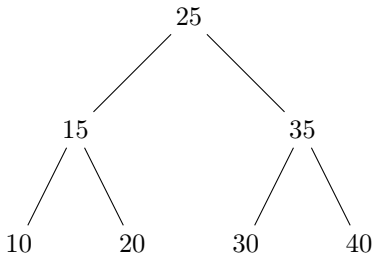


Рис. 10: Пример бинарного дерева поиска

11. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса NodeElement, которые представляют узлы дерева, и класса TreeIndex, который представляет дерево поиска. Класс TreeIndex должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс NodeElement с методом __init__, который принимает параметр elem_value и сохраняет его в атрибуте self.index_key. Атрибуты self.left и self.right должны быть инициализированы как None.

(b) Создайте класс TreeIndex с методом __init__, который инициализирует атрибут self.root_elem как None.

(c) Создайте публичный метод add_key в классе TreeIndex, который вставляет значение в дерево. Если self.root_elem отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_key_rec, передав ему self.root_elem и значение.

(d) Создайте приватный метод _add_key_rec в классе TreeIndex, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод get_key в классе TreeIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _get_key_rec, передав ему self.root_elem и искомое значение.

(f) Создайте приватный метод _get_key_rec в классе TreeIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _get_key_rec для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса TreeIndex и вставьте в него 23 случайных числа от 3 до 43.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ti = TreeIndex()
for i in range(23):
    ti.add_key(random.randint(3, 43))

print("Поиск элементов:")
print(ti.get_key(16))  # Обнаружено, возвращен узел (16)
print(ti.get_key(56))  # Не обнаружено, возвращено None
print(ti.get_key(36))  # Обнаружено, возвращен узел (36)
```

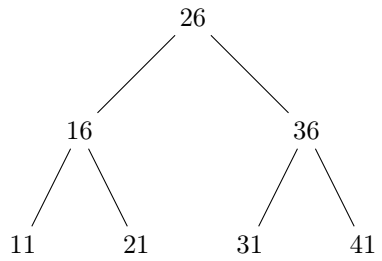


Рис. 11: Пример бинарного дерева поиска

12. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса BinElement, которые представляют узлы дерева, и класса IndexTree, который представляет дерево поиска. Класс IndexTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс BinElement с методом __init__, который принимает параметр bin_val и сохраняет его в атрибуте self.key_value. Атрибуты self.left_node и self.right_node должны быть инициализированы как None.

(b) Создайте класс IndexTree с методом __init__, который инициализирует атрибут self.first_element как None.

(c) Создайте публичный метод insert_key в классе IndexTree, который вставляет значение в дерево. Если self.first_element отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_key_helper, передав ему self.first_element и значение.

(d) Создайте приватный метод _insert_key_helper в классе IndexTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_key в классе IndexTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_key_helper, передав ему self.first_element и искомое значение.

(f) Создайте приватный метод _search_key_helper в классе IndexTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_key_helper для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса IndexTree и вставьте в него 24 случайных числа от 4 до 44.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
it = IndexTree()
for i in range(24):
    it.insert_key(random.randint(4, 44))

print("Поиск элементов:")
print(it.search_key(17))  # Обнаружено, возвращен узел (17)
print(it.search_key(57))  # Не обнаружено, возвращено None
print(it.search_key(37))  # Обнаружено, возвращен узел (37)
```

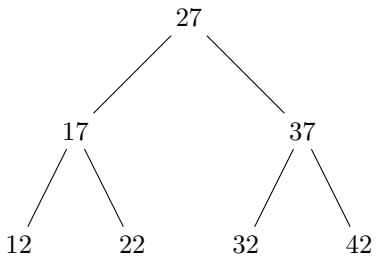


Рис. 12: Пример бинарного дерева поиска

13. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса SearchNode, которые представляют узлы дерева, и класса BinaryTreeIndex, который представляет дерево поиска. Класс BinaryTreeIndex должен содержать методы для вставки, поиска и удаления

элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс SearchNode с методом __init__, который принимает параметр search_val и сохраняет его в атрибуте self.node_key. Атрибуты self.left_child и self.right_child должны быть инициализированы как None.

(b) Создайте класс BinaryTreeIndex с методом __init__, который инициализирует атрибут self.initial_element как None.

(c) Создайте публичный метод add_node в классе BinaryTreeIndex, который вставляет значение в дерево. Если self.initial_element отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_node_recursive, передав ему self.initial_element и значение.

(d) Создайте приватный метод _add_node_recursive в классе BinaryTreeIndex, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_node в классе BinaryTreeIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_node_recursive, передав ему self.initial_element и искомое значение.

(f) Создайте приватный метод _find_node_recursive в классе BinaryTreeIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_node_recursive для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса BinaryTreeIndex и вставьте в него 25 случайных чисел от 5 до 45.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
bti = BinaryTreeIndex()
for i in range(25):
    bti.add_node(random.randint(5, 45))

print("Поиск элементов:")
print(bti.find_node(18))  # Обнаружено, возвращен узел (18)
print(bti.find_node(58))  # Не обнаружено, возвращено None
print(bti.find_node(38))  # Обнаружено, возвращен узел (38)
```

14. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса IndexNode, которые представляют узлы дерева, и класса SearchStructure, который представляет дерево поиска. Класс SearchStructure должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

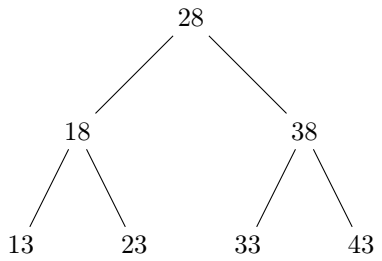


Рис. 13: Пример бинарного дерева поиска

Инструкции:

(a) Создайте класс IndexNode с методом __init__, который принимает параметр idx_value и сохраняет его в атрибуте self.element_key. Атрибуты self.left_elem и self.right_elem должны быть инициализированы как None.

(b) Создайте класс SearchStructure с методом __init__, который инициализирует атрибут self.start_element как None.

(c) Создайте публичный метод insert_elem в классе SearchStructure, который вставляет значение в дерево. Если self.start_element отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_elem_rec, передав ему self.start_element и значение.

(d) Создайте приватный метод _insert_elem_rec в классе SearchStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_elem в классе SearchStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_elem_rec, передав ему self.start_element и искомое значение.

(f) Создайте приватный метод _locate_elem_rec в классе SearchStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_elem_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса SearchStructure и вставьте в него 26 случайных чисел от 6 до 46.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ss = SearchStructure()
for i in range(26):
    ss.insert_elem(random.randint(6, 46))

print("Поиск элементов:")
print(ss.locate_elem(19))  # Обнаружено, возвращен узел (19)
print(ss.locate_elem(59))  # Не обнаружено, возвращено None
print(ss.locate_elem(39))  # Обнаружено, возвращен узел (39)
```

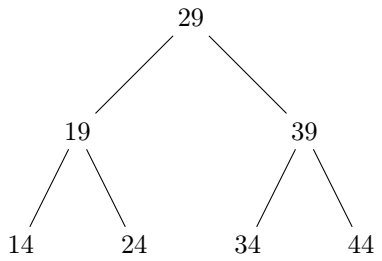


Рис. 14: Пример бинарного дерева поиска

15. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса KeyValueNode, которые представляют узлы дерева, и класса BinaryTreeMap, который представляет дерево поиска. Класс BinaryTreeMap должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс KeyValueNode с методом __init__, который принимает параметр key_val и сохраняет его в атрибуте self.map_key. Атрибуты self.left_branch и self.right_branch должны быть инициализированы как None.

    (b) Создайте класс BinaryTreeMap с методом __init__, который инициализирует атрибут self.root_key как None.

    (c) Создайте публичный метод put_key в классе BinaryTreeMap, который вставляет значение в дерево. Если self.root_key отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _put_key_helper, передав ему self.root_key и значение.

    (d) Создайте приватный метод _put_key_helper в классе BinaryTreeMap, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод get_key в классе BinaryTreeMap, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _get_key_helper, передав ему self.root_key и искомое значение.

    (f) Создайте приватный метод _get_key_helper в классе BinaryTreeMap, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _get_key_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

    (g) Создайте экземпляр класса BinaryTreeMap и вставьте в него 27 случайных чисел от 7 до 47.

    (h) Выполните поиск элементов в дереве и выведите результаты на экран.

    Пример использования:

```
btm = BinaryTreeMap()
for i in range(27):
    btm.put_key(random.randint(7, 47))

print("Поиск элементов:")
print(btm.get_key(20))  # Обнаружено, возвращен узел (20)
print(btm.get_key(60))  # Не обнаружено, возвращено None
print(btm.get_key(40))  # Обнаружено, возвращен узел (40)
```

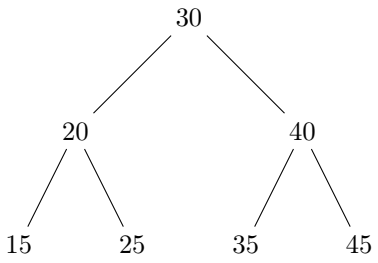


Рис. 15: Пример бинарного дерева поиска

16. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса MapNode, которые представляют узлы дерева, и класса KeyTree, который представляет дерево поиска. Класс KeyTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс MapNode с методом __init__, который принимает параметр map_value и сохраняет его в атрибуте self.tree_key. Атрибуты self.left_part и self.right_part должны быть инициализированы как None.

    (b) Создайте класс KeyTree с методом __init__, который инициализирует атрибут self.base_key как None.

    (c) Создайте публичный метод insert_map в классе KeyTree, который вставляет значение в дерево. Если self.base_key отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_map_rec, передав ему self.base_key и значение.

    (d) Создайте приватный метод _insert_map_rec в классе KeyTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод search_map в классе KeyTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_map_rec, передав ему self.base_key и искомое значение.

    (f) Создайте приватный метод _search_map_rec в классе KeyTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае,

рекурсивно вызывайте метод _search_map_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса KeyTree и вставьте в него 28 случайных чисел от 8 до 48.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
kt = KeyTree()
for i in range(28):
    kt.insert_map(random.randint(8, 48))

print("Поиск элементов:")
print(kt.search_map(21))  # Обнаружено, возвращен узел (21)
print(kt.search_map(61))  # Не обнаружено, возвращено None
print(kt.search_map(41))  # Обнаружено, возвращен узел (41)
```

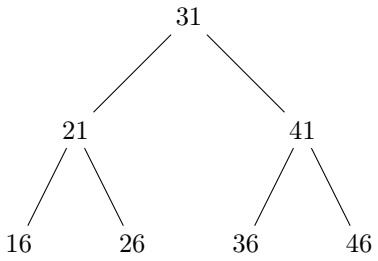


Рис. 16: Пример бинарного дерева поиска

17. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса TreeKeyNode, которые представляют узлы дерева, и класса ValueTree, который представляет дерево поиска. Класс ValueTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс TreeKeyNode с методом __init__, который принимает параметр tree_key_val и сохраняет его в атрибуте self.value_key. Атрибуты self.left и self.right должны быть инициализированы как None.

(b) Создайте класс ValueTree с методом __init__, который инициализирует атрибут self.first_key как None.

(c) Создайте публичный метод add_value в классе ValueTree, который вставляет значение в дерево. Если self.first_key отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_value_helper, передав ему self.first_key и значение.

(d) Создайте приватный метод _add_value_helper в классе ValueTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению

текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод retrieve_value в классе ValueTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _retrieve_value_helper, передав ему self.first_key и искомое значение.

(f) Создайте приватный метод _retrieve_value_helper в классе ValueTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _retrieve_value_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса ValueTree и вставьте в него 29 случайных чисел от 9 до 49.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
vt = ValueTree()
for i in range(29):
    vt.add_value(random.randint(9, 49))

print("Поиск элементов:")
print(vt.retrieve_value(22))  # Обнаружено, возвращен узел (22)
print(vt.retrieve_value(62))  # Не обнаружено, возвращено None
print(vt.retrieve_value(42))  # Обнаружено, возвращен узел (42)
```

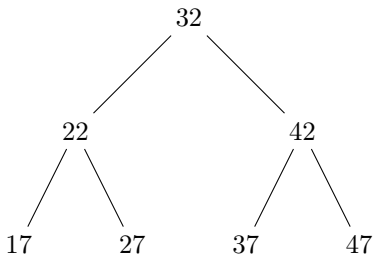


Рис. 17: Пример бинарного дерева поиска

18. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса ValueNode, которые представляют узлы дерева, и класса KeyedTree, который представляет дерево поиска. Класс KeyedTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс ValueNode с методом __init__, который принимает параметр node_value и сохраняет его в атрибуте self.keyed_value. Атрибуты self.left_side и self.right_side должны быть инициализированы как None.

(b) Создайте класс KeyedTree с методом __init__, который инициализирует атрибут self.start_key как None.

(c) Создайте публичный метод store_value в классе KeyedTree, который вставляет значение в дерево. Если self.start_key отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _store_value_rec, передав ему self.start_key и значение.

(d) Создайте приватный метод _store_value_rec в классе KeyedTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод fetch_value в классе KeyedTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _fetch_value_rec, передав ему self.start_key и искомое значение.

(f) Создайте приватный метод _fetch_value_rec в классе KeyedTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _fetch_value_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса KeyedTree и вставьте в него 30 случайных чисел от 10 до 50.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
kt = KeyedTree()
for i in range(30):
    kt.store_value(random.randint(10, 50))

print("Поиск элементов:")
print(kt.fetch_value(23))  # Обнаружено, возвращен узел (23)
print(kt.fetch_value(63))  # Не обнаружено, возвращено None
print(kt.fetch_value(43))  # Обнаружено, возвращен узел (43)
```

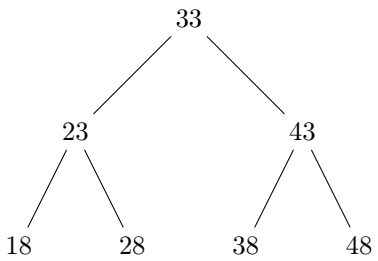


Рис. 18: Пример бинарного дерева поиска

19. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса KeyedNode, которые представляют узлы дерева, и класса ValuedTree, который представляет дерево поиска.

Класс ValuedTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс KeyedNode с методом __init__, который принимает параметр keyed_val и сохраняет его в атрибуте self.node_content. Атрибуты self.left_path и self.right_path должны быть инициализированы как None.

(b) Создайте класс ValuedTree с методом __init__, который инициализирует атрибут self.root_content как None.

(c) Создайте публичный метод insert_content в классе ValuedTree, который вставляет значение в дерево. Если self.root_content отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_content_helper, передав ему self.root_content и значение.

(d) Создайте приватный метод _insert_content_helper в классе ValuedTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_content в классе ValuedTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_content_helper, передав ему self.root_content и искомое значение.

(f) Создайте приватный метод _search_content_helper в классе ValuedTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_content_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса ValuedTree и вставьте в него 31 случайное число от 11 до 51.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
vt = ValuedTree()
for i in range(31):
    vt.insert_content(random.randint(11, 51))

print("Поиск элементов:")
print(vt.search_content(24))  # Обнаружено, возвращен узел (24)
print(vt.search_content(64))  # Не обнаружено, возвращено None
print(vt.search_content(44))  # Обнаружено, возвращен узел (44)
```

20. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса ContentNode, которые представляют узлы дерева, и класса KeyTreeStructure, который представляет дерево поиска. Класс KeyTreeStructure должен содержать методы для вставки, поиска и

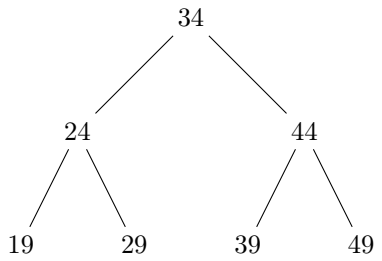


Рис. 19: Пример бинарного дерева поиска

удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс ContentNode с методом __init__, который принимает параметр content_val и сохраняет его в атрибуте self.node_data. Атрибуты self.left_item и self.right_item должны быть инициализированы как None.

(b) Создайте класс KeyTreeStructure с методом __init__, который инициализирует атрибут self.top_data как None.

(c) Создайте публичный метод add_data в классе KeyTreeStructure, который вставляет значение в дерево. Если self.top_data отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_data_rec, передав ему self.top_data и значение.

(d) Создайте приватный метод _add_data_rec в классе KeyTreeStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_data в классе KeyTreeStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_data_rec, передав ему self.top_data и искомое значение.

(f) Создайте приватный метод _find_data_rec в классе KeyTreeStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_data_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса KeyTreeStructure и вставьте в него 32 случайных числа от 12 до 52.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
kts = KeyTreeStructure()
for i in range(32):
    kts.add_data(random.randint(12, 52))
```

```
print("Поиск элементов:")
print(kts.find_data(25))  # Обнаружено, возвращен узел (25)
print(kts.find_data(65))  # Не обнаружено, возвращено None
print(kts.find_data(45))  # Обнаружено, возвращен узел (45)
```

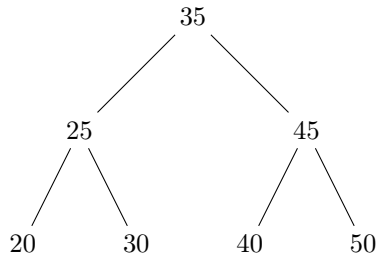


Рис. 20: Пример бинарного дерева поиска

21. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса DataNode, которые представляют узлы дерева, и класса ContentTree, который представляет дерево поиска. Класс ContentTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс DataNode с методом __init__, который принимает параметр data_value и сохраняет его в атрибуте self.tree_content. Атрибуты self.left_entry и self.right_entry должны быть инициализированы как None.

    (b) Создайте класс ContentTree с методом __init__, который инициализирует атрибут self.root_content как None.

    (c) Создайте публичный метод insert_entry в классе ContentTree, который вставляет значение в дерево. Если self.root_content отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_entry_helper, передав ему self.root_content и значение.

    (d) Создайте приватный метод _insert_entry_helper в классе ContentTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод search_entry в классе ContentTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_entry_helper, передав ему self.root_content и искомое значение.

    (f) Создайте приватный метод _search_entry_helper в классе ContentTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_entry_helper для поиска значения

в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса ContentTree и вставьте в него 33 случайных числа от 13 до 53.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ct = ContentTree()
for i in range(33):
    ct.insert_entry(random.randint(13, 53))

print("Поиск элементов:")
print(ct.search_entry(26))  # Обнаружено, возвращен узел (26)
print(ct.search_entry(66))  # Не обнаружено, возвращено None
print(ct.search_entry(46))  # Обнаружено, возвращен узел (46)
```

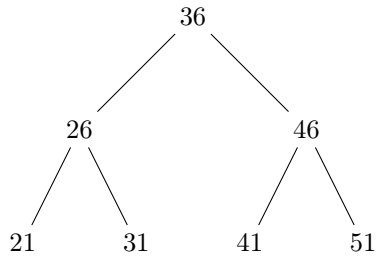


Рис. 21: Пример бинарного дерева поиска

22. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса EntryNode, которые представляют узлы дерева, и класса DataStructureTree, который представляет дерево поиска. Класс DataStructureTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс EntryNode с методом __init__, который принимает параметр entry_val и сохраняет его в атрибуте self.content_item. Атрибуты self.left_data и self.right_data должны быть инициализированы как None.

(b) Создайте класс DataStructureTree с методом __init__, который инициализирует атрибут self.first_item как None.

(c) Создайте публичный метод add_item в классе DataStructureTree, который вставляет значение в дерево. Если self.first_item отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_item_rec, передав ему self.first_item и значение.

(d) Создайте приватный метод _add_item_rec в классе DataStructureTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_item в классе DataStructureTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_item_rec, передав ему self.first_item и искомое значение.

(f) Создайте приватный метод _locate_item_rec в классе DataStructureTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_item_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса DataStructureTree и вставьте в него 34 случайных числа от 14 до 54.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
dst = DataStructureTree()
for i in range(34):
    dst.add_item(random.randint(14, 54))

print("Поиск элементов:")
print(dst.locate_item(27))  # Обнаружено, возвращен узел (27)
print(dst.locate_item(67))  # Не обнаружено, возвращено None
print(dst.locate_item(47))  # Обнаружено, возвращен узел (47)
```

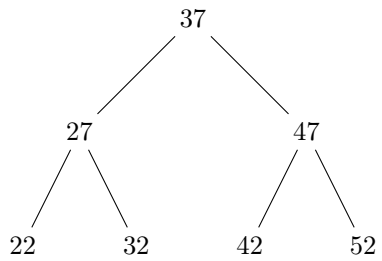


Рис. 22: Пример бинарного дерева поиска

23. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса ItemNode, которые представляют узлы дерева, и класса EntryTree, который представляет дерево поиска. Класс EntryTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс ItemNode с методом __init__, который принимает параметр item_value и сохраняет его в атрибуте self.data_entry. Атрибуты self.left_position и self.right_position должны быть инициализированы как None.

(b) Создайте класс EntryTree с методом __init__, который инициализирует атрибут self.root_entry как None.

(c) Создайте публичный метод insert_position в классе EntryTree, который вставляет значение в дерево. Если self.root_entry отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_position_helper, передав ему self.root_entry и значение.

(d) Создайте приватный метод _insert_position_helper в классе EntryTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_position в классе EntryTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_position_helper, передав ему self.root_entry и искомое значение.

(f) Создайте приватный метод _find_position_helper в классе EntryTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_position_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса EntryTree и вставьте в него 35 случайных чисел от 15 до 55.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
et = EntryTree()
for i in range(35):
    et.insert_position(random.randint(15, 55))

print("Поиск элементов:")
print(et.find_position(28))  # Обнаружено, возвращен узел (28)
print(et.find_position(68))  # Не обнаружено, возвращено None
print(et.find_position(48))  # Обнаружено, возвращен узел (48)
```

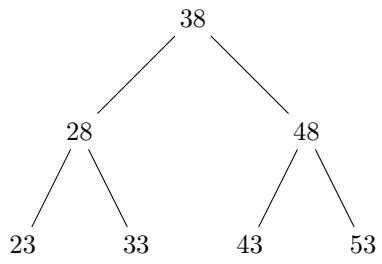


Рис. 23: Пример бинарного дерева поиска

24. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса PositionNode, которые представляют узлы дерева, и класса ItemStructure, который представляет дерево поиска. Класс ItemStructure должен содержать методы для вставки, поиска и удаления

элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс PositionNode с методом __init__, который принимает параметр position_val и сохраняет его в атрибуте self.entry_data. Атрибуты self.left_slot и self.right_slot должны быть инициализированы как None.

(b) Создайте класс ItemStructure с методом __init__, который инициализирует атрибут self.top_entry как None.

(c) Создайте публичный метод add_slot в классе ItemStructure, который вставляет значение в дерево. Если self.top_entry отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_slot_rec, передав ему self.top_entry и значение.

(d) Создайте приватный метод _add_slot_rec в классе ItemStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_slot в классе ItemStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_slot_rec, передав ему self.top_entry и искомое значение.

(f) Создайте приватный метод _search_slot_rec в классе ItemStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_slot_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса ItemStructure и вставьте в него 36 случайных чисел от 16 до 56.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
is_ = ItemStructure()
for i in range(36):
    is_.add_slot(random.randint(16, 56))

print("Поиск элементов:")
print(is_.search_slot(29))  # Обнаружено, возвращен узел (29)
print(is_.search_slot(69))  # Не обнаружено, возвращено None
print(is_.search_slot(49))  # Обнаружено, возвращен узел (49)
```

25. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса SlotNode, которые представляют узлы дерева, и класса PositionTree, который представляет дерево поиска. Класс PositionTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

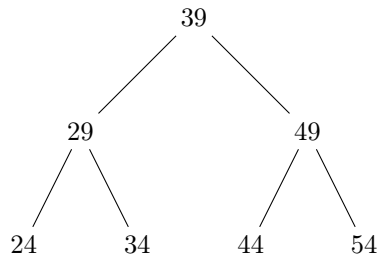


Рис. 24: Пример бинарного дерева поиска

Инструкции:

(a) Создайте класс SlotNode с методом __init__, который принимает параметр slot_value и сохраняет его в атрибуте self.item_position. Атрибуты self.left_place и self.right_place должны быть инициализированы как None.

(b) Создайте класс PositionTree с методом __init__, который инициализирует атрибут self.first_position как None.

(c) Создайте публичный метод insert_place в классе PositionTree, который вставляет значение в дерево. Если self.first_position отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_place_helper, передав ему self.first_position и значение.

(d) Создайте приватный метод _insert_place_helper в классе PositionTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_place в классе PositionTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_place_helper, передав ему self.first_position и искомое значение.

(f) Создайте приватный метод _locate_place_helper в классе PositionTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_place_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса PositionTree и вставьте в него 37 случайных чисел от 17 до 57.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
pt = PositionTree()
for i in range(37):
    pt.insert_place(random.randint(17, 57))

print("Поиск элементов:")
print(pt.locate_place(30))  # Обнаружено, возвращен узел (30)
print(pt.locate_place(70))  # Не обнаружено, возвращено None
print(pt.locate_place(50))  # Обнаружено, возвращен узел (50)
```

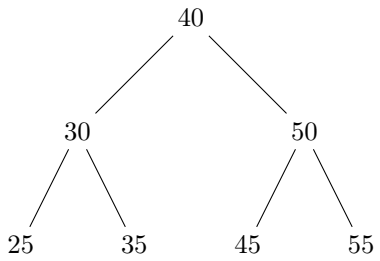


Рис. 25: Пример бинарного дерева поиска

26. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса PlaceNode, которые представляют узлы дерева, и класса SlotTree, который представляет дерево поиска. Класс SlotTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс PlaceNode с методом __init__, который принимает параметр place_val и сохраняет его в атрибуте self.position_item. Атрибуты self.left_spot и self.right_spot должны быть инициализированы как None.

    (b) Создайте класс SlotTree с методом __init__, который инициализирует атрибут self.root_position как None.

    (c) Создайте публичный метод add_spot в классе SlotTree, который вставляет значение в дерево. Если self.root_position отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_spot_rec, передав ему self.root_position и значение.

    (d) Создайте приватный метод _add_spot_rec в классе SlotTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод find_spot в классе SlotTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_spot_rec, передав ему self.root_position и искомое значение.

    (f) Создайте приватный метод _find_spot_rec в классе SlotTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_spot_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

    (g) Создайте экземпляр класса SlotTree и вставьте в него 38 случайных чисел от 18 до 58.

    (h) Выполните поиск элементов в дереве и выведите результаты на экран.

    Пример использования:

```
st = SlotTree()
for i in range(38):
    st.add_spot(random.randint(18, 58))

print("Поиск элементов:")
print(st.find_spot(31))  # Обнаружено, возвращен узел (31)
print(st.find_spot(71))  # Не обнаружено, возвращено None
print(st.find_spot(51))  # Обнаружено, возвращен узел (51)
```

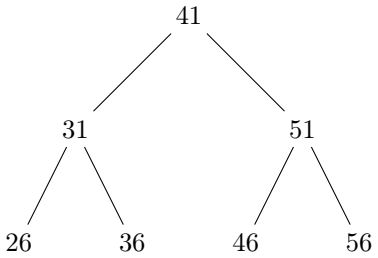


Рис. 26: Пример бинарного дерева поиска

27. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса SpotNode, которые представляют узлы дерева, и класса PlaceIndex, который представляет дерево поиска. Класс PlaceIndex должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс SpotNode с методом __init__, который принимает параметр spot_value и сохраняет его в атрибуте self.index_position. Атрибуты self.left_location и self.right_location должны быть инициализированы как None.

    (b) Создайте класс PlaceIndex с методом __init__, который инициализирует атрибут self.start_position как None.

    (c) Создайте публичный метод insert_location в классе PlaceIndex, который вставляет значение в дерево. Если self.start_position отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_location_helper, передав ему self.start_position и значение.

    (d) Создайте приватный метод _insert_location_helper в классе PlaceIndex, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод search_location в классе PlaceIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_location_helper, передав ему self.start_position и искомое значение.

    (f) Создайте приватный метод _search_location_helper в классе PlaceIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение

текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_location_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса PlaceIndex и вставьте в него 39 случайных чисел от 19 до 59.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
pi = PlaceIndex()
for i in range(39):
    pi.insert_location(random.randint(19, 59))

print("Поиск элементов:")
print(pi.search_location(32))  # Обнаружено, возвращен узел (32)
print(pi.search_location(72))  # Не обнаружено, возвращено None
print(pi.search_location(52))  # Обнаружено, возвращен узел (52)
```

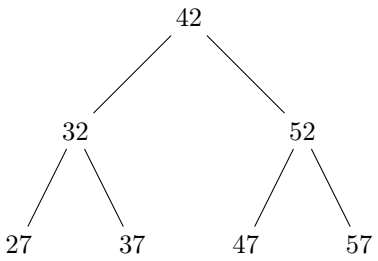


Рис. 27: Пример бинарного дерева поиска

28. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса LocationNode, которые представляют узлы дерева, и класса SpotTree, который представляет дерево поиска. Класс SpotTree должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс LocationNode с методом __init__, который принимает параметр location_val и сохраняет его в атрибуте self.tree_spot. Атрибуты self.left_site и self.right_site должны быть инициализированы как None.

(b) Создайте класс SpotTree с методом __init__, который инициализирует атрибут self.base_spot как None.

(c) Создайте публичный метод add_site в классе SpotTree, который вставляет значение в дерево. Если self.base_spot отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_site_rec, передав ему self.base_spot и значение.

(d) Создайте приватный метод _add_site_rec в классе SpotTree, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_site в классе SpotTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_site_rec, передав ему self.base_spot и искомое значение.

(f) Создайте приватный метод _locate_site_rec в классе SpotTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_site_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса SpotTree и вставьте в него 40 случайных чисел от 20 до 60.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
spot_tree = SpotTree()
for i in range(40):
    spot_tree.add_site(random.randint(20, 60))

print("Поиск элементов:")
print(spot_tree.locate_site(33))  # Обнаружено, возвращен узел (33)
print(spot_tree.locate_site(73))  # Не обнаружено, возвращено None
print(spot_tree.locate_site(53))  # Обнаружено, возвращен узел (53)
```

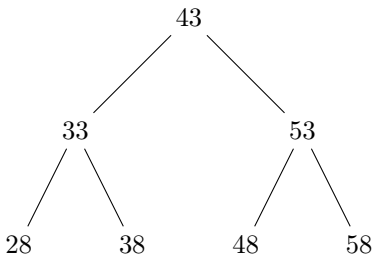


Рис. 28: Пример бинарного дерева поиска

29. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса SiteNode, которые представляют узлы дерева, и класса LocationIndex, который представляет дерево поиска. Класс LocationIndex должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс SiteNode с методом __init__, который принимает параметр site_value и сохраняет его в атрибуте self.index_location. Атрибуты self.left_zone и self.right_zone должны быть инициализированы как None.

(b) Создайте класс LocationIndex с методом __init__, который инициализирует атрибут self.root_location как None.

(c) Создайте публичный метод insert_zone в классе LocationIndex, который вставляет значение в дерево. Если self.root_location отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_zone_helper, передав ему self.root_location и значение.

(d) Создайте приватный метод _insert_zone_helper в классе LocationIndex, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_zone в классе LocationIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_zone_helper, передав ему self.root_location и искомое значение.

(f) Создайте приватный метод _find_zone_helper в классе LocationIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_zone_helper для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса LocationIndex и вставьте в него 41 случайное число от 21 до 61.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
li = LocationIndex()
for i in range(41):
    li.insert_zone(random.randint(21, 61))

print("Поиск элементов:")
print(li.find_zone(34))  # Обнаружено, возвращен узел (34)
print(li.find_zone(74))  # Не обнаружено, возвращено None
print(li.find_zone(54))  # Обнаружено, возвращен узел (54)
```

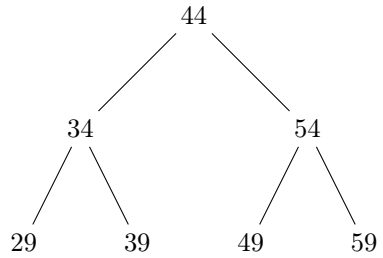


Рис. 29: Пример бинарного дерева поиска

30. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса ZoneNode, которые представляют узлы дерева, и класса SiteStructure, который представляет дерево поиска.

Класс SiteStructure должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс ZoneNode с методом __init__, который принимает параметр zone_val и сохраняет его в атрибуте self.structure_site. Атрибуты self.left_region и self.right_region должны быть инициализированы как None.

(b) Создайте класс SiteStructure с методом __init__, который инициализирует атрибут self.top_site как None.

(c) Создайте публичный метод add_region в классе SiteStructure, который вставляет значение в дерево. Если self.top_site отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_region_rec, передав ему self.top_site и значение.

(d) Создайте приватный метод _add_region_rec в классе SiteStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_region в классе SiteStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_region_rec, передав ему self.top_site и искомое значение.

(f) Создайте приватный метод _search_region_rec в классе SiteStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_region_rec для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса SiteStructure и вставьте в него 42 случайных числа от 22 до 62.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ss = SiteStructure()
for i in range(42):
    ss.add_region(random.randint(22, 62))

print("Поиск элементов:")
print(ss.search_region(35))  # Обнаружено, возвращен узел (35)
print(ss.search_region(75))  # Не обнаружено, возвращено None
print(ss.search_region(55))  # Обнаружено, возвращен узел (55)
```

31. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса RegionNode, которые представляют узлы дерева, и класса ZoneTree, который представляет дерево поиска. Класс ZoneTree должен содержать методы для вставки, поиска и удаления элементов, при

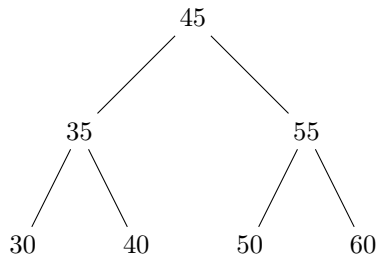


Рис. 30: Пример бинарного дерева поиска

этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс RegionNode с методом __init__, который принимает параметр region_value и сохраняет его в атрибуте self.tree_zone. Атрибуты self.left_area и self.right_area должны быть инициализированы как None.

(b) Создайте класс ZoneTree с методом __init__, который инициализирует атрибут self.first_zone как None.

(c) Создайте публичный метод insert_area в классе ZoneTree, который вставляет значение в дерево. Если self.first_zone отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_area_helper, передав ему self.first_zone и значение.

(d) Создайте приватный метод _insert_area_helper в классе ZoneTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_area в классе ZoneTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_area_rec, передав ему self.first_zone и искомое значение.

(f) Создайте приватный метод _locate_area_rec в классе ZoneTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_area_rec для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса ZoneTree и вставьте в него 43 случайных числа от 23 до 63.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
zt = ZoneTree()
for i in range(43):
    zt.insert_area(random.randint(23, 63))
```

```
print("Поиск элементов:")
print(zt.locate_area(36))  # Обнаружено, возвращен узел (36)
print(zt.locate_area(76))  # Не обнаружено, возвращено None
print(zt.locate_area(56))  # Обнаружено, возвращен узел (56)
```

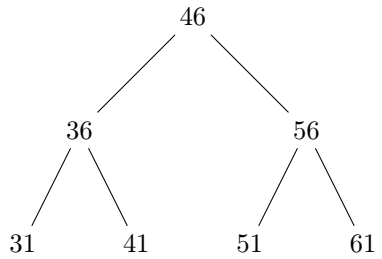


Рис. 31: Пример бинарного дерева поиска

32. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса AreaNode, которые представляют узлы дерева, и класса RegionIndex, который представляет дерево поиска. Класс RegionIndex должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

    Инструкции:

    (a) Создайте класс AreaNode с методом __init__, который принимает параметр area_val и сохраняет его в атрибуте self.index_region. Атрибуты self.left_district и self.right_district должны быть инициализированы как None.

    (b) Создайте класс RegionIndex с методом __init__, который инициализирует атрибут self.root_region как None.

    (c) Создайте публичный метод add_district в классе RegionIndex, который вставляет значение в дерево. Если self.root_region отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_district_rec, передав ему self.root_region и значение.

    (d) Создайте приватный метод _add_district_rec в классе RegionIndex, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

    (e) Создайте публичный метод find_district в классе RegionIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_district_helper, передав ему self.root_region и искомое значение.

    (f) Создайте приватный метод _find_district_helper в классе RegionIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_district_helper для поиска значения

в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса RegionIndex и вставьте в него 44 случайных числа от 24 до 64.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ri = RegionIndex()
for i in range(44):
    ri.add_district(random.randint(24, 64))

print("Поиск элементов:")
print(ri.find_district(37))  # Обнаружено, возвращен узел (37)
print(ri.find_district(77))  # Не обнаружено, возвращено None
print(ri.find_district(57))  # Обнаружено, возвращен узел (57)
```

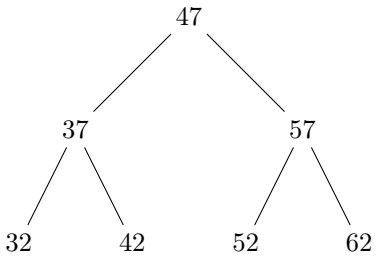


Рис. 32: Пример бинарного дерева поиска

33. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса DistrictNode, которые представляют узлы дерева, и класса AreaTree, который представляет дерево поиска. Класс AreaTree должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс DistrictNode с методом __init__, который принимает параметр district_value и сохраняет его в атрибуте self.tree_area. Атрибуты self.left_sector и self.right_sector должны быть инициализированы как None.

(b) Создайте класс AreaTree с методом __init__, который инициализирует атрибут self.start_area как None.

(c) Создайте публичный метод insert_sector в классе AreaTree, который вставляет значение в дерево. Если self.start_area отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_sector_helper, передав ему self.start_area и значение.

(d) Создайте приватный метод _insert_sector_helper в классе AreaTree, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод search_sector в классе AreaTree, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _search_sector_rec, передав ему self.start_area и искомое значение.

(f) Создайте приватный метод _search_sector_rec в классе AreaTree, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _search_sector_rec для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса AreaTree и вставьте в него 45 случайных чисел от 25 до 65.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
at = AreaTree()
for i in range(45):
    at.insert_sector(random.randint(25, 65))

print("Поиск элементов:")
print(at.search_sector(38))  # Обнаружено, возвращен узел (38)
print(at.search_sector(78))  # Не обнаружено, возвращено None
print(at.search_sector(58))  # Обнаружено, возвращен узел (58)
```

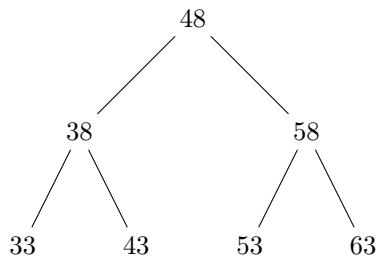


Рис. 33: Пример бинарного дерева поиска

34. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса SectorNode, которые представляют узлы дерева, и класса DistrictStructure, который представляет дерево поиска. Класс DistrictStructure должен содержать методы для вставки, поиска и удаления элементов, при этом все вспомогательные методы должны быть приватными. Программа также должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс SectorNode с методом __init__, который принимает параметр sector_val и сохраняет его в атрибуте self.structure_district. Атрибуты self.left_block и self.right_block должны быть инициализированы как None.

(b) Создайте класс DistrictStructure с методом __init__, который инициализирует атрибут self.top_district как None.

(c) Создайте публичный метод add_block в классе DistrictStructure, который вставляет значение в дерево. Если self.top_district отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _add_block_rec, передав ему self.top_district и значение.

(d) Создайте приватный метод _add_block_rec в классе DistrictStructure, который рекурсивно вставляет значение в дерево. Если значение строго меньше значения текущего узла, вставьте его в левое поддерево. Если значение больше или равно значению текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод locate_block в классе DistrictStructure, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _locate_block_helper, передав ему self.top_district и искомое значение.

(f) Создайте приватный метод _locate_block_helper в классе DistrictStructure, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _locate_block_helper для поиска значения в левом поддереве (если искомое значение меньше текущего) или в правом поддереве (если искомое значение больше или равно текущему).

(g) Создайте экземпляр класса DistrictStructure и вставьте в него 46 случайных чисел от 26 до 66.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
ds = DistrictStructure()
for i in range(46):
    ds.add_block(random.randint(26, 66))

print("Поиск элементов:")
print(ds.locate_block(39))  # Обнаружено, возвращен узел (39)
print(ds.locate_block(79))  # Не обнаружено, возвращено None
print(ds.locate_block(59))  # Обнаружено, возвращен узел (59)
```

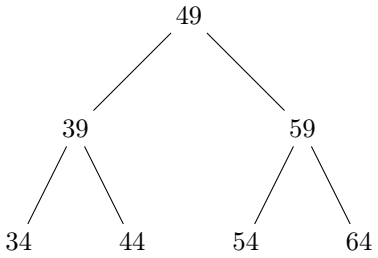


Рис. 34: Пример бинарного дерева поиска

35. Написать программу на Python, которая реализует бинарное дерево поиска с инкапсуляцией. Программа должна создавать экземпляры класса BlockNode, которые представляют узлы дерева, и класса SectorIndex, который представляет дерево поиска. Класс SectorIndex должен содержать методы для вставки, поиска и удаления элементов, при этом все рекурсивные методы должны быть приватными. Программа также

должна создавать дерево поиска, вставлять в него случайные числа и выполнять поиск элементов в дереве.

Инструкции:

(a) Создайте класс BlockNode с методом __init__, который принимает параметр block_value и сохраняет его в атрибуте self.index_sector. Атрибуты self.left_unit и self.right_unit должны быть инициализированы как None.

(b) Создайте класс SectorIndex с методом __init__, который инициализирует атрибут self.root_sector как None.

(c) Создайте публичный метод insert_unit в классе SectorIndex, который вставляет значение в дерево. Если self.root_sector отсутствует, создайте новый узел с вставляемым значением. В противном случае, вызовите приватный метод _insert_unit_helper, передав ему self.root_sector и значение.

(d) Создайте приватный метод _insert_unit_helper в классе SectorIndex, который рекурсивно вставляет значение в дерево. Если значение меньше или равно значению текущего узла, вставьте его в левое поддерево. Если значение строго больше значения текущего узла, вставьте его в правое поддерево.

(e) Создайте публичный метод find_unit в классе SectorIndex, который ищет значение в дереве. Если дерево пустое, верните None. В противном случае, вызовите приватный метод _find_unit_rec, передав ему self.root_sector и искомое значение.

(f) Создайте приватный метод _find_unit_rec в классе SectorIndex, который рекурсивно ищет значение в дереве. Если текущий узел равен None или значение текущего узла равно искомому значению, верните текущий узел. В противном случае, рекурсивно вызывайте метод _find_unit_rec для поиска значения в левом поддереве (если искомое значение меньше или равно текущему) или в правом поддереве (если искомое значение больше).

(g) Создайте экземпляр класса SectorIndex и вставьте в него 47 случайных чисел от 27 до 67.

(h) Выполните поиск элементов в дереве и выведите результаты на экран.

Пример использования:

```
si = SectorIndex()
for i in range(47):
    si.insert_unit(random.randint(27, 67))

print("Поиск элементов:")
print(si.find_unit(40))  # Обнаружено, возвращен узел (40)
print(si.find_unit(80))  # Не обнаружено, возвращено None
print(si.find_unit(60))  # Обнаружено, возвращен узел (60)
```

2.3.2 Задача 2 (стек)

1. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией внутреннего состояния. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

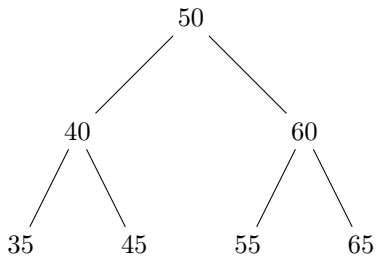


Рис. 35: Пример бинарного дерева поиска

Инструкции:

(a) Создайте класс Stack с методом __init__, который принимает необязательный аргумент initial_element. Если он передан, стек инициализируется с этим элементом (в виде списка из одного элемента), иначе — пустым списком.

(b) Создайте метод push, который принимает элемент в качестве аргумента и вталкивает его в стек только в том случае, если он не равен текущему верхнему элементу (если стек не пуст). Если стек пуст, элемент добавляется без проверки.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, метод должен вернуть None и вывести сообщение "Стек пуст — извлечение невозможно"в стандартный поток ошибок (sys.stderr).

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None и выводит сообщение "Стек пуст — просмотр невозможен"в sys.stderr.

(g) Создайте экземпляр класса Stack, передав в конструктор начальный элемент 10.

(h) Последовательно вызовите метод push с аргументами: 10, 20, 20, 30, 40 (обратите внимание, что повторяющийся элемент 20 не должен быть добавлен дважды подряд).

(i) Выведите размер стека и верхний элемент.

(j) Вызовите метод pop дважды, каждый раз выводя вытолкнутый элемент.

(k) После каждого pop выводите текущий размер стека и результат вызова peek.

Пример использования:

```
import sys

stack = Stack(10)
stack.push(10)    # не добавится, т.к. равен верхнему
stack.push(20)    # добавится
stack.push(20)    # не добавится, т.к. равен верхнему
stack.push(30)
stack.push(40)

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())
```

```
popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())
```

2. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Дополнительно принимает необязательный параметр max_size, ограничивающий максимальное количество элементов в стеке (по умолчанию — None, то есть без ограничений).

   (b) Создайте метод push, который принимает два аргумента: element и force=False. Элемент добавляется в стек, только если не превышает max_size. Если force=True, то элемент добавляется даже при превышении лимита (с заменой самого нижнего элемента, если стек полон).

   (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает строку "Стек пуст".

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

   (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает строку "Нет элементов для просмотра".

   (g) Создайте экземпляр класса Stack с max_size=3.

   (h) Последовательно вызовите push с элементами 5, 15, 25 (все добавятся).

   (i) Попытайтесь добавить 35 без force — не должно добавиться.

   (j) Добавьте 35 с force=True — должен замениться нижний элемент (5), стек станет [15, 25, 35].

   (k) Выведите размер стека и верхний элемент.

   (l) Вызовите pop и выведите результат.

   (m) Повторите вывод размера и верхнего элемента.

   Пример использования:

```
stack = Stack(max_size=3)
stack.push(5)
stack.push(15)
stack.push(25)
```

```
stack.push(35)           # не добавится
stack.push(35, force=True)  # добавится с заменой нижнего

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())
```

3. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Может принимать список элементов в качестве аргумента items, который будет использован для первоначального заполнения стека (в порядке, как в списке: первый элемент — внизу стека).

   (b) Создайте метод push, который принимает один элемент и добавляет его в стек. Если добавляемый элемент отрицательный, он не добавляется, а в sys.stderr выводится предупреждение "Отрицательные значения не допускаются".

   (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, выбрасывает исключение IndexError с сообщением "pop from empty stack".

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

   (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, выбрасывает исключение IndexError с сообщением "peek from empty stack".

   (g) Создайте экземпляр класса Stack, передав в конструктор список [1, 2, 3].

   (h) Добавьте элементы 4, -5 (не добавится), 6.

   (i) Выведите размер стека и результат peek.

   (j) Вызовите pop трижды, каждый раз выводя результат.

   (k) После каждого pop проверяйте is_empty и выводите результат.

   Пример использования:

```
import sys

stack = Stack([1, 2, 3])
stack.push(4)
stack.push(-5)  # не добавится, выведет предупреждение
stack.push(6)
```

```
print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

for _ in range(3):
    popped = stack.pop()
    print("Вытолкнут:", popped)
    print("Стек пуст?", stack.is_empty())
```

4. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает необязательный аргумент allow_duplicates (по умолчанию True). Если False, то дубликаты (элементы, уже присутствующие в стеке) не добавляются.

   (b) Создайте метод push, который принимает элемент и добавляет его в стек, только если allow_duplicates=True или если такого элемента еще нет в стеке. Возвращает True, если элемент добавлен, и False — если не добавлен.

   (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

   (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

   (g) Создайте экземпляр класса Stack с allow_duplicates=False.

   (h) Добавьте элементы 10, 20, 10 (второй 10 не добавится), 30.

   (i) Выведите размер стека и верхний элемент.

   (j) Вызовите pop, выведите результат.

   (k) Повторите вывод размера и верхнего элемента.

   Пример использования:

```
stack = Stack(allow_duplicates=False)
print(stack.push(10))  # True
print(stack.push(20))  # True
print(stack.push(10))  # False (дубликат)
print(stack.push(30))  # True

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())
```

5. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Может принимать параметр name (строка) для именования стека (используется только для отладки, не влияет на логику).

   (b) Создайте метод push, который принимает элемент и добавляет его в стек. Если элемент не является числом (int или float), он не добавляется, а в sys.stderr выводится сообщение "Только числовые значения разрешены".

   (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

   (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

   (g) Создайте экземпляр класса Stack с именем "NumericStack".

   (h) Добавьте элементы: 3.14, 42, "hello"(не добавится), 100, [1,2] (не добавится).

   (i) Выведите размер стека и верхний элемент.

   (j) Вызовите pop дважды, выводя каждый раз результат.

   (k) После каждого pop выводите размер стека.

   Пример использования:

```
import sys

stack = Stack(name="NumericStack")
stack.push(3.14)
stack.push(42)
stack.push("hello")    # не добавится
stack.push(100)
stack.push([1,2])      # не добавится

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())

popped = stack.pop()
print("Вытолкнут:", popped)
print("Размер после pop:", stack.size())
```

6. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает необязательный параметр auto_reverse=False. Если True, то при добавлении элемента он вставляется не наверх, а вниз стека (реализуя поведение, обратное обычному стеку).

   (b) Создайте метод push, который принимает элемент и добавляет его: если auto_reverse=False — наверх (как обычно), если True — вниз (в начало внутреннего списка).

   (c) Создайте метод pop, который выталкивает верхний элемент из стека (последний добавленный, если auto_reverse=False, или первый добавленный, если auto_reverse=True) и возвращает его. Если стек пуст, возвращает "EMPTY".

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

   (f) Создайте метод peek, который возвращает верхний элемент стека (последний в списке, если auto_reverse=False, или первый, если auto_reverse=True), если стек не пуст. Если стек пуст, возвращает "NO ELEMENT".

   (g) Создайте экземпляр класса Stack с auto_reverse=True.

   (h) Добавьте элементы: 1, 2, 3 (в стеке будет [3, 2, 1], где 3 — верх).

   (i) Выведите размер стека и результат peek (должен быть 3).

   (j) Вызовите pop, выведите результат (должен быть 3).

   (k) Повторите вывод размера и peek (теперь верх — 2).

   Пример использования:

```
stack = Stack(auto_reverse=True)
stack.push(1)
stack.push(2)
stack.push(3)  # стек: [3,2,1], верх - 3

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 3
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 2
```

7. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр case_sensitive=True. Используется только если элементы — строки.

(b) Создайте метод push, который принимает элемент. Если элемент — строка и case_sensitive=False, то перед добавлением преобразует её в нижний регистр. Добавляет элемент в стек.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает пустую строку .

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает пустую строку .

(g) Создайте экземпляр класса Stack с case_sensitive=False.

(h) Добавьте строки: "Hello "WORLD "Python".

(i) Выведите размер стека и верхний элемент (должен быть "python").

(j) Вызовите pop, выведите результат.

(k) Повторите вывод размера и верхнего элемента.

Пример использования:

```
stack = Stack(case_sensitive=False)
stack.push("Hello")
stack.push("WORLD")
stack.push("Python")

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())  # "python"

popped = stack.pop()
print("Вытолкнут:", popped)  # "python"
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # "world"
```

8. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр min_value=None. Если задан, то при добавлении элемента проверяется, что он >= min_value.

(b) Создайте метод push, который принимает элемент. Если min_value задан и элемент < min_value, элемент не добавляется, а метод возвращает False. Иначе — добавляет и возвращает True.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с min_value=10.

(h) Добавьте элементы: 5 (не добавится), 15, 20, 8 (не добавится), 25.

(i) Выведите размер стека и верхний элемент.

(j) Вызовите pop, выведите результат.

(k) Повторите вывод размера и верхнего элемента.

Пример использования:

```
stack = Stack(min_value=10)
print(stack.push(5))   # False
print(stack.push(15))  # True
print(stack.push(20))  # True
print(stack.push(8))   # False
print(stack.push(25))  # True

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 25
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 20
```

9. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

   Инструкции:

   (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр max_increments=0 — максимальное количество добавлений. Если 0 — без ограничений.

   (b) Создайте метод push, который принимает элемент. Если max_increments $>$ 0 и количество вызовов push превысило max_increments, элемент не добавляется, метод возвращает False. Иначе — добавляет и возвращает True.

   (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает строку — ".

   (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

   (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает строку — "".

(g) Создайте экземпляр класса Stack с max_increments=3.

(h) Добавьте элементы: 100, 200, 300, 400 (последний не добавится).

(i) Выведите размер стека и верхний элемент.

(j) Вызовите pop, выведите результат.

(k) Повторите вывод размера и верхнего элемента.

Пример использования:

```
stack = Stack(max_increments=3)
print(stack.push(100))  # True
print(stack.push(200))  # True
print(stack.push(300))  # True
print(stack.push(400))  # False

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 300
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 200
```

10. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр validate_type=None. Если задан (например, int), то при добавлении проверяется, что элемент является экземпляром этого типа.

    (b) Создайте метод push, который принимает элемент. Если validate_type задан и элемент не является его экземпляром, элемент не добавляется, метод возвращает False. Иначе — добавляет и возвращает True.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с validate_type=int.

    (h) Добавьте элементы: 10, "20"(не добавится), 30, 40.5 (не добавится), 50.

    (i) Выведите размер стека и верхний элемент.

(j) Вызовите pop, выведите результат.

(k) Повторите вывод размера и верхнего элемента.

Пример использования:

```
stack = Stack(validate_type=int)
print(stack.push(10))    # True
print(stack.push("20"))  # False
print(stack.push(30))    # True
print(stack.push(40.5))  # False
print(stack.push(50))    # True

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 50
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 30
```

11. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр unique_per_session=False. Если True, то не позволяет добавлять один и тот же элемент дважды за всё время жизни стека (даже если он был удален).

    (b) Создайте метод push, который принимает элемент. Если unique_per_session=True и элемент уже когда-либо был добавлен (даже если потом удален), он не добавляется, метод возвращает False. Иначе — добавляет и возвращает True.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с unique_per_session=True.

    (h) Добавьте элементы: 7, 14, 7 (не добавится), 21, 14 (не добавится).

    (i) Выведите размер стека и верхний элемент.

    (j) Вызовите pop, выведите результат.

    (k) Попробуйте добавить 21 снова (не должно добавиться).

    (l) Выведите размер стека.

    Пример использования:

```
stack = Stack(unique_per_session=True)
print(stack.push(7))   # True
print(stack.push(14))  # True
print(stack.push(7))   # False
print(stack.push(21))  # True
print(stack.push(14))  # False

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 21

print(stack.push(21))  # False (уже был)
print("Размер стека:", stack.size())  # по-прежнему 2
```

12. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_limit_per_call=1 (по умолчанию). Если >1, то метод push может принимать несколько элементов (через *args) и добавлять их все за один вызов (но не более push_limit_per_call элементов за вызов).

    (b) Создайте метод push, который принимает один или несколько элементов (если push_limit_per_call > 1). Если передано больше элементов, чем push_limit_per_call, добавляются только первые push_limit_per_call элементов, остальные игнорируются. Возвращает количество реально добавленных элементов.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_limit_per_call=3.

    (h) Вызовите push с элементами 1, 2, 3, 4, 5 — добавятся только 1,2,3.

    (i) Вызовите push с элементами 6, 7 — добавятся оба.

    (j) Выведите размер стека и верхний элемент.

    (k) Вызовите pop, выведите результат.

    (l) Повторите вывод размера и верхнего элемента.

    Пример использования:

```
stack = Stack(push_limit_per_call=3)
added = stack.push(1, 2, 3, 4, 5)  # добавит 1,2,3; вернет 3
print("Добавлено:", added)

added = stack.push(6, 7)  # добавит 6,7; вернет 2
print("Добавлено:", added)

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 7
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 6
```

13. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр pop_multiple=False. Если True, то метод pop может принимать необязательный аргумент count (по умолчанию 1) и возвращать список из count верхних элементов.

    (b) Создайте метод push, который принимает один элемент и добавляет его в стек. Возвращает None.

    (c) Создайте метод pop, который, если pop_multiple=False, выталкивает один верхний элемент и возвращает его. Если pop_multiple=True, принимает count (по умолчанию 1) и возвращает список из count верхних элементов (если запрошено больше, чем есть, возвращает все). Если стек пуст, возвращает пустой список [] (в режиме pop_multiple) или None (в обычном режиме).

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None. Не поддерживает множественный просмотр.

    (g) Создайте экземпляр класса Stack с pop_multiple=True.

    (h) Добавьте элементы: 10, 20, 30, 40, 50.

    (i) Выведите размер стека и верхний элемент.

    (j) Вызовите pop с count=3, выведите результат (должен быть [50,40,30]).

    (k) Выведите размер стека и верхний элемент (теперь 20).

    Пример использования:

```
stack = Stack(pop_multiple=True)
stack.push(10)
stack.push(20)
stack.push(30)
stack.push(40)
stack.push(50)

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop(count=3)
print("Вытолкнуты:", popped)  # [50, 40, 30]

print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 20
```

14. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр on_push_callback=None — функция, которая будет вызываться после каждого успешного добавления элемента (с аргументом — добавленным элементом).

    (b) Создайте метод push, который принимает элемент и добавляет его в стек. Если on_push_callback не None, вызывает её с добавленным элементом. Возвращает добавленный элемент.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте функцию logger(x): print(f"[LOG] Добавлен: x")

    (h) Создайте экземпляр класса Stack, передав logger в on_push_callback.

    (i) Добавьте элементы: 101, 202, 303 (при каждом добавлении должно выводиться сообщение).

    (j) Выведите размер стека и верхний элемент.

    (k) Вызовите pop, выведите результат.

    (l) Повторите вывод размера и верхнего элемента.

    Пример использования:

```
def logger(x):
    print(f"[LOG] Добавлен: {x}")

stack = Stack(on_push_callback=logger)
stack.push(101)  # выведет [LOG] Добавлен: 101
stack.push(202)  # выведет [LOG] Добавлен: 202
stack.push(303)  # выведет [LOG] Добавлен: 303

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())

popped = stack.pop()
print("Вытолкнут:", popped)  # 303
print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # 202
```

15. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр compress_on_push=False. Если True, то при добавлении элемента, равного текущему верхнему, вместо добавления нового элемента увеличивается счетчик дубликатов у верхнего элемента (стек хранит пары (элемент, счетчик)).

    (b) Создайте метод push, который принимает элемент. Если compress_on_push=True и элемент равен текущему верхнему, увеличивает счетчик верхнего элемента. Иначе — добавляет новый элемент (со счетчиком 1, если режим сжатия включен).

    (c) Создайте метод pop, который выталкивает верхний элемент. Если режим сжатия включен и счетчик >1, уменьшает счетчик и возвращает элемент. Если счетчик=1, удаляет элемент. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает общее количество элементов (с учетом счетчиков, если режим сжатия включен).

    (f) Создайте метод peek, который возвращает верхний элемент (не счетчик, а само значение), если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с compress_on_push=True.

    (h) Добавьте элементы: 5, 5, 5, 10, 10, 15.

    (i) Выведите размер стека (должен быть 6) и верхний элемент (15).

    (j) Вызовите pop, выведите результат (15).

    (k) Вызовите pop, выведите результат (10) — счетчик у 10 должен уменьшиться с 2 до 1.

    (l) Выведите размер стека (должен быть 4).

Пример использования:

```
stack = Stack(compress_on_push=True)
stack.push(5)
stack.push(5)
stack.push(5)
stack.push(10)
stack.push(10)
stack.push(15)

print("Размер стека:", stack.size())     # 6
print("Верхний элемент:", stack.peek())   # 15

popped = stack.pop()
print("Вытолкнут:", popped)  # 15

popped = stack.pop()
print("Вытолкнут:", popped)  # 10

print("Размер после двух pop:", stack.size())  # 4
```

16. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр immutable_pop=False. Если True, то метод pop не удаляет элемент из стека, а только возвращает его (поведение как peek, но называется pop).

    (b) Создайте метод push, который принимает элемент и добавляет его в стек.

    (c) Создайте метод pop, который, если immutable_pop=False, выталкивает верхний элемент и возвращает его. Если immutable_pop=True, возвращает верхний элемент, не удаляя его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None. (Поведение не зависит от immutable_pop.)

    (g) Создайте экземпляр класса Stack с immutable_pop=True.

    (h) Добавьте элементы: 1, 3, 5, 7.

    (i) Выведите размер стека и результат pop (должен быть 7, но стек не изменится).

    (j) Снова вызовите pop, снова выведите результат (опять 7).

    (k) Выведите размер стека (по-прежнему 4).

    Пример использования:

```
stack = Stack(immutable_pop=True)
stack.push(1)
stack.push(3)
stack.push(5)
stack.push(7)

print("Размер стека:", stack.size())
print("Первый pop:", stack.pop())  # 7
print("Второй pop:", stack.pop())  # 7 (стек не изменился)
print("Размер стека:", stack.size())  # 4
```

17. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр track_history=False. Если True, то сохраняет историю всех когда-либо находившихся в стеке элементов (даже удаленных) в отдельном списке.

    (b) Создайте метод push, который принимает элемент, добавляет его в стек, и если track_history=True, добавляет его и в историю.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте метод get_history (только если track_history=True), который возвращает копию списка истории.

    (h) Создайте экземпляр класса Stack с track_history=True.

    (i) Добавьте элементы: 2, 4, 6.

    (j) Вызовите pop (вернет 6).

    (k) Добавьте 8.

    (l) Выведите текущий стек (через peek и size) и историю (должна быть [2,4,6,8]).

    Пример использования:

```
stack = Stack(track_history=True)
stack.push(2)
stack.push(4)
stack.push(6)
stack.pop()  # 6
stack.push(8)

print("Текущий размер:", stack.size())      # 2
print("Верхний элемент:", stack.peek())     # 8
print("История:", stack.get_history())      # [2,4,6,8]
```

18. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_only_even=False. Если True, то добавляются только четные числа (остальные игнорируются).

    (b) Создайте метод push, который принимает элемент. Если push_only_even=True и элемент не является четным целым числом, он не добавляется. Иначе — добавляется.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_only_even=True.

    (h) Добавьте элементы: 1 (игнорируется), 2, 3 (игнорируется), 4, 5 (игнорируется), 6.

    (i) Выведите размер стека (должен быть 3) и верхний элемент (6).

    (j) Вызовите pop, выведите результат (6).

    (k) Повторите вывод размера и верхнего элемента (теперь 4).

    Пример использования:

```
stack = Stack(push_only_even=True)
stack.push(1)  # игнорируется
stack.push(2)
stack.push(3)  # игнорируется
stack.push(4)
stack.push(5)  # игнорируется
stack.push(6)

print("Размер стека:", stack.size())     # 3
print("Верхний элемент:", stack.peek())   # 6

popped = stack.pop()
print("Вытолкнут:", popped)  # 6

print("Размер после pop:", stack.size())    # 2
print("Верхний элемент:", stack.peek())     # 4
```

19. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна

создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр reverse_pop=False. Если True, то метод pop возвращает не верхний, а нижний элемент стека (и удаляет его).

(b) Создайте метод push, который принимает элемент и добавляет его в стек (наверх).

(c) Создайте метод pop, который, если reverse_pop=False, выталкивает верхний элемент и возвращает его. Если reverse_pop=True, выталкивает нижний элемент и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None. (Не зависит от reverse_pop.)

(g) Создайте экземпляр класса Stack с reverse_pop=True.

(h) Добавьте элементы: 10, 20, 30 (в стеке: [10,20,30], верх — 30).

(i) Выведите результат peek (должен быть 30).

(j) Вызовите pop — должен вернуться 10 (нижний), стек станет [20,30].

(k) Выведите размер и снова peek (должен быть 30).

Пример использования:

```
stack = Stack(reverse_pop=True)
stack.push(10)
stack.push(20)
stack.push(30)

print("Верхний элемент (peek):", stack.peek())  # 30
popped = stack.pop()
print("Вытолкнут (нижний):", popped)            # 10
print("Размер после pop:", stack.size())        # 2
print("Верхний элемент (peek):", stack.peek())  # 30
```

20. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_with_timestamp=False. Если True, то при добавлении элемент сохраняется вместе с текущим временем (в формате Unix timestamp).

(b) Создайте метод push, который принимает элемент. Если push_with_timestamp=True, сохраняет пару (элемент, time.time()). Иначе — только элемент.

(c) Создайте метод pop, который выталкивает верхний элемент. Если режим с временем включен, возвращает пару (элемент, timestamp). Иначе — только элемент. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент (или пару, если включен режим времени), если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_with_timestamp=True.

(h) Добавьте элементы: "first "second "third".

(i) Выведите размер стека и результат peek (должна быть пара ("third timestamp)).

(j) Вызовите pop, выведите результат (тоже пара).

(k) Повторите вывод размера и peek.

Пример использования:

```
import time

stack = Stack(push_with_timestamp=True)
stack.push("first")
stack.push("second")
stack.push("third")

print("Размер стека:", stack.size())
peek_result = stack.peek()
print("Верхний элемент и время:", peek_result)  # ('third',
    1712345678.123456)

popped = stack.pop()
print("Вытолкнут:", popped)  # ('third', 1712345678.123456)

print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # ('second', ...)
```

21. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_pairs=False. Если True, то метод push ожидает два аргумента (key, value) и сохраняет их как кортеж. Если False — один аргумент.

(b) Создайте метод push, который, если push_pairs=False, принимает один элемент. Если push_pairs=True, принимает два аргумента (key, value) и сохраняет (key, value). Возвращает сохраненный элемент (или кортеж).

(c) Создайте метод pop, который выталкивает верхний элемент (или кортеж) и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент (или кортеж), если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_pairs=True.

(h) Добавьте пары: ("a 1), ("b 2), ("c 3).

(i) Выведите размер стека и результат peek (должен быть ("c 3)).

(j) Вызовите pop, выведите результат.

(k) Повторите вывод размера и peek.

Пример использования:

```
stack = Stack(push_pairs=True)
stack.push("a", 1)
stack.push("b", 2)
stack.push("c", 3)

print("Размер стека:", stack.size())
print("Верхний элемент:", stack.peek())  # ('c', 3)

popped = stack.pop()
print("Вытолкнут:", popped)  # ('c', 3)

print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # ('b', 2)
```

22. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр auto_dedup=False. Если True, то при добавлении элемента, который уже есть в стеке (не обязательно на вершине), сначала удаляет все его предыдущие вхождения.

(b) Создайте метод push, который принимает элемент. Если auto_dedup=True и такой элемент уже есть в стеке, удаляет все его вхождения, затем добавляет новый элемент. Иначе — просто добавляет.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с auto_dedup=True.

(h) Добавьте элементы: 1, 2, 1, 3, 2, 4.

(i) После каждого добавления выводите содержимое стека (реализуйте вспомогательный метод _debug_list, возвращающий список элементов снизу вверх — только для отладки, не включайте в задание студентам; в решении можно использовать stack._items, если инкапсуляция не строгая).

(j) Выведите итоговый размер и верхний элемент.

Пример использования (с отладочным выводом для ясности):

```
# (В решении студент не обязан реализовывать _debug_list, но для проверки мож
    но временно добавить)
stack = Stack(auto_dedup=True)
stack.push(1)  # стек: [1]
stack.push(2)  # стек: [1,2]
stack.push(1)  # удаляет старую 1, добавляет новую -> [2,1]
stack.push(3)  # [2,1,3]
stack.push(2)  # удаляет 2, добавляет новую -> [1,3,2]
stack.push(4)  # [1,3,2,4]

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 4
```

23. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_if_max=False. Если True, то элемент добавляется только если он больше всех текущих элементов в стеке.

(b) Создайте метод push, который принимает элемент. Если push_if_max=True и элемент не является строго больше всех элементов в стеке, он не добавляется. Иначе — добавляется.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_if_max=True.

(h) Добавьте элементы: 5, 3 (не добавится, т.к. 3<5), 10, 7 (не добавится, т.к. 7<10), 15.

(i) Выведите размер стека (должен быть 3) и верхний элемент (15).

(j) Вызовите pop, выведите результат (15).

(k) Повторите вывод размера и верхнего элемента (теперь 10).

Пример использования:

```
stack = Stack(push_if_max=True)
stack.push(5)
stack.push(3)   # не добавится
stack.push(10)
stack.push(7)   # не добавится
stack.push(15)

print("Размер стека:", stack.size())     # 3
print("Верхний элемент:", stack.peek())   # 15

popped = stack.pop()
print("Вытолкнут:", popped)  # 15

print("Размер после pop:", stack.size())    # 2
print("Верхний элемент:", stack.peek())     # 10
```

24. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр cumulative=False. Если True, то при добавлении элемента он суммируется с предыдущим верхним элементом (первый элемент добавляется как есть).

    (b) Создайте метод push, который принимает элемент. Если cumulative=True и стек не пуст, то добавляемый элемент становится element + текущий_верх. Затем этот результат добавляется в стек. Если стек пуст, добавляется element как есть.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с cumulative=True.

    (h) Добавьте элементы: 1, 2, 3, 4.

    (i) Выведите содержимое стека после каждого добавления (для проверки: после 1 → [1]; после 2 → [1,3]; после 3 → [1,3,6]; после 4 → [1,3,6,10]).

    (j) Выведите итоговый размер и верхний элемент (10).

    (k) Вызовите pop, выведите результат (10).

    (l) Повторите вывод размера и верхнего элемента (теперь 6).

Пример использования:

```
stack = Stack(cumulative=True)
stack.push(1)  # [1]
stack.push(2)  # [1, 1+2=3]
stack.push(3)  # [1,3, 3+3=6]
stack.push(4)  # [1,3,6, 6+4=10]

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 10

popped = stack.pop()
print("Вытолкнут:", popped)  # 10

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 6
```

25. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_squared=False. Если True, то при добавлении элемент возводится в квадрат перед добавлением.

    (b) Создайте метод push, который принимает элемент. Если push_squared=True, добавляет element**2. Иначе — element.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_squared=True.

    (h) Добавьте элементы: 2, 3, 4, 5.

    (i) Выведите размер стека и верхний элемент (должен быть 25).

    (j) Вызовите pop, выведите результат (25).

    (k) Повторите вывод размера и верхнего элемента (теперь 16).

    Пример использования:

```
stack = Stack(push_squared=True)
stack.push(2)  # добавит 4
stack.push(3)  # добавит 9
stack.push(4)  # добавит 16
stack.push(5)  # добавит 25

print("Размер стека:", stack.size())     # 4
```

```
print("Верхний элемент:", stack.peek())   # 25

popped = stack.pop()
print("Вытолкнут:", popped)  # 25

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 16
```

26. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_absolute=False. Если True, то при добавлении сохраняется абсолютное значение элемента (abs(element)).

    (b) Создайте метод push, который принимает элемент. Если push_absolute=True, добавляет abs(element). Иначе — element.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_absolute=True.

    (h) Добавьте элементы: -5, 3, -8, 2.

    (i) Выведите размер стека и верхний элемент (должен быть 2).

    (j) Вызовите pop, выведите результат (2).

    (k) Повторите вывод размера и верхнего элемента (теперь 8).

    Пример использования:

```
stack = Stack(push_absolute=True)
stack.push(-5)  # добавит 5
stack.push(3)   # добавит 3
stack.push(-8)  # добавит 8
stack.push(2)   # добавит 2

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 2

popped = stack.pop()
print("Вытолкнут:", popped)  # 2

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 8
```

27. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_rounded=False. Если True, то при добавлении элемент округляется до целого числа (round(element)).

    (b) Создайте метод push, который принимает элемент. Если push_rounded=True, добавляет round(element). Иначе — element.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_rounded=True.

    (h) Добавьте элементы: 3.2, 4.7, 5.1, 6.9.

    (i) Выведите размер стека и верхний элемент (должен быть 7).

    (j) Вызовите pop, выведите результат (7).

    (k) Повторите вывод размера и верхнего элемента (теперь 5).

    Пример использования:

```
stack = Stack(push_rounded=True)
stack.push(3.2)  # 3
stack.push(4.7)  # 5
stack.push(5.1)  # 5
stack.push(6.9)  # 7

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 7

popped = stack.pop()
print("Вытолкнут:", popped)  # 7

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 5
```

28. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_negated=False. Если True, то при добавлении элемент сохраняется с обратным знаком (-element).

(b) Создайте метод push, который принимает элемент. Если push_negated=True, добавляет -element. Иначе — element.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_negated=True.

(h) Добавьте элементы: 10, 20, 30, 40.

(i) Выведите размер стека и верхний элемент (должен быть -40).

(j) Вызовите pop, выведите результат (-40).

(k) Повторите вывод размера и верхнего элемента (теперь -30).

Пример использования:

```
stack = Stack(push_negated=True)
stack.push(10)  # -10
stack.push(20)  # -20
stack.push(30)  # -30
stack.push(40)  # -40

print("Размер стека:", stack.size())      # 4
print("Верхний элемент:", stack.peek())   # -40

popped = stack.pop()
print("Вытолкнут:", popped)  # -40

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # -30
```

29. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_doubled=False. Если True, то при добавлении элемент умножается на 2.

(b) Создайте метод push, который принимает элемент. Если push_doubled=True, добавляет element * 2. Иначе — element.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_doubled=True.

(h) Добавьте элементы: 1, 2, 3, 4.

(i) Выведите размер стека и верхний элемент (должен быть 8).

(j) Вызовите pop, выведите результат (8).

(k) Повторите вывод размера и верхнего элемента (теперь 6).

Пример использования:

```
stack = Stack(push_doubled=True)
stack.push(1)  # 2
stack.push(2)  # 4
stack.push(3)  # 6
stack.push(4)  # 8

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 8

popped = stack.pop()
print("Вытолкнут:", popped)  # 8

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 6
```

30. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_halved=False. Если True, то при добавлении элемент делится на 2.0.

(b) Создайте метод push, который принимает элемент. Если push_halved=True, добавляет element / 2.0. Иначе — element.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_halved=True.

(h) Добавьте элементы: 4, 8, 12, 16.

(i) Выведите размер стека и верхний элемент (должен быть 8.0).

(j) Вызовите pop, выведите результат (8.0).

(k) Повторите вывод размера и верхнего элемента (теперь 6.0).

Пример использования:

```
stack = Stack(push_halved=True)
stack.push(4)   # 2.0
stack.push(8)   # 4.0
stack.push(12)  # 6.0
stack.push(16)  # 8.0

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 8.0

popped = stack.pop()
print("Вытолкнут:", popped)  # 8.0

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # 6.0
```

31. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

Инструкции:

(a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_as_string=False. Если True, то при добавлении элемент преобразуется в строку str(element).

(b) Создайте метод push, который принимает элемент. Если push_as_string=True, добавляет str(element). Иначе — element.

(c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

(d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

(e) Создайте метод size, который возвращает текущее количество элементов в стеке.

(f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

(g) Создайте экземпляр класса Stack с push_as_string=True.

(h) Добавьте элементы: 100, 200, 300, 400.

(i) Выведите размер стека и верхний элемент (должен быть "400").

(j) Вызовите pop, выведите результат ("400").

(k) Повторите вывод размера и верхнего элемента (теперь "300").

Пример использования:

```
stack = Stack(push_as_string=True)
stack.push(100)  # "100"
stack.push(200)  # "200"
stack.push(300)  # "300"
stack.push(400)  # "400"

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # "400"

popped = stack.pop()
print("Вытолкнут:", popped)  # "400"

print("Размер после pop:", stack.size())    # 3
print("Верхний элемент:", stack.peek())     # "300"
```

32. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_with_index=False. Если True, то при добавлении сохраняется кортеж (элемент, порядковый_номер_добавления).

    (b) Создайте метод push, который принимает элемент. Если push_with_index=True, добавляет (element, self._counter), где _counter — внутренний счетчик, увеличивающийся при каждом добавлении. Иначе — element.

    (c) Создайте метод pop, который выталкивает верхний элемент (или кортеж) и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент (или кортеж), если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_with_index=True.

    (h) Добавьте элементы: "alpha "beta "gamma".

    (i) Выведите размер стека и результат peek (должен быть ("gamma 2) — если считать с 0).

    (j) Вызовите pop, выведите результат.

    (k) Повторите вывод размера и peek.

    Пример использования:

```
stack = Stack(push_with_index=True)
stack.push("alpha")  # ("alpha", 0)
stack.push("beta")   # ("beta", 1)
stack.push("gamma")  # ("gamma", 2)

print("Размер стека:", stack.size())
```

```
print("Верхний элемент:", stack.peek())  # ('gamma', 2)

popped = stack.pop()
print("Вытолкнут:", popped)  # ('gamma', 2)

print("Размер после pop:", stack.size())
print("Верхний элемент:", stack.peek())  # ('beta', 1)
```

33. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_unique_top=False. Если True, то при добавлении, если элемент равен текущему верхнему, он не добавляется.

    (b) Создайте метод push, который принимает элемент. Если push_unique_top=True и стек не пуст и element == текущий_верх, то элемент не добавляется. Иначе — добавляется.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_unique_top=True.

    (h) Добавьте элементы: 1, 2, 2, 3, 3, 3, 4.

    (i) Выведите размер стека (должен быть 4) и верхний элемент (4).

    (j) Вызовите pop, выведите результат (4).

    (k) Повторите вывод размера и верхнего элемента (теперь 3).

    Пример использования:

```
stack = Stack(push_unique_top=True)
stack.push(1)
stack.push(2)
stack.push(2)  # не добавится
stack.push(3)
stack.push(3)  # не добавится
stack.push(3)  # не добавится
stack.push(4)

print("Размер стека:", stack.size())     # 4
print("Верхний элемент:", stack.peek())   # 4

popped = stack.pop()
print("Вытолкнут:", popped)  # 4
```

```
print("Размер после pop:", stack.size())     # 3
print("Верхний элемент:", stack.peek())      # 3
```

34. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_even_only=False. Если True, то добавляются только четные числа.

    (b) Создайте метод push, который принимает элемент. Если push_even_only=True и element % 2 != 0, элемент не добавляется. Иначе — добавляется.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_even_only=True.

    (h) Добавьте элементы: 1 (не добавится), 2, 3 (не добавится), 4, 5 (не добавится), 6.

    (i) Выведите размер стека (должен быть 3) и верхний элемент (6).

    (j) Вызовите pop, выведите результат (6).

    (k) Повторите вывод размера и верхнего элемента (теперь 4).

    Пример использования:

```
stack = Stack(push_even_only=True)
stack.push(1)  # нет
stack.push(2)
stack.push(3)  # нет
stack.push(4)
stack.push(5)  # нет
stack.push(6)

print("Размер стека:", stack.size())     # 3
print("Верхний элемент:", stack.peek())   # 6

popped = stack.pop()
print("Вытолкнут:", popped)  # 6

print("Размер после pop:", stack.size())     # 2
print("Верхний элемент:", stack.peek())      # 4
```

35. Написать программу на Python, которая создает класс Stack для представления стека с инкапсуляцией. Класс должен содержать методы push, pop, is_empty, size и peek, которые реализуют операции вталкивания, выталкивания, проверки пустоты, получения размера и просмотра вершины стека соответственно. Программа также должна создавать экземпляр класса Stack, вталкивать в него элементы, выталкивать элементы и выводить информацию о стеке на экран.

    Инструкции:

    (a) Создайте класс Stack с методом __init__, который инициализирует пустой стек. Принимает параметр push_odd_only=False. Если True, то добавляются только нечетные числа.

    (b) Создайте метод push, который принимает элемент. Если push_odd_only=True и element % 2 == 0, элемент не добавляется. Иначе — добавляется.

    (c) Создайте метод pop, который выталкивает верхний элемент из стека и возвращает его. Если стек пуст, возвращает None.

    (d) Создайте метод is_empty, который возвращает True, если стек пуст, и False в противном случае.

    (e) Создайте метод size, который возвращает текущее количество элементов в стеке.

    (f) Создайте метод peek, который возвращает верхний элемент стека, если стек не пуст. Если стек пуст, возвращает None.

    (g) Создайте экземпляр класса Stack с push_odd_only=True.

    (h) Добавьте элементы: 2 (не добавится), 1, 4 (не добавится), 3, 6 (не добавится), 5.

    (i) Выведите размер стека (должен быть 3) и верхний элемент (5).

    (j) Вызовите pop, выведите результат (5).

    (k) Повторите вывод размера и верхнего элемента (теперь 3).

    Пример использования:

```
stack = Stack(push_odd_only=True)
stack.push(2)  # нет
stack.push(1)
stack.push(4)  # нет
stack.push(3)
stack.push(6)  # нет
stack.push(5)

print("Размер стека:", stack.size())      # 3
print("Верхний элемент:", stack.peek())   # 5

popped = stack.pop()
print("Вытолкнут:", popped)  # 5

print("Размер после pop:", stack.size())     # 2
print("Верхний элемент:", stack.peek())      # 3
```

2.3.3 Задача 3 (двусвязный список)

1. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией внутренней структуры. Класс должен

содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

Инструкции:

(a) Создайте класс Node с методом __init__, который принимает данные data и сохраняет их в атрибуте self._data. Также инициализирует self._next и self._prev как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head и self._tail как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит все элементы списка через пробел, двигаясь от головы к хвосту. Если список пуст — выводит "Список пуст".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет новый узел **в конец списка**. Обновляет self._tail и ссылки prev/next.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое** вхождение узла с этим значением. Корректно обновляет соседние ссылки и self._head/self._tail при необходимости.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы со значениями 10, 20, 30, 40.

(h) Вызовите display и выведите результат.

(i) Вставьте узел со значением 50.

(j) Снова вызовите display.

(k) Удалите узел со значением 20.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(10)
dll.insert(20)
dll.insert(30)
dll.insert(40)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(50)
print("After inserting 50:")
dll.display()

dll.delete(20)
print("After deleting 20:")
dll.display()
```

2. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

(a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._value. Инициализирует self._next и self._previous как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first и self._last как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы списка от первого к последнему, разделенные запятыми. Если список пуст — выводит "Нет элементов".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает элемент и вставляет его **в начало списка**. Обновляет self._first и ссылки.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее** вхождение узла с этим значением. Корректно обновляет связи и границы списка.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 5, 15, 25, 15.

(h) Вызовите display.

(i) Вставьте узел 35 в начало.

(j) Снова вызовите display.

(k) Удалите последнее вхождение 15.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(5)
dll.insert(15)
dll.insert(25)
dll.insert(15)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(35)
print("After inserting 35 at start:")
dll.display()

dll.delete(15)
print("After deleting last occurrence of 15:")
dll.display()
```

3. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает content и сохраняет его в self._payload. Инициализирует self._forward и self._backward как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._root и self._end как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в формате "[элемент1] <-> [элемент2] <-> ...". Если пуст — "Пусто".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **после первого узла** (если список не пуст; если пуст — вставляет как первый).

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения** этого значения. Обновляет ссылки и границы.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 100, 200, 300.

(h) Вызовите display.

(i) Вставьте 150 после первого узла.

(j) Снова вызовите display.

(k) Удалите все вхождения 150.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(100)
dll.insert(200)
dll.insert(300)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(150)
print("After inserting 150 after first:")
dll.display()

dll.delete(150)
print("After deleting all 150s:")
dll.display()
```

4. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает entry и сохраняет его в self._item. Инициализирует self._succ и self._pred как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._top и self._bottom как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в обратном порядке (от хвоста к голове), разделенные ". Если пуст — "Обратный просмотр: пусто".

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **перед последним узлом** (если узлов >1; если 0 или 1 — вставляет в конец).

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет первый найденный узел. Если узел — единственный, обнуляет self._top и self._bottom.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 7, 14, 21.

(h) Вызовите display.

(i) Вставьте 18 перед последним узлом.

(j) Снова вызовите display.

(k) Удалите узел со значением 14.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(7)
dll.insert(14)
dll.insert(21)

print("Initial Doubly Linked List (reversed):")
dll.display()

dll.insert(18)
print("After inserting 18 before last:")
dll.display()

dll.delete(14)
print("After deleting 14:")
dll.display()
```

5. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._key. Инициализирует self._link_next и self._link_prev как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._header и self._trailer как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в квадратных скобках через запятую: [a, b, c]. Если пуст — [].

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если такого значения еще нет в списке**. Вставляет в конец.

   (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет узел, если он существует. Если не существует — ничего не делает.

   (f) Создайте экземпляр класса DoublyLinkedList.

   (g) Вставьте узлы: 3, 6, 9, 6 (второй 6 не вставится).

   (h) Вызовите display.

(i) Вставьте 12.

(j) Снова вызовите display.

(k) Удалите 6.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(3)
dll.insert(6)
dll.insert(9)
dll.insert(6)  # игнорируется

print("Initial Doubly Linked List:")
dll.display()

dll.insert(12)
print("After inserting 12:")
dll.display()

dll.delete(6)
print("After deleting 6:")
dll.display()
```

6. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает data_point и сохраняет его в self._datum. Инициализирует self._next_node и self._prev_node как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._start и self._finish как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в формате "Элементы: val1 -> val2 -> val3 двигаясь от начала к концу. Если пуст — "Элементы: (нет)".

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно больше последнего элемента** (если список не пуст). Если пуст — вставляет. Иначе — игнорирует.

   (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первый узел**, если он равен значению. Не ищет дальше.

   (f) Создайте экземпляр класса DoublyLinkedList.

   (g) Вставьте узлы: 1, 5, 3 (игнорируется), 10.

   (h) Вызовите display.

   (i) Вставьте 7 (игнорируется, т.к. $7 < 10$).

   (j) Снова вызовите display.

   (k) Удалите 5.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(1)
dll.insert(5)
dll.insert(3)  # игнорируется
dll.insert(10)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(7)  # игнорируется
print("After attempting to insert 7:")
dll.display()

dll.delete(5)
print("After deleting 5:")
dll.display()
```

7. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает item_value и сохраняет его в self._content. Инициализирует self._ptr_next и self._ptr_prev как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_node и self._tail_node как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде строки, разделенной точками: "a.b.c". Если пуст — "пусто".

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **в середину списка** (если четное количество — после левой средней позиции; если нечетное — в центр). Если список пуст — вставляет как первый.

   (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все узлы с этим значением**.

   (f) Создайте экземпляр класса DoublyLinkedList.

   (g) Вставьте узлы: 10, 20, 30.

   (h) Вызовите display.

   (i) Вставьте 25 в середину (между 20 и 30).

   (j) Снова вызовите display.

   (k) Удалите все вхождения 25.

   (l) Снова вызовите display.

   Пример использования:

```
dll = DoublyLinkedList()
dll.insert(10)
dll.insert(20)
dll.insert(30)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(25)
print("After inserting 25 in middle:")
dll.display()

dll.delete(25)
print("After deleting 25:")
dll.display()
```

8. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает node_data и сохраняет его в self._info. Инициализирует self._nxt и self._prv как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._front и self._rear как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в формате "Front->Back: [значения]"и "Back->Front: [значения в обратном порядке]". Если пуст — "Список пуст в обоих направлениях".

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **в начало, только если значение четное**. Если нечетное — вставляет в конец.

   (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

   (f) Создайте экземпляр класса DoublyLinkedList.

   (g) Вставьте узлы: 4 (в начало), 7 (в конец), 6 (в начало), 9 (в конец).

   (h) Вызовите display.

   (i) Вставьте 8 (в начало).

   (j) Снова вызовите display.

   (k) Удалите 7.

   (l) Снова вызовите display.

   Пример использования:

```
dll = DoublyLinkedList()
dll.insert(4)
dll.insert(7)
dll.insert(6)
dll.insert(9)

print("Initial Doubly Linked List:")
```

```
dll.display()

dll.insert(8)
print("After inserting 8:")
dll.display()

dll.delete(7)
print("After deleting 7:")
dll.display()
```

9. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

   Инструкции:

   (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._element. Инициализирует self._next_elem и self._prev_elem как None.

   (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_elem и self._tail_elem как None.

   (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "HEAD <-> val1 <-> val2 <-> ... <-> TAIL". Если пуст — "HEAD <-> TAIL (пусто)".

   (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **после узла с наименьшим значением** (если несколько — после первого). Если список пуст — вставляет как единственный.

   (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

   (f) Создайте экземпляр класса DoublyLinkedList.

   (g) Вставьте узлы: 50, 30, 40.

   (h) Вызовите display.

   (i) Вставьте 35 (после 30 — минимального).

   (j) Снова вызовите display.

   (k) Удалите последнее вхождение 40.

   (l) Снова вызовите display.

   Пример использования:

```
dll = DoublyLinkedList()
dll.insert(50)
dll.insert(30)
dll.insert(40)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(35)
print("After inserting 35 after min:")
dll.display()

dll.delete(40)
print("After deleting last occurrence of 40:")
dll.display()
```

10. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._val. Инициализирует self._link_f и self._link_b как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_item и self._last_item как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Элементы (прямой порядок): ... а затем "Элементы (обратный порядок): ...". Если пуст — "Нет данных".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **перед узлом с наибольшим значением** (если несколько — перед первым). Если список пуст — вставляет как единственный.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 5, 15, 10.

    (h) Вызовите display.

    (i) Вставьте 12 (перед 15 — максимальным).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 10.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(5)
dll.insert(15)
dll.insert(10)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(12)
print("After inserting 12 before max:")
dll.display()

dll.delete(10)
print("After deleting all 10s:")
dll.display()
```

11. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

(a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._data_field. Инициализирует self._next_ref и self._prev_ref как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._entry_point и self._exit_point как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в одну строку, разделенные -> и в конце добавляет -> None". Если пуст — "None".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **в позицию, равную значению по модулю длины списка** (если список не пуст; если пуст — вставляет как первый). Например, при длине 3 и значении 7: $7 \% 3 = 1 \rightarrow$ вставка на позицию 1 (второй элемент).

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 2, 4, 6.

(h) Вызовите display.

(i) Вставьте 5 ($5 \% 3 = 2 \rightarrow$ вставка на позицию 2, т.е. после 4, перед 6).

(j) Снова вызовите display.

(k) Удалите 4.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(2)
dll.insert(4)
dll.insert(6)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(5)
print("After inserting 5 at position 5 % 3 = 2:")
dll.display()

dll.delete(4)
print("After deleting 4:")
dll.display()
```

12. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

Инструкции:

(a) Создайте класс Node с методом __init__, который принимает content и сохраняет его в self._stored_value. Инициализирует self._connection_next и self._connection_prev как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._input и self._output как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в формате: "List: [значения через пробел] (размер: N)". Если пуст — "List: [] (размер: 0)".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно не отрицательное**. Вставляет в конец.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение, только если значение положительное**. Если значение <=0 — ничего не делает.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: -1 (игнорируется), 8, 0, 12, -5 (игнорируется).

(h) Вызовите display.

(i) Вставьте 10.

(j) Снова вызовите display.

(k) Удалите 0 (не удаляется, т.к. не положительное).

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(-1)  # игнорируется
dll.insert(8)
dll.insert(0)
dll.insert(12)
dll.insert(-5)  # игнорируется

print("Initial Doubly Linked List:")
dll.display()

dll.insert(10)
print("After inserting 10:")
dll.display()

dll.delete(0)  # не удаляется
print("After attempting to delete 0:")
dll.display()
```

13. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

Инструкции:

(a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._record. Инициализирует self._next_entry и self._prev_entry как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_record и self._tail_record как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Записи: val1, val2, ..., valN". Если пуст — "Записей нет".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **в начало, если значение нечетное, и в конец, если четное**.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все узлы с этим значением**.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 3 (в начало), 4 (в конец), 5 (в начало), 6 (в конец).

(h) Вызовите display.

(i) Вставьте 7 (в начало).

(j) Снова вызовите display.

(k) Удалите все вхождения 4.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(3)
dll.insert(4)
dll.insert(5)
dll.insert(6)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(7)
print("After inserting 7:")
dll.display()

dll.delete(4)
print("After deleting all 4s:")
dll.display()
```

14. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._cell. Инициализирует self._cell_next и self._cell_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_cell и self._last_cell как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Ячейки: [значения]"и отдельно "Количество: N". Если пуст — "Список ячеек пуст".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **после каждого узла, значение которого кратно 3** (если таких нет — вставляет в конец).

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 6, 9, 4.

(h) Вызовите display.

(i) Вставьте 12 (вставится после 6 и после 9 — но по условию вставляется только один узел; вставим после первого кратного 3, т.е. после 6).

(j) Снова вызовите display.

(k) Удалите 9.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(6)
dll.insert(9)
dll.insert(4)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(12)
print("After inserting 12 after first multiple of 3:")
dll.display()

dll.delete(9)
print("After deleting 9:")
dll.display()
```

15. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._slot. Инициализирует self._slot_next и self._slot_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._start_slot и self._end_slot как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Слоты: val1 | val2 | val3". Если пуст — "Слоты отсутствуют".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **перед каждым узлом, значение которого кратно 5** (если таких нет — вставляет в начало).

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 10, 15, 7.

    (h) Вызовите display.

    (i) Вставьте 5 (вставится перед 10 и перед 15 — но по условию вставляется только один узел; вставим перед первым кратным 5, т.е. перед 10).

    (j) Снова вызовите display.

    (k) Удалите последнее вхождение 15.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(10)
dll.insert(15)
dll.insert(7)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(5)
print("After inserting 5 before first multiple of 5:")
dll.display()

dll.delete(15)
print("After deleting last occurrence of 15:")
dll.display()
```

16. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._block. Инициализирует self._block_next и self._block_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_block и self._tail_block как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Блоки: [значения]"и "Обратный порядок: [значения в обратном порядке]". Если пуст — "Нет блоков".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если сумма цифр значения четная**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 23 (2+3=5 — нечет, не вставляется), 24 (2+4=6 — чет, вставляется), 35 (3+5=8 — чет, вставляется), 13 (1+3=4 — чет, вставляется).

    (h) Вызовите display.

    (i) Вставьте 46 (4+6=10 — чет, вставляется).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 24.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(23)  # не вставляется
dll.insert(24)
```

```
dll.insert(35)
dll.insert(13)

print("Initial Doubly Linked List:")
dll.display()

dll.insert(46)
print("After inserting 46:")
dll.display()

dll.delete(24)
print("After deleting all 24s:")
dll.display()
```

17. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._unit. Инициализирует self._unit_next и self._unit_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_unit и self._last_unit как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Единицы: val1 → val2 → val3 → null". Если пуст — "null".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно простое число** (используйте вспомогательную функцию is_prime). Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте вспомогательную функцию is_prime(n).

    (g) Создайте экземпляр класса DoublyLinkedList.

    (h) Вставьте узлы: 4 (не простое), 5 (простое), 6 (не простое), 7 (простое), 8 (не простое), 11 (простое).

    (i) Вызовите display.

    (j) Вставьте 13 (простое).

    (k) Снова вызовите display.

    (l) Удалите 7.

    (m) Снова вызовите display.

    Пример использования:

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

```
dll = DoublyLinkedList()
dll.insert(4)    # нет
dll.insert(5)    # да
dll.insert(6)    # нет
dll.insert(7)    # да
dll.insert(8)    # нет
dll.insert(11)   # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(13)
print("After inserting 13:")
dll.display()

dll.delete(7)
print("After deleting 7:")
dll.display()
```

18. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._segment. Инициализирует self._seg_next и self._seg_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_seg и self._tail_seg как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Сегменты (вперед): ... "Сегменты (назад): ...". Если пуст — "Список сегментов пуст".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно палиндром** (например, 121, 33). Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 12 (не палиндром), 22 (палиндром), 34 (не палиндром), 55 (палиндром), 121 (палиндром).

    (h) Вызовите display.

    (i) Вставьте 33 (палиндром).

    (j) Снова вызовите display.

    (k) Удалите последнее вхождение 55.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(12)  # нет
dll.insert(22)  # да
dll.insert(34)  # нет
dll.insert(55)  # да
dll.insert(121) # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(33)
print("After inserting 33:")
dll.display()

dll.delete(55)
print("After deleting last occurrence of 55:")
dll.display()
```

19. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._piece. Инициализирует self._piece_next и self._piece_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_piece и self._last_piece как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Части: val1 - val2 - val3". Если пуст — "Нет частей".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно степень двойки** (1,2,4,8,16...). Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 3 (нет), 4 (да), 5 (нет), 8 (да), 9 (нет), 16 (да).

    (h) Вызовите display.

    (i) Вставьте 32 (да).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 8.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(3)   # нет
dll.insert(4)   # да
dll.insert(5)   # нет
dll.insert(8)   # да
dll.insert(9)   # нет
```

```
dll.insert(16)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(32)
print("After inserting 32:")
dll.display()

dll.delete(8)
print("After deleting all 8s:")
dll.display()
```

20. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._fragment. Инициализирует self._frag_next и self._frag_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._start_frag и self._end_frag как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Фрагменты → val1 → val2 → val3 → конец". Если пуст — "Фрагменты: конец".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно делится на 3 без остатка**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 1 (нет), 3 (да), 4 (нет), 6 (да), 7 (нет), 9 (да).

    (h) Вызовите display.

    (i) Вставьте 12 (да).

    (j) Снова вызовите display.

    (k) Удалите 6.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(1)  # нет
dll.insert(3)  # да
dll.insert(4)  # нет
dll.insert(6)  # да
dll.insert(7)  # нет
dll.insert(9)  # да

print("Initial Doubly Linked List:")
dll.display()
```

```
dll.insert(12)
print("After inserting 12:")
dll.display()

dll.delete(6)
print("After deleting 6:")
dll.display()
```

21. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._chunk. Инициализирует self._chunk_next и self._chunk_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_chunk и self._tail_chunk как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Чанки: [значения]"и "Размер: N". Если пуст — "Чанков нет".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно не делится на 5**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 10 (делится на 5 — не вставляется), 11 (не делится — вставляется), 15 (делится — не вставляется), 16 (не делится — вставляется), 20 (делится — не вставляется), 21 (не делится — вставляется).

    (h) Вызовите display.

    (i) Вставьте 26 (не делится — вставляется).

    (j) Снова вызовите display.

    (k) Удалите последнее вхождение 16.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(10)  # нет
dll.insert(11)  # да
dll.insert(15)  # нет
dll.insert(16)  # да
dll.insert(20)  # нет
dll.insert(21)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(26)
print("After inserting 26:")
dll.display()
```

```
dll.delete(16)
print("After deleting last occurrence of 16:")
dll.display()
```

22. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._item_data. Инициализирует self._next_item и self._prev_item как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_data и self._last_data как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Данные (→): val1, val2, val3"и "Данные (←): val3, val2, val1". Если пуст — "Данные отсутствуют".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно больше 10**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 5 (нет), 15 (да), 8 (нет), 20 (да), 12 (да).

    (h) Вызовите display.

    (i) Вставьте 25 (да).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 20.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(5)    # нет
dll.insert(15)   # да
dll.insert(8)    # нет
dll.insert(20)   # да
dll.insert(12)   # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(25)
print("After inserting 25:")
dll.display()

dll.delete(20)
print("After deleting all 20s:")
dll.display()
```

23. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._node_value. Инициализирует self._node_next и self._node_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._start_node и self._end_node как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Узлы: val1 <-> val2 <-> val3". Если пуст — "Нет узлов".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно меньше 50**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 60 (нет), 30 (да), 70 (нет), 40 (да), 45 (да).

    (h) Вызовите display.

    (i) Вставьте 25 (да).

    (j) Снова вызовите display.

    (k) Удалите 40.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(60)  # нет
dll.insert(30)  # да
dll.insert(70)  # нет
dll.insert(40)  # да
dll.insert(45)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(25)
print("After inserting 25:")
dll.display()

dll.delete(40)
print("After deleting 40:")
dll.display()
```

24. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

(a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._data_item. Инициализирует self._item_next и self._item_prev как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_item и self._tail_item как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Элементы списка: val1 val2 val3 (всего N)". Если пуст — "Список пуст".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно не равно 0**. Вставляет в конец.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 0 (нет), 10 (да), 0 (нет), 20 (да), 30 (да).

(h) Вызовите display.

(i) Вставьте 40 (да).

(j) Снова вызовите display.

(k) Удалите последнее вхождение 20.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(0)   # нет
dll.insert(10)  # да
dll.insert(0)   # нет
dll.insert(20)  # да
dll.insert(30)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(40)
print("After inserting 40:")
dll.display()

dll.delete(20)
print("After deleting last occurrence of 20:")
dll.display()
```

25. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._list_data. Инициализирует self._data_next и self._data_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_list и self._last_list как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Список: val1 | val2 | val3 | ...". Если пуст — "Пустой список".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно положительное**. Вставляет в начало.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: -5 (нет), 15 (да), -3 (нет), 25 (да), 0 (нет, если считать 0 не положительным).

(h) Вызовите display.

(i) Вставьте 35 (да).

(j) Снова вызовите display.

(k) Удалите все вхождения 25.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(-5)  # нет
dll.insert(15)  # да
dll.insert(-3)  # нет
dll.insert(25)  # да
dll.insert(0)   # нет

print("Initial Doubly Linked List:")
dll.display()

dll.insert(35)
print("After inserting 35:")
dll.display()

dll.delete(25)
print("After deleting all 25s:")
dll.display()
```

26. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

Инструкции:

(a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._entry_value. Инициализирует self._value_next и self._value_prev как None.

(b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_value и self._tail_value как None.

(c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Значения → val1 → val2 → val3 → конец". Если пуст — "→ конец".

(d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно нечетное**. Вставляет в конец.

(e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

(f) Создайте экземпляр класса DoublyLinkedList.

(g) Вставьте узлы: 2 (нет), 3 (да), 4 (нет), 5 (да), 6 (нет), 7 (да).

(h) Вызовите display.

(i) Вставьте 9 (да).

(j) Снова вызовите display.

(k) Удалите 5.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(2)  # нет
dll.insert(3)  # да
dll.insert(4)  # нет
dll.insert(5)  # да
dll.insert(6)  # нет
dll.insert(7)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(9)
print("After inserting 9:")
dll.display()

dll.delete(5)
print("After deleting 5:")
dll.display()
```

27. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._data_point. Инициализирует self._point_next и self._point_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._start_point и self._end_point как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Точки: val1, val2, val3 (обратно: val3, val2, val1)". Если пуст — "Точек нет".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно четное**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 1 (нет), 4 (да), 3 (нет), 6 (да), 5 (нет), 8 (да).

(h) Вызовите display.

(i) Вставьте 10 (да).

(j) Снова вызовите display.

(k) Удалите последнее вхождение 6.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(1)  # нет
dll.insert(4)  # да
dll.insert(3)  # нет
dll.insert(6)  # да
dll.insert(5)  # нет
dll.insert(8)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(10)
print("After inserting 10:")
dll.display()

dll.delete(6)
print("After deleting last occurrence of 6:")
dll.display()
```

28. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._node_data. Инициализирует self._data_link_next и self._data_link_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_link и self._last_link как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Связи: val1 <-> val2 <-> val3". Если пуст — "Связи отсутствуют".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно кратно 4**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 2 (нет), 4 (да), 6 (нет), 8 (да), 10 (нет), 12 (да).

    (h) Вызовите display.

    (i) Вставьте 16 (да).

    (j) Снова вызовите display.

(k) Удалите все вхождения 8.

(l) Снова вызовите display.

Пример использования:

```
dll = DoublyLinkedList()
dll.insert(2)    # нет
dll.insert(4)    # да
dll.insert(6)    # нет
dll.insert(8)    # да
dll.insert(10)   # нет
dll.insert(12)   # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(16)
print("After inserting 16:")
dll.display()

dll.delete(8)
print("After deleting all 8s:")
dll.display()
```

29. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._item_val. Инициализирует self._val_next и self._val_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_val и self._tail_val как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Значения: val1 - val2 - val3 (размер N)". Если пуст — "Нет значений".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно заканчивается на 5**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 10 (нет), 15 (да), 20 (нет), 25 (да), 30 (нет), 35 (да).

    (h) Вызовите display.

    (i) Вставьте 45 (да).

    (j) Снова вызовите display.

    (k) Удалите 25.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(10)  # нет
dll.insert(15)  # да
dll.insert(20)  # нет
dll.insert(25)  # да
dll.insert(30)  # нет
dll.insert(35)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(45)
print("After inserting 45:")
dll.display()

dll.delete(25)
print("After deleting 25:")
dll.display()
```

30. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._data_field. Инициализирует self._field_next и self._field_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_field и self._last_field как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Поля: val1 → val2 → val3 → null". Если пуст — "null".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если первая цифра числа — 1**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 5 (нет), 12 (да), 23 (нет), 18 (да), 31 (нет), 19 (да).

    (h) Вызовите display.

    (i) Вставьте 11 (да).

    (j) Снова вызовите display.

    (k) Удалите последнее вхождение 18.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(5)   # нет
dll.insert(12)  # да
dll.insert(23)  # нет
dll.insert(18)  # да
```

```
dll.insert(31)  # нет
dll.insert(19)  # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(11)
print("After inserting 11:")
dll.display()

dll.delete(18)
print("After deleting last occurrence of 18:")
dll.display()
```

31. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._record_data. Инициализирует self._data_record_next и self._data_record_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_record и self._tail_record как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Записи: [val1, val2, val3]". Если пуст — "[]".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно начинается с цифры 2**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 15 (нет), 25 (да), 35 (нет), 28 (да), 45 (нет), 22 (да).

    (h) Вызовите display.

    (i) Вставьте 20 (да).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 28.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(15)  # нет
dll.insert(25)  # да
dll.insert(35)  # нет
dll.insert(28)  # да
dll.insert(45)  # нет
dll.insert(22)  # да

print("Initial Doubly Linked List:")
dll.display()
```

```
dll.insert(20)
print("After inserting 20:")
dll.display()

dll.delete(28)
print("After deleting all 28s:")
dll.display()
```

32. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._cell_value. Инициализирует self._value_cell_next и self._value_cell_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_cell и self._last_cell как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Ячейки: val1 | val2 | val3 (всего N)". Если пуст — "Нет ячеек".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если сумма его цифр нечетная**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 12 (1+2=3 — нечет, да), 14 (1+4=5 — нечет, да), 16 (1+6=7 — нечет, да), 18 (1+8=9 — нечет, да), 20 (2+0=2 — чет, нет).

    (h) Вызовите display.

    (i) Вставьте 21 (2+1=3 — нечет, да).

    (j) Снова вызовите display.

    (k) Удалите 16.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(12)  # да
dll.insert(14)  # да
dll.insert(16)  # да
dll.insert(18)  # да
dll.insert(20)  # нет

print("Initial Doubly Linked List:")
dll.display()

dll.insert(21)
print("After inserting 21:")
dll.display()
```

```
dll.delete(16)
print("After deleting 16:")
dll.display()
```

33. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает item и сохраняет его в self._slot_data. Инициализирует self._data_slot_next и self._data_slot_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_slot и self._tail_slot как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Слоты → val1 → val2 → val3 → конец". Если пуст — "→ конец".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно заканчивается на 0**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **последнее вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 5 (нет), 10 (да), 15 (нет), 20 (да), 25 (нет), 30 (да).

    (h) Вызовите display.

    (i) Вставьте 40 (да).

    (j) Снова вызовите display.

    (k) Удалите последнее вхождение 20.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(5)    # нет
dll.insert(10)   # да
dll.insert(15)   # нет
dll.insert(20)   # да
dll.insert(25)   # нет
dll.insert(30)   # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(40)
print("After inserting 40:")
dll.display()

dll.delete(20)
print("After deleting last occurrence of 20:")
dll.display()
```

34. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает data и сохраняет его в self._block_data. Инициализирует self._data_block_next и self._data_block_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._first_block и self._last_block как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Блоки: val1, val2, val3 (обратный: val3, val2, val1)". Если пуст — "Пусто".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно простое и больше 10**. Вставляет в конец.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **все вхождения**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 7 (простое, но <=10 — нет), 11 (да), 13 (да), 15 (нет), 17 (да), 9 (нет).

    (h) Вызовите display.

    (i) Вставьте 19 (да).

    (j) Снова вызовите display.

    (k) Удалите все вхождения 13.

    (l) Снова вызовите display.

    Пример использования:

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

dll = DoublyLinkedList()
dll.insert(7)    # нет
dll.insert(11)   # да
dll.insert(13)   # да
dll.insert(15)   # нет
dll.insert(17)   # да
dll.insert(9)    # нет

print("Initial Doubly Linked List:")
dll.display()

dll.insert(19)
print("After inserting 19:")
dll.display()

dll.delete(13)
```

```
print("After deleting all 13s:")
dll.display()
```

35. Написать программу на Python, которая создает класс DoublyLinkedList, представляющий **двусвязный список** с инкапсуляцией. Класс должен содержать методы для отображения данных, вставки и удаления узлов. Программа также должна создавать экземпляр класса, вставлять узлы и удалять узлы.

    Инструкции:

    (a) Создайте класс Node с методом __init__, который принимает value и сохраняет его в self._unit_value. Инициализирует self._value_unit_next и self._value_unit_prev как None.

    (b) Создайте класс DoublyLinkedList с методом __init__, который инициализирует self._head_unit и self._tail_unit как None.

    (c) Создайте метод display в классе DoublyLinkedList, который выводит элементы в виде: "Единицы: val1 <-> val2 <-> val3". Если пуст — "Нет данных".

    (d) Создайте метод insert в классе DoublyLinkedList, который принимает значение и вставляет его **только если оно палиндром и двузначное**. Вставляет в начало.

    (e) Создайте метод delete в классе DoublyLinkedList, который принимает значение и удаляет **первое вхождение**.

    (f) Создайте экземпляр класса DoublyLinkedList.

    (g) Вставьте узлы: 121 (трехзначное — нет), 22 (да), 34 (нет), 55 (да), 5 (однозначное — нет), 66 (да).

    (h) Вызовите display.

    (i) Вставьте 77 (да).

    (j) Снова вызовите display.

    (k) Удалите 55.

    (l) Снова вызовите display.

    Пример использования:

```
dll = DoublyLinkedList()
dll.insert(121)   # нет
dll.insert(22)    # да
dll.insert(34)    # нет
dll.insert(55)    # да
dll.insert(5)     # нет
dll.insert(66)    # да

print("Initial Doubly Linked List:")
dll.display()

dll.insert(77)
print("After inserting 77:")
dll.display()

dll.delete(55)
print("After deleting 55:")
dll.display()
```

2.3.4 Задача 4 (очередь)

1. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (внутренний список _elements). Принимает необязательный параметр max_size (по умолчанию None — без ограничений).

   (b) Создайте метод enqueue, который принимает элемент и добавляет его в конец очереди, только если не превышает max_size. Если превышает — выбрасывает ValueError("Очередь переполнена").

   (c) Создайте метод dequeue, который удаляет и возвращает элемент из начала очереди. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

   (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

   (e) Создайте приватный метод _debug_list (только для отладки, не включайте в задание студентам; в решении можно использовать queue._elements) для вывода внутреннего состояния.

   (f) Создайте экземпляр класса Queue с max_size=5.

   (g) Добавьте элементы: 100, 200, 300, 400, 500.

   (h) Попытайтесь добавить 600 — должно вызвать исключение (перехватите его и выведите сообщение).

   (i) Выведите текущее состояние очереди.

   (j) Вызовите dequeue дважды, выводя каждый раз удаленный элемент.

   (k) Выведите обновленное состояние очереди.

   Пример использования:

```
queue = Queue(max_size=5)
queue.enqueue(100)
queue.enqueue(200)
queue.enqueue(300)
queue.enqueue(400)
queue.enqueue(500)

try:
    queue.enqueue(600)
except ValueError as e:
    print("Ошибка:", e)

print("Current Queue:", queue._elements)  # только для проверки

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)

dequeued_item = queue.dequeue()
```

```
print("Dequeued item:", dequeued_item)

print("Updated Queue:", queue._elements)
```

2. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _items). Принимает параметр allow_duplicates=True. Если False, то не добавляет элемент, если он уже есть в очереди.

   (b) Создайте метод enqueue, который принимает элемент. Если allow_duplicates=False и элемент уже есть в очереди — не добавляет и возвращает False. Иначе — добавляет в конец и возвращает True.

   (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — возвращает None (не выбрасывает исключение).

   (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

   (e) Создайте экземпляр класса Queue с allow_duplicates=False.

   (f) Добавьте элементы: 10, 20, 10 (не добавится), 30, 20 (не добавится), 40.

   (g) Выведите текущее состояние очереди.

   (h) Вызовите dequeue трижды, выводя каждый раз удаленный элемент.

   (i) Выведите обновленное состояние очереди.

   Пример использования:

```
queue = Queue(allow_duplicates=False)
print(queue.enqueue(10))  # True
print(queue.enqueue(20))  # True
print(queue.enqueue(10))  # False
print(queue.enqueue(30))  # True
print(queue.enqueue(20))  # False
print(queue.enqueue(40))  # True

print("Current Queue:", queue._items)

for _ in range(3):
    dequeued_item = queue.dequeue()
    print("Dequeued item:", dequeued_item)

print("Updated Queue:", queue._items)
```

3. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _data). Принимает параметр auto_reverse=False. Если True, то enqueue добавляет в начало, а dequeue удаляет с конца (поведение стека, но интерфейс очереди).

(b) Создайте метод enqueue, который добавляет элемент: если auto_reverse=False — в конец, если True — в начало.

(c) Создайте метод dequeue, который удаляет и возвращает элемент: если auto_reverse=False — из начала, если True — из конца. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с auto_reverse=True.

(f) Добавьте элементы: 1, 2, 3, 4, 5.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue дважды, выводя каждый раз удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(auto_reverse=True)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.enqueue(4)
queue.enqueue(5)

print("Current Queue:", queue._data)  # [5,4,3,2,1]

dequeued_item = queue.dequeue()  # удаляет 1
print("Dequeued item:", dequeued_item)

dequeued_item = queue.dequeue()  # удаляет 2
print("Dequeued item:", dequeued_item)

print("Updated Queue:", queue._data)  # [5,4,3]
```

4. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _buffer). Принимает параметр dequeue_all_at_once=False. Если True, то dequeue возвращает список всех элементов и очищает очередь.

   (b) Создайте метод enqueue, который добавляет элемент в конец очереди.

(c) Создайте метод dequeue, который, если dequeue_all_at_once=False, удаляет и возвращает первый элемент. Если True — возвращает список всех элементов и очищает очередь. Если очередь пуста — возвращает пустой список [].

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с dequeue_all_at_once=True.

(f) Добавьте элементы: 5, 15, 25, 35.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue (вернет [5,15,25,35] и очистит очередь).

(i) Выведите результат dequeue и состояние очереди после вызова.

Пример использования:

```
queue = Queue(dequeue_all_at_once=True)
queue.enqueue(5)
queue.enqueue(15)
queue.enqueue(25)
queue.enqueue(35)

print("Current Queue:", queue._buffer)

dequeued_items = queue.dequeue()
print("Dequeued items:", dequeued_items)  # [5,15,25,35]
print("Updated Queue:", queue._buffer)    # []
```

5. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _store). Принимает параметр on_enqueue_callback=None — функция, вызываемая при каждом добавлении (с аргументом — добавленным элементом).

   (b) Создайте метод enqueue, который добавляет элемент в конец и, если on_enqueue_callback не None, вызывает её с элементом.

   (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Нельзя извлечь из пустой очереди").

   (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

   (e) Создайте функцию printer(x): print(f"[+] Добавлен: x")

   (f) Создайте экземпляр класса Queue, передав printer в on_enqueue_callback.

   (g) Добавьте элементы: 101, 202, 303.

   (h) Выведите текущее состояние очереди.

   (i) Вызовите dequeue, выведите удаленный элемент.

(j) Выведите обновленное состояние очереди.

Пример использования:

```
def printer(x):
    print(f"[+] Добавлен: {x}")

queue = Queue(on_enqueue_callback=printer)
queue.enqueue(101)  # [+] Добавлен: 101
queue.enqueue(202)  # [+] Добавлен: 202
queue.enqueue(303)  # [+] Добавлен: 303

print("Current Queue:", queue._store)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)

print("Updated Queue:", queue._store)
```

6. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _pool). Принимает параметр compress_on_enqueue=False. Если True, то при добавлении элемента, равного последнему в очереди, вместо добавления увеличивает счетчик дубликатов у последнего элемента (хранит пары (элемент, счетчик)).

   (b) Создайте метод enqueue, который, если compress_on_enqueue=True и очередь не пуста и элемент == последний_элемент, увеличивает счетчик последнего элемента. Иначе — добавляет новый элемент (со счетчиком 1, если режим сжатия включен).

   (c) Создайте метод dequeue, который удаляет первый элемент. Если режим сжатия включен и счетчик >1, уменьшает счетчик и возвращает элемент. Если счетчик=1, удаляет элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

   (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

   (e) Создайте экземпляр класса Queue с compress_on_enqueue=True.

   (f) Добавьте элементы: 7, 7, 7, 14, 14, 21.

   (g) Выведите текущее состояние очереди (внутреннее представление).

   (h) Вызовите dequeue трижды, выводя каждый раз удаленный элемент.

   (i) Выведите обновленное состояние очереди.

   Пример использования:

```
queue = Queue(compress_on_enqueue=True)
queue.enqueue(7)
queue.enqueue(7)
queue.enqueue(7)
queue.enqueue(14)
queue.enqueue(14)
queue.enqueue(21)

print("Current Queue:", queue._pool)  # [(7,3), (14,2), (21,1)]

for _ in range(3):
    dequeued_item = queue.dequeue()
    print("Dequeued item:", dequeued_item)  # 7, 7, 7

print("Updated Queue:", queue._pool)  # [(14,2), (21,1)]
```

7. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _line). Принимает параметр immutable_dequeue=False. Если True, то dequeue возвращает первый элемент, но не удаляет его.

   (b) Создайте метод enqueue, который добавляет элемент в конец очереди.

   (c) Создайте метод dequeue, который, если immutable_dequeue=False, удаляет и возвращает первый элемент. Если True — возвращает первый элемент, не удаляя его. Если очередь пуста — возвращает None.

   (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

   (e) Создайте экземпляр класса Queue с immutable_dequeue=True.

   (f) Добавьте элементы: 1, 3, 5.

   (g) Выведите текущее состояние очереди.

   (h) Вызовите dequeue дважды, выводя каждый раз результат (должен быть 1 оба раза).

   (i) Выведите состояние очереди (не должно измениться).

   Пример использования:

```
queue = Queue(immutable_dequeue=True)
queue.enqueue(1)
queue.enqueue(3)
queue.enqueue(5)

print("Current Queue:", queue._line)

print("Dequeued item:", queue.dequeue())  # 1
print("Dequeued item:", queue.dequeue())  # 1 (не удалилось)

print("Updated Queue:", queue._line)  # [1,3,5]
```

8. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _stream). Принимает параметр track_history=False. Если True, сохраняет историю всех когда-либо добавленных элементов (даже удаленных) в отдельном списке _history.

   (b) Создайте метод enqueue, который добавляет элемент в конец _stream и, если track_history=True, добавляет его в _history.

   (c) Создайте метод dequeue, который удаляет и возвращает первый элемент из _stream. Если очередь пуста — выбрасывает IndexError("Пусто").

   (d) Создайте метод is_empty, который возвращает True, если _stream пуст, и False в противном случае.

   (e) Создайте метод get_history (только если track_history=True), возвращающий копию _history.

   (f) Создайте экземпляр класса Queue с track_history=True.

   (g) Добавьте элементы: 2, 4, 6.

   (h) Вызовите dequeue (вернет 2).

   (i) Добавьте 8.

   (j) Выведите текущую очередь и историю.

   Пример использования:

```
queue = Queue(track_history=True)
queue.enqueue(2)
queue.enqueue(4)
queue.enqueue(6)
queue.dequeue()  # 2
queue.enqueue(8)

print("Current Queue:", queue._stream)     # [4,6,8]
print("History:", queue.get_history())     # [2,4,6,8]
```

9. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

   Инструкции:

   (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _flow). Принимает параметр enqueue_only_even=False. Если True, то добавляются только четные числа.

(b) Создайте метод enqueue, который добавляет элемент в конец, только если enqueue_only_even=False или элемент четный.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_only_even=True.

(f) Добавьте элементы: 1 (игнорируется), 2, 3 (игнорируется), 4, 5 (игнорируется), 6.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_only_even=True)
queue.enqueue(1)  # игнорируется
queue.enqueue(2)
queue.enqueue(3)  # игнорируется
queue.enqueue(4)
queue.enqueue(5)  # игнорируется
queue.enqueue(6)

print("Current Queue:", queue._flow)  # [2,4,6]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 2

print("Updated Queue:", queue._flow)  # [4,6]
```

10. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _pipe). Принимает параметр reverse_dequeue=False. Если True, то dequeue удаляет и возвращает не первый, а последний элемент.

    (b) Создайте метод enqueue, который добавляет элемент в конец очереди.

    (c) Создайте метод dequeue, который, если reverse_dequeue=False, удаляет и возвращает первый элемент. Если True — удаляет и возвращает последний элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с reverse_dequeue=True.

    (f) Добавьте элементы: 10, 20, 30.

    (g) Выведите текущее состояние очереди.

(h) Вызовите dequeue — должен вернуться 30 (последний).

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(reverse_dequeue=True)
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print("Current Queue:", queue._pipe)  # [10,20,30]

dequeued_item = queue.dequeue()  # 30
print("Dequeued item:", dequeued_item)

print("Updated Queue:", queue._pipe)  # [10,20]
```

11. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _channel). Принимает параметр enqueue_with_timestamp=False. Если True, то при добавлении сохраняет пару (элемент, time.time()).

    (b) Создайте метод enqueue, который, если enqueue_with_timestamp=True, добавляет (элемент, timestamp). Иначе — элемент.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент (или пару). Если очередь пуста — выбрасывает IndexError("Очередь пуста").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_with_timestamp=True.

    (f) Добавьте элементы: "first "second "third".

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите результат (пару).

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
import time

queue = Queue(enqueue_with_timestamp=True)
queue.enqueue("first")
queue.enqueue("second")
queue.enqueue("third")

print("Current Queue:", queue._channel)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # ('first', timestamp)

print("Updated Queue:", queue._channel)
```

12. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _tube). Принимает параметр enqueue_pairs=False. Если True, то enqueue принимает два аргумента (key, value) и сохраняет кортеж (key, value).

    (b) Создайте метод enqueue, который, если enqueue_pairs=False, принимает один элемент. Если True — два аргумента и сохраняет кортеж.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент (или кортеж). Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_pairs=True.

    (f) Добавьте пары: ("a 1), ("b 2), ("c 3).

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите результат.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_pairs=True)
queue.enqueue("a", 1)
queue.enqueue("b", 2)
queue.enqueue("c", 3)

print("Current Queue:", queue._tube)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # ('a', 1)

print("Updated Queue:", queue._tube)
```

13. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _conduit). Принимает параметр auto_dedup=False. Если True, то при добавлении, если элемент уже есть в очереди, сначала удаляет все его предыдущие вхождения.

(b) Создайте метод enqueue, который, если auto_dedup=True и элемент уже есть, удаляет все его вхождения, затем добавляет в конец. Иначе — просто добавляет.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с auto_dedup=True.

(f) Добавьте элементы: 1, 2, 1, 3, 2, 4.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(auto_dedup=True)
queue.enqueue(1)  # [1]
queue.enqueue(2)  # [1,2]
queue.enqueue(1)  # удаляет старую 1 -> [2,1]
queue.enqueue(3)  # [2,1,3]
queue.enqueue(2)  # удаляет 2 -> [1,3,2]
queue.enqueue(4)  # [1,3,2,4]

print("Current Queue:", queue._conduit)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 1

print("Updated Queue:", queue._conduit)  # [3,2,4]
```

14. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _duct). Принимает параметр enqueue_if_max=False. Если True, то элемент добавляется только если он больше всех текущих элементов в очереди.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_if_max=False или элемент > всех элементов в очереди.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_if_max=True.

(f) Добавьте элементы: 5, 3 (не добавится), 10, 7 (не добавится), 15.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_if_max=True)
queue.enqueue(5)
queue.enqueue(3)    # не добавится
queue.enqueue(10)
queue.enqueue(7)    # не добавится
queue.enqueue(15)

print("Current Queue:", queue._duct)  # [5,10,15]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._duct)  # [10,15]
```

15. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _pipe). Принимает параметр cumulative=False. Если True, то при добавлении элемент становится element + последний_элемент (если очередь не пуста). Первый элемент добавляется как есть.

    (b) Создайте метод enqueue, который, если cumulative=True и очередь не пуста, добавляет element + последний_элемент. Иначе — element.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с cumulative=True.

    (f) Добавьте элементы: 1, 2, 3, 4.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(cumulative=True)
queue.enqueue(1)  # [1]
queue.enqueue(2)  # [1, 1+2=3]
queue.enqueue(3)  # [1,3, 3+3=6]
queue.enqueue(4)  # [1,3,6, 6+4=10]

print("Current Queue:", queue._pipe)
```

```
dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 1

print("Updated Queue:", queue._pipe)  # [3,6,10]
```

16. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _line). Принимает параметр enqueue_squared=False. Если True, то при добавлении сохраняется element**2.

    (b) Создайте метод enqueue, который добавляет element**2, если enqueue_squared=True, иначе — element.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_squared=True.

    (f) Добавьте элементы: 2, 3, 4, 5.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_squared=True)
queue.enqueue(2)  # 4
queue.enqueue(3)  # 9
queue.enqueue(4)  # 16
queue.enqueue(5)  # 25

print("Current Queue:", queue._line)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 4

print("Updated Queue:", queue._line)  # [9,16,25]
```

17. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _stream). Принимает параметр enqueue_absolute=False. Если True, то при добавлении сохраняется abs(element).

(b) Создайте метод enqueue, который добавляет abs(element), если enqueue_absolute=True, иначе — element.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_absolute=True.

(f) Добавьте элементы: -5, 3, -8, 2.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_absolute=True)
queue.enqueue(-5)   # 5
queue.enqueue(3)    # 3
queue.enqueue(-8)   # 8
queue.enqueue(2)    # 2

print("Current Queue:", queue._stream)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._stream)  # [3,8,2]
```

18. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _buffer). Принимает параметр enqueue_rounded=False. Если True, то при добавлении сохраняется round(element).

    (b) Создайте метод enqueue, который добавляет round(element), если enqueue_rounded=True, иначе — element.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_rounded=True.

    (f) Добавьте элементы: 3.2, 4.7, 5.1, 6.9.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_rounded=True)
queue.enqueue(3.2)  # 3
queue.enqueue(4.7)  # 5
queue.enqueue(5.1)  # 5
queue.enqueue(6.9)  # 7

print("Current Queue:", queue._buffer)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 3

print("Updated Queue:", queue._buffer)  # [5,5,7]
```

19. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _store). Принимает параметр enqueue_negated=False. Если True, то при добавлении сохраняется -element.

    (b) Создайте метод enqueue, который добавляет -element, если enqueue_negated=True, иначе — element.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_negated=True.

    (f) Добавьте элементы: 10, 20, 30, 40.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_negated=True)
queue.enqueue(10)  # -10
queue.enqueue(20)  # -20
queue.enqueue(30)  # -30
queue.enqueue(40)  # -40

print("Current Queue:", queue._store)
```

```
dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # -10

print("Updated Queue:", queue._store)  # [-20,-30,-40]
```

20. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _pool). Принимает параметр enqueue_doubled=False. Если True, то при добавлении сохраняется element * 2.

    (b) Создайте метод enqueue, который добавляет element * 2, если enqueue_doubled=True, иначе — element.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_doubled=True.

    (f) Добавьте элементы: 1, 2, 3, 4.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_doubled=True)
queue.enqueue(1)  # 2
queue.enqueue(2)  # 4
queue.enqueue(3)  # 6
queue.enqueue(4)  # 8

print("Current Queue:", queue._pool)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 2

print("Updated Queue:", queue._pool)  # [4,6,8]
```

21. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _reservoir). Принимает параметр enqueue_halved=False. Если True, то при добавлении сохраняется element / 2.0.

(b) Создайте метод enqueue, который добавляет element / 2.0, если enqueue_halved=True, иначе — element.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_halved=True.

(f) Добавьте элементы: 4, 8, 12, 16.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_halved=True)
queue.enqueue(4)    # 2.0
queue.enqueue(8)    # 4.0
queue.enqueue(12)   # 6.0
queue.enqueue(16)   # 8.0

print("Current Queue:", queue._reservoir)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 2.0

print("Updated Queue:", queue._reservoir)  # [4.0,6.0,8.0]
```

22. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _tank). Принимает параметр enqueue_as_string=False. Если True, то при добавлении сохраняется str(element).

(b) Создайте метод enqueue, который добавляет str(element), если enqueue_as_string=True, иначе — element.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_as_string=True.

(f) Добавьте элементы: 100, 200, 300, 400.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_as_string=True)
queue.enqueue(100)  # "100"
queue.enqueue(200)  # "200"
queue.enqueue(300)  # "300"
queue.enqueue(400)  # "400"

print("Current Queue:", queue._tank)

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # "100"

print("Updated Queue:", queue._tank)  # ["200","300","400"]
```

23. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _container). Принимает параметр enqueue_with_index=False. Если True, то при добавлении сохраняется кортеж (element, порядковый_номер_добавления).

(b) Создайте метод enqueue, который добавляет (element, self._counter), где _counter — внутренний счетчик, увеличивающийся при каждом добавлении. Иначе — element.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент (или кортеж). Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_with_index=True.

(f) Добавьте элементы: "alpha "beta "gamma".

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_with_index=True)
queue.enqueue("alpha")  # ("alpha", 0)
queue.enqueue("beta")   # ("beta", 1)
queue.enqueue("gamma")  # ("gamma", 2)

print("Current Queue:", queue._container)

dequeued_item = queue.dequeue()
```

```
print("Dequeued item:", dequeued_item)  # ('alpha', 0)

print("Updated Queue:", queue._container)  # [('beta',1), ('gamma',2)]
```

24. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _vessel). Принимает параметр enqueue_unique_rear=False. Если True, то при добавлении, если элемент равен текущему последнему, он не добавляется.

    (b) Создайте метод enqueue, который добавляет элемент, только если enqueue_unique_rear=False или очередь пуста или element != последний_элемент.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_unique_rear=True.

    (f) Добавьте элементы: 1, 2, 2, 3, 3, 3, 4.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_unique_rear=True)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(2)  # не добавится
queue.enqueue(3)
queue.enqueue(3)  # не добавится
queue.enqueue(3)  # не добавится
queue.enqueue(4)

print("Current Queue:", queue._vessel)  # [1,2,3,4]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 1

print("Updated Queue:", queue._vessel)  # [2,3,4]
```

25. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _bin). Принимает параметр enqueue_even_only=False. Если True, то добавляются только четные числа.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_even_only=False или element % 2 == 0.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_even_only=True.

(f) Добавьте элементы: 1 (не добавится), 2, 3 (не добавится), 4, 5 (не добавится), 6.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_even_only=True)
queue.enqueue(1)  # нет
queue.enqueue(2)
queue.enqueue(3)  # нет
queue.enqueue(4)
queue.enqueue(5)  # нет
queue.enqueue(6)

print("Current Queue:", queue._bin)  # [2,4,6]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 2

print("Updated Queue:", queue._bin)  # [4,6]
```

26. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _box). Принимает параметр enqueue_odd_only=False. Если True, то добавляются только нечетные числа.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_odd_only=False или element % 2 != 0.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_odd_only=True.

(f) Добавьте элементы: 2 (не добавится), 1, 4 (не добавится), 3, 6 (не добавится), 5.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_odd_only=True)
queue.enqueue(2)  # нет
queue.enqueue(1)
queue.enqueue(4)  # нет
queue.enqueue(3)
queue.enqueue(6)  # нет
queue.enqueue(5)

print("Current Queue:", queue._box)  # [1,3,5]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 1

print("Updated Queue:", queue._box)  # [3,5]
```

27. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _crate). Принимает параметр enqueue_positive_only=False. Если True, то добавляются только положительные числа (>0).

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_positive_only=False или element > 0.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_positive_only=True.

(f) Добавьте элементы: -1 (не добавится), 0 (не добавится), 1, 2, -5 (не добавится), 3.

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_positive_only=True)
queue.enqueue(-1)  # нет
queue.enqueue(0)   # нет
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(-5)  # нет
queue.enqueue(3)

print("Current Queue:", queue._crate)  # [1,2,3]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 1

print("Updated Queue:", queue._crate)  # [2,3]
```

28. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _carton). Принимает параметр enqueue_nonzero_only=False. Если True, то добавляются только ненулевые числа.

    (b) Создайте метод enqueue, который добавляет элемент, только если enqueue_nonzero_only=False или element != 0.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_nonzero_only=True.

    (f) Добавьте элементы: 0 (не добавится), 5, 0 (не добавится), 10, 15.

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_nonzero_only=True)
queue.enqueue(0)   # нет
queue.enqueue(5)
queue.enqueue(0)   # нет
queue.enqueue(10)
queue.enqueue(15)

print("Current Queue:", queue._carton)  # [5,10,15]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._carton)  # [10,15]
```

29. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _package). Принимает параметр enqueue_prime_only=False. Если True, то добавляются только простые числа (реализуйте простую проверку).

    (b) Создайте метод enqueue, который добавляет элемент, только если enqueue_prime_only=False или element — простое число.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте вспомогательную функцию is_prime(n) (вне класса).

    (f) Создайте экземпляр класса Queue с enqueue_prime_only=True.

    (g) Добавьте элементы: 4 (не простое), 5 (простое), 6 (не простое), 7 (простое), 8 (не простое), 11 (простое).

    (h) Выведите текущее состояние очереди.

    (i) Вызовите dequeue, выведите удаленный элемент.

    (j) Выведите обновленное состояние очереди.

    Пример использования:

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

queue = Queue(enqueue_prime_only=True)
queue.enqueue(4)   # нет
queue.enqueue(5)   # да
queue.enqueue(6)   # нет
queue.enqueue(7)   # да
queue.enqueue(8)   # нет
queue.enqueue(11)  # да

print("Current Queue:", queue._package)  # [5,7,11]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._package)  # [7,11]
```

30. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue,

dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _parcel). Принимает параметр enqueue_fibonacci_only=False. Если True, то добавляются только числа Фибоначчи (до 100: 0,1,1,2,3,5,8,13,21,34,55,89).

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_fibonacci_only=False или element входит в FIB_SET.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_fibonacci_only=True.

(f) Добавьте элементы: 4 (не Фибоначчи), 5 (Фибоначчи), 6 (не Фибоначчи), 8 (Фибоначчи), 7 (не Фибоначчи), 13 (Фибоначчи).

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
FIB_SET = {0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89}

queue = Queue(enqueue_fibonacci_only=True)
queue.enqueue(4)   # нет
queue.enqueue(5)   # да
queue.enqueue(6)   # нет
queue.enqueue(8)   # да
queue.enqueue(7)   # нет
queue.enqueue(13)  # да

print("Current Queue:", queue._parcel)  # [5,8,13]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._parcel)  # [8,13]
```

31. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _sack). Принимает параметр enqueue_palindrome_only=False. Если True, то добавляются только числа-палиндромы.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_palindrome_only=False или element — палиндром (str(element) == str(element)[::-1]).

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_palindrome_only=True.

(f) Добавьте элементы: 12 (не палиндром), 22 (палиндром), 34 (не палиндром), 55 (палиндром), 123 (не палиндром), 121 (палиндром).

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_palindrome_only=True)
queue.enqueue(12)   # нет
queue.enqueue(22)   # да
queue.enqueue(34)   # нет
queue.enqueue(55)   # да
queue.enqueue(123)  # нет
queue.enqueue(121)  # да

print("Current Queue:", queue._sack)  # [22,55,121]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 22

print("Updated Queue:", queue._sack)  # [55,121]
```

32. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _bag). Принимает параметр enqueue_power_of_two=False. Если True, то добавляются только степени двойки.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_power_of_two=False или element $>$ 0 и (element & (element-1)) == 0.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_power_of_two=True.

(f) Добавьте элементы: 3 (не степень), 4 (степень), 5 (не степень), 8 (степень), 9 (не степень), 16 (степень).

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_power_of_two=True)
queue.enqueue(3)   # нет
queue.enqueue(4)   # да
queue.enqueue(5)   # нет
queue.enqueue(8)   # да
queue.enqueue(9)   # нет
queue.enqueue(16)  # да

print("Current Queue:", queue._bag)  # [4,8,16]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 4

print("Updated Queue:", queue._bag)  # [8,16]
```

33. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

Инструкции:

(a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _suitcase). Принимает параметр enqueue_divisible_by_three=False. Если True, то добавляются только числа, делящиеся на 3.

(b) Создайте метод enqueue, который добавляет элемент, только если enqueue_divisible_by_three=False или element % 3 == 0.

(c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

(d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

(e) Создайте экземпляр класса Queue с enqueue_divisible_by_three=True.

(f) Добавьте элементы: 1 (нет), 3 (да), 4 (нет), 6 (да), 7 (нет), 9 (да).

(g) Выведите текущее состояние очереди.

(h) Вызовите dequeue, выведите удаленный элемент.

(i) Выведите обновленное состояние очереди.

Пример использования:

```
queue = Queue(enqueue_divisible_by_three=True)
queue.enqueue(1)  # нет
queue.enqueue(3)  # да
queue.enqueue(4)  # нет
queue.enqueue(6)  # да
queue.enqueue(7)  # нет
```

```
queue.enqueue(9)  # да

print("Current Queue:", queue._suitcase)  # [3,6,9]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 3

print("Updated Queue:", queue._suitcase)  # [6,9]
```

34. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _luggage). Принимает параметр enqueue_greater_than_prev=False. Если True, то элемент добавляется только если он строго больше предыдущего добавленного элемента (первый — всегда).

    (b) Создайте метод enqueue, который добавляет элемент, только если enqueue_greater_than_prev=False или очередь пуста или element > последний_элемент.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Пусто").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_greater_than_prev=True.

    (f) Добавьте элементы: 5, 3 (не добавится), 7, 6 (не добавится), 10, 8 (не добавится).

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_greater_than_prev=True)
queue.enqueue(5)
queue.enqueue(3)  # нет
queue.enqueue(7)
queue.enqueue(6)  # нет
queue.enqueue(10)
queue.enqueue(8)  # нет

print("Current Queue:", queue._luggage)  # [5,7,10]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 5

print("Updated Queue:", queue._luggage)  # [7,10]
```

35. Написать программу на Python, которая создает класс Queue для представления структуры данных очереди с инкапсуляцией. Класс должен содержать методы enqueue, dequeue и is_empty, которые реализуют операции добавления элементов в очередь, удаления элементов из очереди и проверки пустоты очереди соответственно. Программа также должна создавать экземпляр класса Queue, добавлять элементы в очередь, удалять элементы из очереди и выводить информацию о состоянии очереди на экран.

    Инструкции:

    (a) Создайте класс Queue с методом __init__, который инициализирует пустую очередь (список _trunk). Принимает параметр enqueue_less_than_prev=False. Если True, то элемент добавляется только если он строго меньше предыдущего добавленного элемента (первый — всегда).

    (b) Создайте метод enqueue, который добавляет элемент, только если enqueue_less_than_prev=False или очередь пуста или element < последний_элемент.

    (c) Создайте метод dequeue, который удаляет и возвращает первый элемент. Если очередь пуста — выбрасывает IndexError("Очередь пуста").

    (d) Создайте метод is_empty, который возвращает True, если очередь пуста, и False в противном случае.

    (e) Создайте экземпляр класса Queue с enqueue_less_than_prev=True.

    (f) Добавьте элементы: 10, 15 (не добавится), 8, 9 (не добавится), 5, 7 (не добавится).

    (g) Выведите текущее состояние очереди.

    (h) Вызовите dequeue, выведите удаленный элемент.

    (i) Выведите обновленное состояние очереди.

    Пример использования:

```
queue = Queue(enqueue_less_than_prev=True)
queue.enqueue(10)
queue.enqueue(15)  # нет
queue.enqueue(8)
queue.enqueue(9)   # нет
queue.enqueue(5)
queue.enqueue(7)   # нет

print("Current Queue:", queue._trunk)  # [10,8,5]

dequeued_item = queue.dequeue()
print("Dequeued item:", dequeued_item)  # 10

print("Updated Queue:", queue._trunk)  # [8,5]
```

2.4 Семинар «Структуры данных (закрепление) и __new__» (2 часа)

При создании подкласса неизменяемого встроенного типа данных (например, `float`, `str`, `int`) возникает проблема: значение объекта устанавливается *в момент его создания*, и метод `__init__` вызывается уже *после* этого, когда изменить базовое значение невозможно.

Кроме того, конструктор родительского неизменяемого типа (например, `float.__new__()`) часто не принимает дополнительные аргументы так же гибко, как `object.__new__()`, что приводит к ошибкам.

Решение: Использовать метод `__new__` для инициализации объекта *в момент его создания*.

Листинг 63: Пример: Класс Distance с использованием __new__

```
class Distance(float):
    def __new__(cls, value, unit):
        # 1. Создаем новый экземпляр float с заданным значением
        instance = super().__new__(cls, value)
        # 2. Настраиваем экземпляр, добавляя изменяемый атрибут
        instance.unit = unit
        # 3. Возвращаем настроенный экземпляр
        return instance

# Использование:
d = Distance(10.5, "km")
print(d)      # 10.5
print(d.unit) # km
d.unit = "m"  # Атрибут unit изменяем!
print(d.unit) # m
```

В этом примере `__new__` выполняет три шага:

1. Создает новый экземпляр текущего класса `cls`, вызывая `super().__new__(cls, value)`. Это обращение к `float.__new__()`, который создает и инициализирует новый экземпляр `float`.

2. Настраивает новый экземпляр, добавляя к нему изменяемый атрибут `unit`.

3. Возвращает новый, настроенный экземпляр.

Теперь класс `Distance` работает корректно, позволяя хранить единицы измерения в изменяемом атрибуте `unit`.

Замечение: для упрщения мы не применяли свойство ООП *инкапсуляция* в примере.

2.4.1 Задача 1 (Singleton)

Реализуйте задание согласно своему варианту. Обратите внимание, что мы не реализуем логику работы сложных вещей, а только её имитируем во всех вариантах.

Замечание: Singleton – это антипаттерн, в production его использовать не стоит, но для учебных целей он хорош и, кроме того, знание его сущности обязательно для разработчика.

1. Написать программу на Python, которая создает класс ‘DataBase‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton (один экземпляр). Программа должна принимать параметры при создании и выводить сообщение при подключении.

   Инструкции:

   (a) Создайте класс ‘DataBase‘.

(b) Добавьте приватный атрибут класса ‘_instance‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он проверял, существует ли уже экземпляр. Если нет — создает новый с помощью ‘super().__new__(cls)‘ и присваивает его ‘_instance‘. Возвращает ‘_instance‘.

(d) Переопределите метод ‘__init__‘, принимающий ‘user‘, ‘psw‘, ‘port‘. Устанавливает эти атрибуты экземпляра, но только если они еще не были установлены (чтобы не перезаписывать при повторном "создании").

(e) Добавьте метод ‘connect‘, который выводит сообщение "Подключение к БД: {user}, {psw}, {port}".

(f) Добавьте метод ‘__del__‘, который выводит "Закрытие соединения с БД".

(g) Добавьте метод ‘get_data‘, который возвращает строку "Данные получены".

(h) Добавьте метод ‘set_data‘, который принимает ‘data‘ и выводит "Данные ’{data}’ записаны".

(i) Создайте два экземпляра ‘db1‘ и ‘db2‘ с разными параметрами.

(j) Вызовите ‘connect‘ для ‘db1‘, затем для ‘db2‘.

(k) Выведите ‘id(db1)‘ и ‘id(db2)‘ — они должны совпадать.

Пример использования:

```
db1 = DataBase("admin", "secret", 5432)
db2 = DataBase("user", "12345", 3306)  # Это тот же объект, что и db1!

db1.connect()
db2.connect()  # Выведет те же параметры, что и db1

print("ID db1:", id(db1))
print("ID db2:", id(db2))  # ID будут одинаковыми
```

2. Написать программу на Python, которая создает класс ‘ConnectionManager‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘host‘, ‘username‘, ‘timeout‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘ConnectionManager‘.

   (b) Добавьте приватный атрибут класса ‘_shared_instance‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал существующий экземпляр, если он есть, или создавал новый.

   (d) Переопределите метод ‘__init__‘, принимающий ‘host‘, ‘username‘, ‘timeout‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘establish‘, который выводит "Соединение установлено с {host} под пользователем {username} (таймаут: {timeout})".

   (f) Добавьте метод ‘__del__‘, который выводит "Соединение разорвано".

   (g) Добавьте метод ‘fetch‘, который возвращает "Запрос выполнен".

   (h) Добавьте метод ‘commit‘, который принимает ‘transaction‘ и выводит "Транзакция ’{transaction}’ зафиксирована".

(i) Создайте два экземпляра ‘cm1‘ и ‘cm2‘ с разными параметрами.

(j) Вызовите ‘establish‘ для ‘cm1‘, затем для ‘cm2‘.

(k) Выведите ‘cm1 is cm2‘ — должно быть ‘True‘.

Пример использования:

```
cm1 = ConnectionManager("localhost", "root", 30)
cm2 = ConnectionManager("remote.server", "guest", 60)

cm1.establish()
cm2.establish()  # Параметры будут от cm1

print("cm1 is cm2:", cm1 is cm2)  # True
```

3. Написать программу на Python, которая создает класс ‘ConfigLoader‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘config_file‘, ‘env‘, ‘debug‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘ConfigLoader‘.

   (b) Добавьте приватный атрибут класса ‘_instance_ref‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он обеспечивал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘config_file‘, ‘env‘, ‘debug‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘load‘, который выводит "Конфигурация загружена из ’{config_file}’ для среды ’{env}’ (debug={debug})".

   (f) Добавьте метод ‘__del__‘, который выводит "Конфигурация выгружена".

   (g) Добавьте метод ‘get_setting‘, который принимает ‘key‘ и возвращает "Значение для {key}".

   (h) Добавьте метод ‘set_setting‘, который принимает ‘key‘, ‘value‘ и выводит "Настройка {key} установлена в {value}".

   (i) Создайте два экземпляра ‘cfg1‘ и ‘cfg2‘ с разными параметрами.

   (j) Вызовите ‘load‘ для ‘cfg1‘, затем для ‘cfg2‘.

   (k) Проверьте, что ‘cfg1.debug == cfg2.debug‘ (должно быть ‘True‘, если первый был создан с ‘debug=True‘).

   Пример использования:

```
cfg1 = ConfigLoader("app.yaml", "prod", True)
cfg2 = ConfigLoader("dev.yaml", "dev", False)

cfg1.load()
cfg2.load()  # Параметры будут от cfg1

print("Debug mode (cfg1):", cfg1.debug)
print("Debug mode (cfg2):", cfg2.debug)  # Будет True, как у cfg1
```

4. Написать программу на Python, которая создает класс ‘Logger‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘log_level‘, ‘output_file‘, ‘rotate‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘Logger‘.

   (b) Добавьте приватный атрибут класса ‘_the_logger‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘log_level‘, ‘output_file‘, ‘rotate‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘log‘, который принимает ‘message‘ и выводит "[{log_level}] {message} -> {output_file}".

   (f) Добавьте метод ‘__del__‘, который выводит "Логгер остановлен".

   (g) Добавьте метод ‘set_level‘, который принимает ‘level‘ и устанавливает ‘self.log_level = level‘.

   (h) Добавьте метод ‘flush‘, который выводит "Буфер логов сброшен".

   (i) Создайте два экземпляра ‘log1‘ и ‘log2‘ с разными параметрами.

   (j) Вызовите ‘log‘ для ‘log1‘, затем ‘set_level("ERROR")‘ для ‘log2‘.

   (k) Вызовите ‘log‘ для ‘log1‘ снова — уровень должен измениться.

   Пример использования:

```
log1 = Logger("INFO", "app.log", True)
log2 = Logger("DEBUG", "debug.log", False)

log1.log("Старт приложения")
log2.set\_level("ERROR")  # Меняет уровень для log1 тоже!
log1.log("Ошибка!")  # Выведет [ERROR] Ошибка! -> app.log
```

5. Написать программу на Python, которая создает класс ‘Cache‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘max_size‘, ‘ttl‘, ‘strategy‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘Cache‘.

   (b) Добавьте приватный атрибут класса ‘_cache_instance‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он обеспечивал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘max_size‘, ‘ttl‘, ‘strategy‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘put‘, который принимает ‘key‘, ‘value‘ и выводит "Ключ ’{key}’ закеширован (стратегия: {strategy})".

   (f) Добавьте метод ‘__del__‘, который выводит "Кеш очищен".

(g) Добавьте метод ‘get‘, который принимает ‘key‘ и возвращает "Значение для {key}".

(h) Добавьте метод ‘clear‘, который выводит "Кеш принудительно очищен".

(i) Создайте два экземпляра ‘cache1‘ и ‘cache2‘ с разными параметрами.

(j) Вызовите ‘put‘ для ‘cache1‘, затем ‘clear‘ для ‘cache2‘.

(k) Проверьте, что ‘cache1.max_size == cache2.max_size‘.

Пример использования:

```
cache1 = Cache(1000, 3600, "LRU")
cache2 = Cache(500, 1800, "FIFO")

cache1.put("user\_123", \{"name": "Alice"\})
cache2.clear()  # Очищает кеш cache1 тоже

print("Max size:", cache1.max\_size)  # 1000 (от первого вызова)
```

6. Написать программу на Python, которая создает класс ‘SessionHandler‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘session_id‘, ‘timeout‘, ‘secure‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘SessionHandler‘.

   (b) Добавьте приватный атрибут класса ‘_handler‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘session_id‘, ‘timeout‘, ‘secure‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘start‘, который выводит "Сессия {session_id} начата (timeout={timeout}, secure={secure})".

   (f) Добавьте метод ‘__del__‘, который выводит "Сессия завершена".

   (g) Добавьте метод ‘get_session_data‘, который возвращает "Данные сессии".

   (h) Добавьте метод ‘invalidate‘, который выводит "Сессия аннулирована".

   (i) Создайте два экземпляра ‘sh1‘ и ‘sh2‘ с разными параметрами.

   (j) Вызовите ‘start‘ для ‘sh1‘, затем ‘invalidate‘ для ‘sh2‘.

   (k) Выведите ‘sh1.session_id‘ и ‘sh2.session_id‘ — они должны быть одинаковыми.

   Пример использования:

```
sh1 = SessionHandler("SID-001", 1800, True)
sh2 = SessionHandler("SID-999", 600, False)

sh1.start()
sh2.invalidate()  # Аннулирует сессию sh1

print("Session ID sh1:", sh1.session\_id)  # SID-001
print("Session ID sh2:", sh2.session\_id)  # SID-001
```

7. Написать программу на Python, которая создает класс ‘ResourceManager‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘resource_type‘, ‘capacity‘, ‘priority‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘ResourceManager‘.

   (b) Добавьте приватный атрибут класса ‘_manager‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘resource_type‘, ‘capacity‘, ‘priority‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘allocate‘, который выводит "Выделено {capacity} ресурсов типа {resource_type} (приоритет: {priority})".

   (f) Добавьте метод ‘__del__‘, который выводит "Освобождение ресурсов".

   (g) Добавьте метод ‘status‘, который возвращает "Ресурсы доступны".

   (h) Добавьте метод ‘release‘, который выводит "Ресурсы освобождены".

   (i) Создайте два экземпляра ‘rm1‘ и ‘rm2‘ с разными параметрами.

   (j) Вызовите ‘allocate‘ для ‘rm1‘, затем ‘release‘ для ‘rm2‘.

   (k) Проверьте, что ‘rm1.capacity == rm2.capacity‘.

   Пример использования:

```
rm1 = ResourceManager("CPU", 4, 1)
rm2 = ResourceManager("GPU", 2, 2)

rm1.allocate()
rm2.release()  # Освобождает ресурсы rm1

print("Capacity:", rm1.capacity)  # 4
```

8. Написать программу на Python, которая создает класс ‘PrinterPool‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘printer_id‘, ‘speed‘, ‘color‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘PrinterPool‘.

   (b) Добавьте приватный атрибут класса ‘_pool‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘printer_id‘, ‘speed‘, ‘color‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘print‘, который выводит "Печать документа на принтере {printer_id} (скорость: {speed}, цвет: {color})".

   (f) Добавьте метод ‘__del__‘, который выводит "Принтер выключен".

(g) Добавьте метод ‘get_status‘, который возвращает "Готов к печати".

(h) Добавьте метод ‘add_job‘, который принимает ‘job‘ и выводит "Добавлено задание: {job}".

(i) Создайте два экземпляра ‘pp1‘ и ‘pp2‘ с разными параметрами.

(j) Вызовите ‘print‘ для ‘pp1‘, затем ‘add_job‘ для ‘pp2‘.

(k) Проверьте, что ‘pp1.speed == pp2.speed‘.

Пример использования:

```
pp1 = PrinterPool("P100", 10, "Yes")
pp2 = PrinterPool("P200", 8, "No")

pp1.print()
pp2.add\_job("Report.pdf")  # Добавляет задание для pp1

print("Speed:", pp1.speed)  # 10
```

9. Написать программу на Python, которая создает класс ‘NetworkInterface‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘interface_name‘, ‘ip‘, ‘mac‘ при создании экземпляра.

   Инструкции:

   (a) Создайте класс ‘NetworkInterface‘.

   (b) Добавьте приватный атрибут класса ‘_interface‘ и инициализируйте его значением ‘None‘.

   (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

   (d) Переопределите метод ‘__init__‘, принимающий ‘interface_name‘, ‘ip‘, ‘mac‘. Устанавливает атрибуты, только если они еще не заданы.

   (e) Добавьте метод ‘connect‘, который выводит "Подключение к сети через {interface_name} ({ip}, {mac})".

   (f) Добавьте метод ‘__del__‘, который выводит "Отключение от сети".

   (g) Добавьте метод ‘ping‘, который возвращает "Пинг успешен".

   (h) Добавьте метод ‘configure‘, который принимает ‘new_ip‘ и выводит "IP изменен на {new_ip}".

   (i) Создайте два экземпляра ‘ni1‘ и ‘ni2‘ с разными параметрами.

   (j) Вызовите ‘connect‘ для ‘ni1‘, затем ‘configure‘ для ‘ni2‘.

   (k) Проверьте, что ‘ni1.ip == ni2.ip‘.

   Пример использования:

```
ni1 = NetworkInterface("eth0", "192.168.1.100", "AA:BB:CC:DD:EE:FF")
ni2 = NetworkInterface("wlan0", "192.168.1.101", "11:22:33:44:55:66")

ni1.connect()
ni2.configure("192.168.1.102")  # Изменяет IP для ni1

print("IP:", ni1.ip)  # 192.168.1.102
```

10. Написать программу на Python, которая создает класс ‘FileManager‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘path‘, ‘mode‘, ‘buffered‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘FileManager‘.

    (b) Добавьте приватный атрибут класса ‘_manager‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘path‘, ‘mode‘, ‘buffered‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘open‘, который выводит "Открытие файла ’{path}’ в режиме ’{mode}’ (буферизация: {buffered})".

    (f) Добавьте метод ‘__del__‘, который выводит "Файл закрыт".

    (g) Добавьте метод ‘read‘, который возвращает "Данные прочитаны".

    (h) Добавьте метод ‘write‘, который принимает ‘data‘ и выводит "Данные ’{data}’ записаны".

    (i) Создайте два экземпляра ‘fm1‘ и ‘fm2‘ с разными параметрами.

    (j) Вызовите ‘open‘ для ‘fm1‘, затем ‘write‘ для ‘fm2‘.

    (k) Проверьте, что ‘fm1.mode == fm2.mode‘.

    Пример использования:

```
fm1 = FileManager("data.txt", "r", True)
fm2 = FileManager("log.txt", "w", False)

fm1.open()
fm2.write("Hello")  # Записывает в файл fm1

print("Mode:", fm1.mode)  # r
```

11. Написать программу на Python, которая создает класс ‘DatabaseConnector‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘dbname‘, ‘host‘, ‘port‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘DatabaseConnector‘.

    (b) Добавьте приватный атрибут класса ‘_connector‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘dbname‘, ‘host‘, ‘port‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘connect‘, который выводит "Подключение к базе данных ’{dbname}’ на {host}:{port}".

    (f) Добавьте метод ‘__del__‘, который выводит "Отключение от базы данных".

(g) Добавьте метод ‘query‘, который возвращает "Запрос выполнен".

(h) Добавьте метод ‘disconnect‘, который выводит "Разрыв соединения".

(i) Создайте два экземпляра ‘dc1‘ и ‘dc2‘ с разными параметрами.

(j) Вызовите ‘connect‘ для ‘dc1‘, затем ‘disconnect‘ для ‘dc2‘.

(k) Проверьте, что ‘dc1.port == dc2.port‘.

Пример использования:

```
dc1 = DatabaseConnector("users", "localhost", 5432)
dc2 = DatabaseConnector("products", "db.example.com", 5432)

dc1.connect()
dc2.disconnect()  # Разрывает соединение dc1

print("Port:", dc1.port)  # 5432
```

12. Написать программу на Python, которая создает класс ‘MessageQueue‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘queue_name‘, ‘max_messages‘, ‘timeout‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘MessageQueue‘.

(b) Добавьте приватный атрибут класса ‘_queue‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘queue_name‘, ‘max_messages‘, ‘timeout‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘send‘, который принимает ‘message‘ и выводит "Отправка сообщения ’{message}’ в очередь {queue_name}".

(f) Добавьте метод ‘__del__‘, который выводит "Очередь закрыта".

(g) Добавьте метод ‘receive‘, который возвращает "Сообщение получено".

(h) Добавьте метод ‘clear‘, который выводит "Очередь очищена".

(i) Создайте два экземпляра ‘mq1‘ и ‘mq2‘ с разными параметрами.

(j) Вызовите ‘send‘ для ‘mq1‘, затем ‘clear‘ для ‘mq2‘.

(k) Проверьте, что ‘mq1.max_messages == mq2.max_messages‘.

Пример использования:

```
mq1 = MessageQueue("orders", 100, 30)
mq2 = MessageQueue("notifications", 50, 60)

mq1.send("New order")
mq2.clear()  # Очищает очередь mq1

print("Max messages:", mq1.max\_messages)  # 100
```

13. Написать программу на Python, которая создает класс ‘StorageDevice‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘device_id‘, ‘capacity‘, ‘type‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘StorageDevice‘.

    (b) Добавьте приватный атрибут класса ‘_device‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘device_id‘, ‘capacity‘, ‘type‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘mount‘, который выводит "Подключение устройства {device_id} (тип: {type}, емкость: {capacity})".

    (f) Добавьте метод ‘__del__‘, который выводит "Отключение устройства".

    (g) Добавьте метод ‘read‘, который возвращает "Чтение данных".

    (h) Добавьте метод ‘write‘, который принимает ‘data‘ и выводит "Запись данных ’{data}’".

    (i) Создайте два экземпляра ‘sd1‘ и ‘sd2‘ с разными параметрами.

    (j) Вызовите ‘mount‘ для ‘sd1‘, затем ‘write‘ для ‘sd2‘.

    (k) Проверьте, что ‘sd1.capacity == sd2.capacity‘.

    Пример использования:

```
sd1 = StorageDevice("SSD-001", 512, "SSD")
sd2 = StorageDevice("HDD-002", 1024, "HDD")

sd1.mount()
sd2.write("File.txt")  # Записывает в устройство sd1

print("Capacity:", sd1.capacity)  # 512
```

14. Написать программу на Python, которая создает класс ‘APIGateway‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘api_url‘, ‘token‘, ‘version‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘APIGateway‘.

    (b) Добавьте приватный атрибут класса ‘_gateway‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘api_url‘, ‘token‘, ‘version‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘call‘, который принимает ‘endpoint‘ и выводит "Вызов API {endpoint} на {api_url} (версия: {version})".

    (f) Добавьте метод ‘__del__‘, который выводит "API шлюз отключен".

(g) Добавьте метод ‘get‘, который возвращает "Данные получены".

(h) Добавьте метод ‘post‘, который принимает ‘data‘ и выводит "Отправлено: {data}".

(i) Создайте два экземпляра ‘ag1‘ и ‘ag2‘ с разными параметрами.

(j) Вызовите ‘call‘ для ‘ag1‘, затем ‘post‘ для ‘ag2‘.

(k) Проверьте, что ‘ag1.version == ag2.version‘.

Пример использования:

```
ag1 = APIGateway("https://api.example.com", "abc123", "v1")
ag2 = APIGateway("https://api.test.com", "def456", "v2")

ag1.call("/users")
ag2.post("Hello")  # Отправляет данные через ag1

print("Version:", ag1.version)  # v2
```

15. Написать программу на Python, которая создает класс ‘TaskScheduler‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘scheduler_id‘, ‘interval‘, ‘enabled‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘TaskScheduler‘.

(b) Добавьте приватный атрибут класса ‘_scheduler‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘scheduler_id‘, ‘interval‘, ‘enabled‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘start‘, который выводит "Запуск планировщика {scheduler_id} (интервал: {interval}, включен: {enabled})".

(f) Добавьте метод ‘__del__‘, который выводит "Планировщик остановлен".

(g) Добавьте метод ‘schedule‘, который принимает ‘task‘ и выводит "Запланирована задача: {task}".

(h) Добавьте метод ‘stop‘, который выводит "Остановка планировщика".

(i) Создайте два экземпляра ‘ts1‘ и ‘ts2‘ с разными параметрами.

(j) Вызовите ‘start‘ для ‘ts1‘, затем ‘schedule‘ для ‘ts2‘.

(k) Проверьте, что ‘ts1.interval == ts2.interval‘.

Пример использования:

```
ts1 = TaskScheduler("daily", 3600, True)
ts2 = TaskScheduler("hourly", 300, False)

ts1.start()
ts2.schedule("Backup")  # Запланирована задача для ts1

print("Interval:", ts1.interval)  # 3600
```

16. Написать программу на Python, которая создает класс ‘ServiceMonitor‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘service_name‘, ‘check_interval‘, ‘threshold‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘ServiceMonitor‘.

    (b) Добавьте приватный атрибут класса ‘_monitor‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘service_name‘, ‘check_interval‘, ‘threshold‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘start‘, который выводит "Мониторинг службы {service_name} запущен (интервал: {check_interval}, порог: {threshold})".

    (f) Добавьте метод ‘__del__‘, который выводит "Мониторинг остановлен".

    (g) Добавьте метод ‘check‘, который возвращает "Проверка завершена".

    (h) Добавьте метод ‘alert‘, который принимает ‘message‘ и выводит "Алерт: {message}".

    (i) Создайте два экземпляра ‘sm1‘ и ‘sm2‘ с разными параметрами.

    (j) Вызовите ‘start‘ для ‘sm1‘, затем ‘alert‘ для ‘sm2‘.

    (k) Проверьте, что ‘sm1.check_interval == sm2.check_interval‘.

    Пример использования:

```
sm1 = ServiceMonitor("web", 60, 0.9)
sm2 = ServiceMonitor("db", 30, 0.8)

sm1.start()
sm2.alert("High load")  # Алерт для sm1

print("Check interval:", sm1.check\_interval)  # 60
```

17. Написать программу на Python, которая создает класс ‘EventBus‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘bus_id‘, ‘topic‘, ‘max_listeners‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘EventBus‘.

    (b) Добавьте приватный атрибут класса ‘_bus‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘bus_id‘, ‘topic‘, ‘max_listeners‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘publish‘, который принимает ‘event‘ и выводит "Опубликовано событие ’{event}’ в топик {topic}".

    (f) Добавьте метод ‘__del__‘, который выводит "Шина событий закрыта".

(g) Добавьте метод ‘subscribe‘, который принимает ‘listener‘ и выводит "Подписан слушатель: {listener}".

(h) Добавьте метод ‘unsubscribe‘, который принимает ‘listener‘ и выводит "Отписка слушателя: {listener}".

(i) Создайте два экземпляра ‘eb1‘ и ‘eb2‘ с разными параметрами.

(j) Вызовите ‘publish‘ для ‘eb1‘, затем ‘subscribe‘ для ‘eb2‘.

(k) Проверьте, что ‘eb1.max_listeners == eb2.max_listeners‘.

Пример использования:

```
eb1 = EventBus("main", "system", 10)
eb2 = EventBus("backup", "alerts", 5)

eb1.publish("Start")
eb2.subscribe("User")  # Подписка на eb1

print("Max listeners:", eb1.max\_listeners)  # 10
```

18. Написать программу на Python, которая создает класс ‘SignalProcessor‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘processor_id‘, ‘sample_rate‘, ‘filter_type‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘SignalProcessor‘.

(b) Добавьте приватный атрибут класса ‘_processor‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘processor_id‘, ‘sample_rate‘, ‘filter_type‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘process‘, который принимает ‘signal‘ и выводит "Обработка сигнала ’{signal}’ (частота: {sample_rate}, фильтр: {filter_type})".

(f) Добавьте метод ‘__del__‘, который выводит "Процессор сигналов остановлен".

(g) Добавьте метод ‘analyze‘, который возвращает "Анализ завершен".

(h) Добавьте метод ‘apply_filter‘, который принимает ‘filter_params‘ и выводит "Применён фильтр с параметрами: {filter_params}".

(i) Создайте два экземпляра ‘sp1‘ и ‘sp2‘ с разными параметрами.

(j) Вызовите ‘process‘ для ‘sp1‘, затем ‘apply_filter‘ для ‘sp2‘.

(k) Проверьте, что ‘sp1.sample_rate == sp2.sample_rate‘.

Пример использования:

```
sp1 = SignalProcessor("audio", 44100, "lowpass")
sp2 = SignalProcessor("video", 30000, "bandpass")

sp1.process("sound")
sp2.apply\_filter(\{"cutoff": 1000\})  # Применяет фильтр для sp1

print("Sample rate:", sp1.sample\_rate)  # 44100
```

19. Написать программу на Python, которая создает класс ‘DataPipeline‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘pipeline_id‘, ‘source‘, ‘destination‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘DataPipeline‘.

    (b) Добавьте приватный атрибут класса ‘_pipeline‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘pipeline_id‘, ‘source‘, ‘destination‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘start‘, который выводит "Запуск потока данных {pipeline_id} ({source} → {destination})".

    (f) Добавьте метод ‘__del__‘, который выводит "Поток данных остановлен".

    (g) Добавьте метод ‘transform‘, который возвращает "Трансформация завершена".

    (h) Добавьте метод ‘transfer‘, который принимает ‘data‘ и выводит "Передача данных ’{data}’".

    (i) Создайте два экземпляра ‘dp1‘ и ‘dp2‘ с разными параметрами.

    (j) Вызовите ‘start‘ для ‘dp1‘, затем ‘transfer‘ для ‘dp2‘.

    (k) Проверьте, что ‘dp1.destination == dp2.destination‘.

    Пример использования:

```
dp1 = DataPipeline("etl", "db", "cloud")
dp2 = DataPipeline("backup", "local", "cloud")

dp1.start()
dp2.transfer("records")  # Передача данных через dp1

print("Destination:", dp1.destination)  # cloud
```

20. Написать программу на Python, которая создает класс ‘SecurityGuard‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘guard_id‘, ‘access_level‘, ‘rules‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘SecurityGuard‘.

    (b) Добавьте приватный атрибут класса ‘_guard‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘guard_id‘, ‘access_level‘, ‘rules‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘authorize‘, который принимает ‘request‘ и выводит "Авторизация запроса ’{request}’ (уровень: {access_level})".

(f) Добавьте метод ‘__del__‘, который выводит "Система безопасности отключена".

(g) Добавьте метод ‘audit‘, который возвращает "Аудит завершен".

(h) Добавьте метод ‘block‘, который принимает ‘entity‘ и выводит "Блокировка сущности: {entity}".

(i) Создайте два экземпляра ‘sg1‘ и ‘sg2‘ с разными параметрами.

(j) Вызовите ‘authorize‘ для ‘sg1‘, затем ‘block‘ для ‘sg2‘.

(k) Проверьте, что ‘sg1.access_level == sg2.access_level‘.

Пример использования:

```
sg1 = SecurityGuard("main", "admin", ["rule1"])
sg2 = SecurityGuard("backup", "user", ["rule2"])

sg1.authorize("login")
sg2.block("malware")  # Блокировка для sg1

print("Access level:", sg1.access\_level)  # admin
```

21. Написать программу на Python, которая создает класс ‘SystemTray‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘tray_id‘, ‘icon‘, ‘tooltip‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘SystemTray‘.

(b) Добавьте приватный атрибут класса ‘_tray‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘tray_id‘, ‘icon‘, ‘tooltip‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘show‘, который выводит "Показать значок {icon} в трее (подсказка: {tooltip})".

(f) Добавьте метод ‘__del__‘, который выводит "Значок скрыт".

(g) Добавьте метод ‘hide‘, который выводит "Скрыть значок".

(h) Добавьте метод ‘notify‘, который принимает ‘message‘ и выводит "Уведомление: {message}".

(i) Создайте два экземпляра ‘st1‘ и ‘st2‘ с разными параметрами.

(j) Вызовите ‘show‘ для ‘st1‘, затем ‘notify‘ для ‘st2‘.

(k) Проверьте, что ‘st1.icon == st2.icon‘.

Пример использования:

```
st1 = SystemTray("app", "app.ico", "My App")
st2 = SystemTray("tool", "tool.ico", "My Tool")

st1.show()
st2.notify("Update available")  # Уведомление для st1

print("Icon:", st1.icon)  # app.ico
```

22. Написать программу на Python, которая создает класс ‘ApplicationLauncher‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘launcher_id‘, ‘apps‘, ‘auto_start‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘ApplicationLauncher‘.

    (b) Добавьте приватный атрибут класса ‘_launcher‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘launcher_id‘, ‘apps‘, ‘auto_start‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘launch‘, который принимает ‘app‘ и выводит "Запуск приложения ’{app}’ (автозапуск: {auto_start})".

    (f) Добавьте метод ‘__del__‘, который выводит "Запускатор остановлен".

    (g) Добавьте метод ‘list_apps‘, который возвращает "Список приложений: {apps}".

    (h) Добавьте метод ‘add_app‘, который принимает ‘app‘ и выводит "Добавлено приложение: {app}".

    (i) Создайте два экземпляра ‘al1‘ и ‘al2‘ с разными параметрами.

    (j) Вызовите ‘launch‘ для ‘al1‘, затем ‘add_app‘ для ‘al2‘.

    (k) Проверьте, что ‘al1.apps == al2.apps‘.

    Пример использования:

```
al1 = ApplicationLauncher("main", ["browser", "editor"], True)
al2 = ApplicationLauncher("backup", ["backup", "sync"], False)

al1.launch("browser")
al2.add\_app("calc")  # Добавляет приложение в al1

print("Apps:", al1.apps)  # ['browser', 'editor', 'calc']
```

23. Написать программу на Python, которая создает класс ‘NetworkScanner‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘scanner_id‘, ‘target‘, ‘timeout‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘NetworkScanner‘.

    (b) Добавьте приватный атрибут класса ‘_scanner‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘scanner_id‘, ‘target‘, ‘timeout‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘scan‘, который выводит "Сканирование сети {target} (таймаут: {timeout})".

    (f) Добавьте метод ‘__del__‘, который выводит "Сканер остановлен".

(g) Добавьте метод ‘report‘, который возвращает "Отчет о сканировании".

(h) Добавьте метод ‘ping‘, который принимает ‘host‘ и выводит "Проверка доступности {host}".

(i) Создайте два экземпляра ‘ns1‘ и ‘ns2‘ с разными параметрами.

(j) Вызовите ‘scan‘ для ‘ns1‘, затем ‘ping‘ для ‘ns2‘.

(k) Проверьте, что ‘ns1.timeout == ns2.timeout‘.

Пример использования:

```
ns1 = NetworkScanner("fast", "192.168.1.0/24", 1)
ns2 = NetworkScanner("slow", "10.0.0.0/8", 5)

ns1.scan()
ns2.ping("192.168.1.1")  # Проверка для ns1

print("Timeout:", ns1.timeout)  # 1
```

24. Написать программу на Python, которая создает класс ‘HealthChecker‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘checker_id‘, ‘services‘, ‘frequency‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘HealthChecker‘.

(b) Добавьте приватный атрибут класса ‘_checker‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘checker_id‘, ‘services‘, ‘frequency‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘check‘, который выводит "Проверка состояния сервисов {services} (частота: {frequency})".

(f) Добавьте метод ‘__del__‘, который выводит "Проверка здоровья остановлена".

(g) Добавьте метод ‘status‘, который возвращает "Состояние: OK".

(h) Добавьте метод ‘alert‘, который принимает ‘service‘ и выводит "Алерт: {service} не отвечает".

(i) Создайте два экземпляра ‘h1‘ и ‘h2‘ с разными параметрами.

(j) Вызовите ‘check‘ для ‘h1‘, затем ‘alert‘ для ‘h2‘.

(k) Проверьте, что ‘h1.frequency == h2.frequency‘.

Пример использования:

```
h1 = HealthChecker("main", ["web", "db"], 60)
h2 = HealthChecker("backup", ["cache", "redis"], 30)

h1.check()
h2.alert("redis")  # Алерт для h1

print("Frequency:", h1.frequency)  # 60
```

25. Написать программу на Python, которая создает класс ‘PerformanceMonitor‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘monitor_id‘, ‘metrics‘, ‘interval‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘PerformanceMonitor‘.

    (b) Добавьте приватный атрибут класса ‘_monitor‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘monitor_id‘, ‘metrics‘, ‘interval‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘start‘, который выводит "Мониторинг производительности {monitor_id} запущен (метрики: {metrics}, интервал: {interval})".

    (f) Добавьте метод ‘__del__‘, который выводит "Мониторинг остановлен".

    (g) Добавьте метод ‘collect‘, который возвращает "Сбор метрик завершен".

    (h) Добавьте метод ‘report‘, который принимает ‘data‘ и выводит "Отчет: {data}".

    (i) Создайте два экземпляра ‘pm1‘ и ‘pm2‘ с разными параметрами.

    (j) Вызовите ‘start‘ для ‘pm1‘, затем ‘report‘ для ‘pm2‘.

    (k) Проверьте, что ‘pm1.interval == pm2.interval‘.

    Пример использования:

```
pm1 = PerformanceMonitor("cpu", ["usage", "temp"], 10)
pm2 = PerformanceMonitor("memory", ["ram", "swap"], 5)

pm1.start()
pm2.report("High CPU load")  # Отчет для pm1

print("Interval:", pm1.interval)  # 10
```

26. Написать программу на Python, которая создает класс ‘LogAggregator‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘aggregator_id‘, ‘sources‘, ‘format‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘LogAggregator‘.

    (b) Добавьте приватный атрибут класса ‘_aggregator‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘aggregator_id‘, ‘sources‘, ‘format‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘aggregate‘, который выводит "Агрегация логов из {sources} (формат: {format})".

    (f) Добавьте метод ‘__del__‘, который выводит "Агрегатор остановлен".

(g) Добавьте метод ‘forward‘, который принимает ‘logs‘ и выводит "Передача логов: {logs}".

(h) Добавьте метод ‘filter‘, который принимает ‘criteria‘ и выводит "Фильтрация по критерию: {criteria}".

(i) Создайте два экземпляра ‘la1‘ и ‘la2‘ с разными параметрами.

(j) Вызовите ‘aggregate‘ для ‘la1‘, затем ‘forward‘ для ‘la2‘.

(k) Проверьте, что ‘la1.format == la2.format‘.

Пример использования:

```
la1 = LogAggregator("main", ["app", "db"], "json")
la2 = LogAggregator("backup", ["web", "api"], "text")

la1.aggregate()
la2.forward("error logs")  # Передача для la1

print("Format:", la1.format)  # json
```

27. Написать программу на Python, которая создает класс ‘ResourceTracker‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘tracker_id‘, ‘resources‘, ‘threshold‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘ResourceTracker‘.

(b) Добавьте приватный атрибут класса ‘_tracker‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘tracker_id‘, ‘resources‘, ‘threshold‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘track‘, который выводит "Отслеживание ресурсов {resources} (порог: {threshold})".

(f) Добавьте метод ‘__del__‘, который выводит "Отслеживание остановлено".

(g) Добавьте метод ‘update‘, который принимает ‘data‘ и выводит "Обновление данных: {data}".

(h) Добавьте метод ‘alarm‘, который принимает ‘resource‘ и выводит "Авария: {resource} исчерпан".

(i) Создайте два экземпляра ‘rt1‘ и ‘rt2‘ с разными параметрами.

(j) Вызовите ‘track‘ для ‘rt1‘, затем ‘alarm‘ для ‘rt2‘.

(k) Проверьте, что ‘rt1.threshold == rt2.threshold‘.

Пример использования:

```
rt1 = ResourceTracker("cpu", ["cores", "freq"], 0.8)
rt2 = ResourceTracker("memory", ["ram", "swap"], 0.9)

rt1.track()
rt2.alarm("ram")  # Авария для rt1

print("Threshold:", rt1.threshold)  # 0.8
```

28. Написать программу на Python, которая создает класс ‘NotificationCenter‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘center_id‘, ‘channels‘, ‘priority‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘NotificationCenter‘.

    (b) Добавьте приватный атрибут класса ‘_center‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘center_id‘, ‘channels‘, ‘priority‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘notify‘, который принимает ‘message‘ и выводит "Уведомление: {message} (каналы: {channels}, приоритет: {priority})".

    (f) Добавьте метод ‘__del__‘, который выводит "Центр уведомлений остановлен".

    (g) Добавьте метод ‘subscribe‘, который принимает ‘channel‘ и выводит "Подписан на канал: {channel}".

    (h) Добавьте метод ‘unsubcribe‘, который принимает ‘channel‘ и выводит "Отписка от канала: {channel}".

    (i) Создайте два экземпляра ‘nc1‘ и ‘nc2‘ с разными параметрами.

    (j) Вызовите ‘notify‘ для ‘nc1‘, затем ‘subscribe‘ для ‘nc2‘.

    (k) Проверьте, что ‘nc1.priority == nc2.priority‘.

    Пример использования:

```
nc1 = NotificationCenter("main", ["email", "push"], 1)
nc2 = NotificationCenter("backup", ["sms", "phone"], 2)

nc1.notify("Update ready")
nc2.subscribe("email")  # Подписка на nc1

print("Priority:", nc1.priority)  # 1
```

29. Написать программу на Python, которая создает класс ‘ConfigurationManager‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘manager_id‘, ‘config_files‘, ‘reload_on_change‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘ConfigurationManager‘.

    (b) Добавьте приватный атрибут класса ‘_manager‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘manager_id‘, ‘config_files‘, ‘reload_on_change‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘load‘, который выводит "Загрузка конфигураций из {config_files} (перезагрузка при изменении: {reload_on_change})".

(f) Добавьте метод ‘__del__‘, который выводит "Конфигурации выгружены".

(g) Добавьте метод ‘get‘, который принимает ‘key‘ и возвращает "Значение для {key}".

(h) Добавьте метод ‘set‘, который принимает ‘key‘, ‘value‘ и выводит "Настройка {key} установлена в {value}".

(i) Создайте два экземпляра ‘cm1‘ и ‘cm2‘ с разными параметрами.

(j) Вызовите ‘load‘ для ‘cm1‘, затем ‘set‘ для ‘cm2‘.

(k) Проверьте, что ‘cm1.reload_on_change == cm2.reload_on_change‘.

Пример использования:

```
cm1 = ConfigurationManager("app", ["config.yaml"], True)
cm2 = ConfigurationManager("test", ["test.conf"], False)

cm1.load()
cm2.set("debug", True)  # Установка для cm1

print("Reload on change:", cm1.reload\_on\_change)  # True
```

30. Написать программу на Python, которая создает класс ‘JobScheduler‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘scheduler_id‘, ‘jobs‘, ‘time_zone‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘JobScheduler‘.

(b) Добавьте приватный атрибут класса ‘_scheduler‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘scheduler_id‘, ‘jobs‘, ‘time_zone‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘schedule‘, который выводит "Запланированы задания {jobs} (часовой пояс: {time_zone})".

(f) Добавьте метод ‘__del__‘, который выводит "Планировщик остановлен".

(g) Добавьте метод ‘run‘, который возвращает "Выполнение заданий".

(h) Добавьте метод ‘cancel‘, который принимает ‘job‘ и выводит "Отмена задания: {job}".

(i) Создайте два экземпляра ‘js1‘ и ‘js2‘ с разными параметрами.

(j) Вызовите ‘schedule‘ для ‘js1‘, затем ‘cancel‘ для ‘js2‘.

(k) Проверьте, что ‘js1.time_zone == js2.time_zone‘.

Пример использования:

```
js1 = JobScheduler("daily", ["backup", "cleanup"], "UTC")
js2 = JobScheduler("weekly", ["report", "archive"], "Europe/Moscow")

js1.schedule()
js2.cancel("report")  # Отмена для js1

print("Time zone:", js1.time\_zone)  # UTC
```

31. Написать программу на Python, которая создает класс ‘AnalyticsEngine‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘engine_id‘, ‘datasets‘, ‘model‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘AnalyticsEngine‘.

    (b) Добавьте приватный атрибут класса ‘_engine‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘engine_id‘, ‘datasets‘, ‘model‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘analyze‘, который выводит "Анализ данных {datasets} с моделью {model}".

    (f) Добавьте метод ‘__del__‘, который выводит "Аналитический движок остановлен".

    (g) Добавьте метод ‘train‘, который возвращает "Обучение модели завершено".

    (h) Добавьте метод ‘predict‘, который принимает ‘data‘ и выводит "Прогноз на основе данных: {data}".

    (i) Создайте два экземпляра ‘ae1‘ и ‘ae2‘ с разными параметрами.

    (j) Вызовите ‘analyze‘ для ‘ae1‘, затем ‘predict‘ для ‘ae2‘.

    (k) Проверьте, что ‘ae1.model == ae2.model‘.

    Пример использования:

```
ae1 = AnalyticsEngine("sales", ["orders", "customers"], "linear")
ae2 = AnalyticsEngine("marketing", ["ads", "clicks"], "neural")

ae1.analyze()
ae2.predict("next month")  # Прогноз для ae1

print("Model:", ae1.model)  # linear
```

32. Написать программу на Python, которая создает класс ‘AuditTrail‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘trail_id‘, ‘events‘, ‘retention‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘AuditTrail‘.

    (b) Добавьте приватный атрибут класса ‘_trail‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘trail_id‘, ‘events‘, ‘retention‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘log‘, который принимает ‘action‘ и выводит "Запись действия ’{action}’ в журнал (события: {events}, срок хранения: {retention})".

(f) Добавьте метод ‘__del__‘, который выводит "Журнал аудита закрыт".

(g) Добавьте метод ‘search‘, который принимает ‘query‘ и возвращает "Результаты поиска: {query}".

(h) Добавьте метод ‘purge‘, который выводит "Очистка журнала".

(i) Создайте два экземпляра ‘at1‘ и ‘at2‘ с разными параметрами.

(j) Вызовите ‘log‘ для ‘at1‘, затем ‘search‘ для ‘at2‘.

(k) Проверьте, что ‘at1.retention == at2.retention‘.

Пример использования:

```
at1 = AuditTrail("security", ["login", "logout"], 365)
at2 = AuditTrail("operations", ["start", "stop"], 90)

at1.log("User login")
at2.search("logout")  # Поиск в at1

print("Retention:", at1.retention)  # 365
```

33. Написать программу на Python, которая создает класс ‘ContentFilter‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘filter_id‘, ‘rules‘, ‘strict‘ при создании экземпляра.

Инструкции:

(a) Создайте класс ‘ContentFilter‘.

(b) Добавьте приватный атрибут класса ‘_filter‘ и инициализируйте его значением ‘None‘.

(c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

(d) Переопределите метод ‘__init__‘, принимающий ‘filter_id‘, ‘rules‘, ‘strict‘. Устанавливает атрибуты, только если они еще не заданы.

(e) Добавьте метод ‘filter‘, который принимает ‘content‘ и выводит "Фильтрация контента ’{content}’ (правила: {rules}, строгий режим: {strict})".

(f) Добавьте метод ‘__del__‘, который выводит "Фильтр отключен".

(g) Добавьте метод ‘get_rules‘, который возвращает "Правила: {rules}".

(h) Добавьте метод ‘add_rule‘, который принимает ‘rule‘ и выводит "Добавлено правило: {rule}".

(i) Создайте два экземпляра ‘cf1‘ и ‘cf2‘ с разными параметрами.

(j) Вызовите ‘filter‘ для ‘cf1‘, затем ‘add_rule‘ для ‘cf2‘.

(k) Проверьте, что ‘cf1.strict == cf2.strict‘.

Пример использования:

```
cf1 = ContentFilter("main", ["bad", "spam"], True)
cf2 = ContentFilter("backup", ["offensive", "inappropriate"], False)

cf1.filter("This is spam")
cf2.add\_rule("hate")  # Добавление правила для cf1

print("Strict:", cf1.strict)  # True
```

34. Написать программу на Python, которая создает класс ‘RateLimiter‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘limiter_id‘, ‘rate‘, ‘burst‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘RateLimiter‘.

    (b) Добавьте приватный атрибут класса ‘_limiter‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘limiter_id‘, ‘rate‘, ‘burst‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘limit‘, который принимает ‘request‘ и выводит "Ограничение запроса ’{request}’ (скорость: {rate}, burst: {burst})".

    (f) Добавьте метод ‘__del__‘, который выводит "Ограничитель отключен".

    (g) Добавьте метод ‘allow‘, который возвращает "Запрос разрешен".

    (h) Добавьте метод ‘deny‘, который принимает ‘reason‘ и выводит "Запрос отклонен: {reason}".

    (i) Создайте два экземпляра ‘rl1‘ и ‘rl2‘ с разными параметрами.

    (j) Вызовите ‘limit‘ для ‘rl1‘, затем ‘deny‘ для ‘rl2‘.

    (k) Проверьте, что ‘rl1.rate == rl2.rate‘.

    Пример использования:

```
rl1 = RateLimiter("api", 10, 5)
rl2 = RateLimiter("web", 5, 3)

rl1.limit("GET /users")
rl2.deny("Too many requests")  # Отклонение для rl1

print("Rate:", rl1.rate)  # 10
```

35. Написать программу на Python, которая создает класс ‘CacheManager‘ с использованием метода ‘__new__‘ для реализации паттерна Singleton. Программа должна принимать параметры ‘manager_id‘, ‘cache_size‘, ‘eviction_policy‘ при создании экземпляра.

    Инструкции:

    (a) Создайте класс ‘CacheManager‘.

    (b) Добавьте приватный атрибут класса ‘_manager‘ и инициализируйте его значением ‘None‘.

    (c) Переопределите метод ‘__new__‘, чтобы он возвращал единственный экземпляр.

    (d) Переопределите метод ‘__init__‘, принимающий ‘manager_id‘, ‘cache_size‘, ‘eviction_policy‘. Устанавливает атрибуты, только если они еще не заданы.

    (e) Добавьте метод ‘put‘, который принимает ‘key‘, ‘value‘ и выводит "Кэширование ключа ’{key}’ (размер: {cache_size}, политика: {eviction_policy})".

(f) Добавьте метод ‘__del__‘, который выводит "Кэш очищен".

(g) Добавьте метод ‘get‘, который принимает ‘key‘ и возвращает "Значение для {key}".

(h) Добавьте метод ‘evict‘, который выводит "Освобождение места в кэше".

(i) Создайте два экземпляра ‘cm1‘ и ‘cm2‘ с разными параметрами.

(j) Вызовите ‘put‘ для ‘cm1‘, затем ‘evict‘ для ‘cm2‘.

(k) Проверьте, что ‘cm1.cache_size == cm2.cache_size‘.

Пример использования:

```
cm1 = CacheManager("main", 1000, "LRU")
cm2 = CacheManager("backup", 500, "FIFO")

cm1.put("user\_123", \{"name": "Alice"\})
cm2.evict()  # Освобождение для cm1

print("Cache size:", cm1.cache\_size)  # 1000
```

2.4.2 Задача 2 (ограничение количества экземпляров)

1. Написать программу на Python, которая создает класс ‘LimitedInstances‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 5.

   Инструкции:

   (a) Создайте класс ‘LimitedInstances‘.

   (b) Добавьте атрибут класса ‘_instances‘ и инициализируйте его пустым списком.

   (c) Добавьте атрибут класса ‘_limit‘ и инициализируйте его значением 5.

   (d) Переопределите метод ‘__new__‘. Если ‘len(_instances) >= _limit‘, выбросьте ‘RuntimeError("Превышен лимит объектов: 5")‘. Иначе, создайте экземпляр с помощью ‘super().__new__(cls)‘, добавьте его в ‘_instances‘ и верните.

   (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_instances‘ при уничтожении объекта.

   (f) Переопределите метод ‘__init__‘, который принимает ‘name‘ и устанавливает ‘self.name = name‘.

   (g) Создайте 5 экземпляров класса.

   (h) Попытайтесь создать 6-й экземпляр - должно возникнуть исключение ‘RuntimeError‘.

   (i) Удалите один из первых 5 экземпляров (например, ‘del obj1‘).

   (j) Создайте 6-й экземпляр - теперь это должно сработать.

   Пример использования:

```
# Создаем 5 объектов
objs = [LimitedInstances(f"Obj{i}") for i in range(1, 6)]

# Попытка создать 6-й - вызовет ошибку
try:
    obj6 = LimitedInstances("Obj6")
except RuntimeError as e:
    print(e)
```

```
# Удаляем один объект
del objs[0]

# Теперь можно создать 6-й
obj6 = LimitedInstances("Obj6")
print("Успешно создан 6-й объект:", obj6.name)
```

2. Написать программу на Python, которая создает класс ‘BoundedObjects‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 3.

   Инструкции:

   (a) Создайте класс ‘BoundedObjects‘.

   (b) Добавьте атрибут класса ‘_pool‘ и инициализируйте его пустым списком.

   (c) Добавьте атрибут класса ‘MAX_OBJECTS‘ и инициализируйте его значением 3.

   (d) Переопределите метод ‘__new__‘. Если ‘len(_pool) >= MAX_OBJECTS‘, выбросьте ‘RuntimeError("Максимум 3 объекта!")‘. Иначе, создайте экземпляр, добавьте в ‘_pool‘, верните.

   (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_pool‘.

   (f) Переопределите метод ‘__init__‘, который принимает ‘id‘ и устанавливает ‘self.object_id = id‘.

   (g) Создайте 3 экземпляра.

   (h) Попытайтесь создать 4-й - поймайте и выведите исключение.

   (i) Удалите один экземпляр.

   (j) Создайте 4-й экземпляр - должно сработать.

   Пример использования:

```
# Создаем 3 объекта
obj1 = BoundedObjects(1)
obj2 = BoundedObjects(2)
obj3 = BoundedObjects(3)

# Попытка создать 4-й
try:
    obj4 = BoundedObjects(4)
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем один
del obj1

# Создаем 4-й - успешно
obj4 = BoundedObjects(4)
print("ID нового объекта:", obj4.object_id)
```

3. Написать программу на Python, которая создает класс ‘ResourcePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 10.

   Инструкции:

   (a) Создайте класс ‘ResourcePool‘.

(b) Добавьте атрибут класса ‘_allocated‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘CAPACITY‘ и инициализируйте его значением 10.

(d) Переопределите метод ‘__new__‘. Если ‘len(_allocated) >= CAPACITY‘, выбросьте ‘RuntimeError("Ресурсы исчерпаны!")‘. Иначе, создайте экземпляр, добавьте в ‘_allocated‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_allocated‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘resource_type‘ и устанавливает ‘self.type = resource_type‘.

(g) Создайте 10 экземпляров.

(h) Попытайтесь создать 11-й - поймайте и выведите исключение.

(i) Удалите два экземпляра.

(j) Создайте 11-й и 12-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 10 объектов
resources = [ResourcePool(f"Type{i}") for i in range(10)]

# Попытка создать 11-й
try:
    r11 = ResourcePool("Type11")
except RuntimeError as e:
    print(e)

# Удаляем два
del resources[0], resources[1]

# Создаем 11-й и 12-й - успешно
r11 = ResourcePool("Type11")
r12 = ResourcePool("Type12")
print("Созданы:", r11.type, r12.type)
```

4. Написать программу на Python, которая создает класс ‘CarFleet‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 7.

   Инструкции:

   (a) Создайте класс ‘CarFleet‘.

   (b) Добавьте атрибут класса ‘_cars‘ и инициализируйте его пустым списком.

   (c) Добавьте атрибут класса ‘FLEET_SIZE‘ и инициализируйте его значением 7.

   (d) Переопределите метод ‘__new__‘. Если ‘len(_cars) >= FLEET_SIZE‘, выбросьте ‘RuntimeError("Автопарк переполнен!")‘. Иначе, создайте экземпляр, добавьте в ‘_cars‘, верните.

   (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_cars‘.

   (f) Переопределите метод ‘__init__‘, который принимает ‘model‘ и устанавливает ‘self.model = model‘.

   (g) Создайте 7 экземпляров.

   (h) Попытайтесь создать 8-й - поймайте и выведите исключение.

   (i) Удалите три экземпляра.

(j) Создайте 8-й, 9-й и 10-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 7 машин
fleet = [CarFleet(f"Model{i}") for i in range(7)]

# Попытка создать 8-ю
try:
    car8 = CarFleet("Model8")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем три
del fleet[0], fleet[1], fleet[2]

# Создаем 8-ю, 9-ю, 10-ю - успешно
car8 = CarFleet("Model8")
car9 = CarFleet("Model9")
car10 = CarFleet("Model10")
print("Новые модели:", car8.model, car9.model, car10.model)
```

5. Написать программу на Python, которая создает класс ‘StudentGroup‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 30.

   Инструкции:

   (a) Создайте класс ‘StudentGroup‘.

   (b) Добавьте атрибут класса ‘_students‘ и инициализируйте его пустым списком.

   (c) Добавьте атрибут класса ‘GROUP_MAX‘ и инициализируйте его значением 30.

   (d) Переопределите метод ‘__new__‘. Если ‘len(_students) >= GROUP_MAX‘, выбросьте ‘RuntimeError("Группа заполнена!")‘. Иначе, создайте экземпляр, добавьте в ‘_students‘, верните.

   (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_students‘.

   (f) Переопределите метод ‘__init__‘, который принимает ‘student_name‘ и устанавливает ‘self.name = student_name‘.

   (g) Создайте 30 экземпляров.

   (h) Попытайтесь создать 31-й - поймайте и выведите исключение.

   (i) Удалите пять экземпляров.

   (j) Создайте 31-й, 32-й, 33-й, 34-й, 35-й экземпляры - должно сработать.

   Пример использования:

```
# Создаем 30 студентов
students = [StudentGroup(f"Student{i}") for i in range(30)]

# Попытка создать 31-го
try:
    s31 = StudentGroup("Alice")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем пять
for i in range(5):
```

```
    del students[0]

# Создаем 31-го, 32-го, 33-го, 34-го, 35-го - успешно
new_students = [StudentGroup(f"New{i}") for i in range(31, 36)]
for s in new_students:
    print("Добавлен:", s.name)
```

6. Написать программу на Python, которая создает класс ‘TaskQueue‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 100.

   Инструкции:

   (a) Создайте класс ‘TaskQueue‘.

   (b) Добавьте атрибут класса ‘_tasks‘ и инициализируйте его пустым списком.

   (c) Добавьте атрибут класса ‘QUEUE_LIMIT‘ и инициализируйте его значением 100.

   (d) Переопределите метод ‘__new__‘. Если ‘len(_tasks) >= QUEUE_LIMIT‘, выбросьте ‘RuntimeError("Очередь задач переполнена!")‘. Иначе, создайте экземпляр, добавьте в ‘_tasks‘, верните.

   (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_tasks‘.

   (f) Переопределите метод ‘__init__‘, который принимает ‘task_name‘ и устанавливает ‘self.task = task_name‘.

   (g) Создайте 100 экземпляров.

   (h) Попытайтесь создать 101-й - поймайте и выведите исключение.

   (i) Удалите десять экземпляров.

   (j) Создайте 101-й, 102-й, ..., 110-й экземпляры - должно сработать.

   Пример использования:

```
# Создаем 100 задач
tasks = [TaskQueue(f"Task{i}") for i in range(100)]

# Попытка создать 101-ю
try:
    t101 = TaskQueue("FinalTask")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 10 задач
for i in range(10):
    del tasks[0]

# Создаем 101-ю, 102-ю, ..., 110-ю - успешно
new_tasks = [TaskQueue(f"NewTask{i}") for i in range(101, 111)]
for t in new_tasks:
    print("Добавлена задача:", t.task)
```

7. Написать программу на Python, которая создает класс ‘ConnectionPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 8.

   Инструкции:

   (a) Создайте класс ‘ConnectionPool‘.

(b) Добавьте атрибут класса ‘_connections‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘POOL_SIZE‘ и инициализируйте его значением 8.

(d) Переопределите метод ‘__new__‘. Если ‘len(_connections) >= POOL_SIZE‘, выбросьте ‘RuntimeError("Пул соединений полон!")‘. Иначе, создайте экземпляр, добавьте в ‘_connections‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_connections‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘connection_id‘ и устанавливает ‘self.id = connection_id‘.

(g) Создайте 8 экземпляров.

(h) Попытайтесь создать 9-й - поймайте и выведите исключение.

(i) Удалите четыре экземпляра.

(j) Создайте 9-й, 10-й, 11-й, 12-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 8 соединений
pool = [ConnectionPool(f"Conn{i}") for i in range(8)]

# Попытка создать 9-е
try:
    conn9 = ConnectionPool("Conn9")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 4
del pool[0], pool[1], pool[2], pool[3]

# Создаем 9-е, 10-е, 11-е, 12-е - успешно
new_conns = [ConnectionPool(f"Conn{i}") for i in range(9, 13)]
for c in new_conns:
    print("Создано соединение:", c.id)
```

8. Написать программу на Python, которая создает класс ‘DeviceManager‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 15.

Инструкции:

(a) Создайте класс ‘DeviceManager‘.

(b) Добавьте атрибут класса ‘_devices‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘MANAGER_LIMIT‘ и инициализируйте его значением 15.

(d) Переопределите метод ‘__new__‘. Если ‘len(_devices) >= MANAGER_LIMIT‘, выбросьте ‘RuntimeError("Менеджер устройств перегружен!")‘. Иначе, создайте экземпляр, добавьте в ‘_devices‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_devices‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘device_name‘ и устанавливает ‘self.device = device_name‘.

(g) Создайте 15 экземпляров.

(h) Попытайтесь создать 16-й - поймайте и выведите исключение.

(i) Удалите семь экземпляров.

(j) Создайте 16-й, 17-й, ..., 22-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 15 устройств
devices = [DeviceManager(f"Device{i}") for i in range(15)]

# Попытка создать 16-е
try:
    d16 = DeviceManager("NewDevice")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 7
for i in range(7):
    del devices[0]

# Создаем 16-е, 17-е, ..., 22-е - успешно
new_devices = [DeviceManager(f"Device{i}") for i in range(16, 23)]
for d in new_devices:
    print("Добавлено устройство:", d.device)
```

9. Написать программу на Python, которая создает класс ‘SessionPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 6.

Инструкции:

(a) Создайте класс ‘SessionPool‘.

(b) Добавьте атрибут класса ‘_sessions‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘SESSION_LIMIT‘ и инициализируйте его значением 6.

(d) Переопределите метод ‘__new__‘. Если ‘len(_sessions) >= SESSION_LIMIT‘, выбросьте ‘RuntimeError("Пул сессий исчерпан!")‘. Иначе, создайте экземпляр, добавьте в ‘_sessions‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_sessions‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘session_token‘ и устанавливает ‘self.token = session_token‘.

(g) Создайте 6 экземпляров.

(h) Попытайтесь создать 7-й - поймайте и выведите исключение.

(i) Удалите два экземпляра.

(j) Создайте 7-й и 8-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 6 сессий
sessions = [SessionPool(f"Token{i}") for i in range(6)]

# Попытка создать 7-ю
try:
    s7 = SessionPool("Token7")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 2
```

```
del sessions[0], sessions[1]

# Создаем 7-ю и 8-ю - успешно
s7 = SessionPool("Token7")
s8 = SessionPool("Token8")
print("Созданы токены:", s7.token, s8.token)
```

10. Написать программу на Python, которая создает класс ‘ThreadPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 12.

    Инструкции:

    (a) Создайте класс ‘ThreadPool‘.

    (b) Добавьте атрибут класса ‘_threads‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘THREAD_MAX‘ и инициализируйте его значением 12.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_threads) >= THREAD_MAX‘, выбросьте ‘RuntimeError("Достигнут лимит потоков!")‘. Иначе, создайте экземпляр, добавьте в ‘_threads‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_threads‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘thread_id‘ и устанавливает ‘self.thread = thread_id‘.

    (g) Создайте 12 экземпляров.

    (h) Попытайтесь создать 13-й - поймайте и выведите исключение.

    (i) Удалите три экземпляра.

    (j) Создайте 13-й, 14-й, 15-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 12 потоков
threads = [ThreadPool(f"Thread{i}") for i in range(12)]

# Попытка создать 13-й
try:
    t13 = ThreadPool("Thread13")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 3
del threads[0], threads[1], threads[2]

# Создаем 13-й, 14-й, 15-й - успешно
t13 = ThreadPool("Thread13")
t14 = ThreadPool("Thread14")
t15 = ThreadPool("Thread15")
print("Созданы потоки:", t13.thread, t14.thread, t15.thread)
```

11. Написать программу на Python, которая создает класс ‘CachePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 20.

    Инструкции:

    (a) Создайте класс ‘CachePool‘.

    (b) Добавьте атрибут класса ‘_caches‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘CACHE_LIMIT‘ и инициализируйте его значением 20.

(d) Переопределите метод ‘__new__‘. Если ‘len(_caches) >= CACHE_LIMIT‘, выбросьте ‘RuntimeError("Кэш-пул переполнен!")‘. Иначе, создайте экземпляр, добавьте в ‘_caches‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_caches‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘cache_key‘ и устанавливает ‘self.key = cache_key‘.

(g) Создайте 20 экземпляров.

(h) Попытайтесь создать 21-й - поймайте и выведите исключение.

(i) Удалите пять экземпляров.

(j) Создайте 21-й, 22-й, ..., 25-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 20 кэшей
caches = [CachePool(f"Key{i}") for i in range(20)]

# Попытка создать 21-й
try:
    c21 = CachePool("Key21")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 5
for i in range(5):
    del caches[0]

# Создаем 21-й, 22-й, ..., 25-й - успешно
new_caches = [CachePool(f"Key{i}") for i in range(21, 26)]
for c in new_caches:
    print("Создан ключ:", c.key)
```

12. Написать программу на Python, которая создает класс ‘DatabasePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 4.

Инструкции:

(a) Создайте класс ‘DatabasePool‘.

(b) Добавьте атрибут класса ‘_databases‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘DB_LIMIT‘ и инициализируйте его значением 4.

(d) Переопределите метод ‘__new__‘. Если ‘len(_databases) >= DB_LIMIT‘, выбросьте ‘RuntimeError("Базы данных: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_databases‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_databases‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘db_name‘ и устанавливает ‘self.name = db_name‘.

(g) Создайте 4 экземпляра.

(h) Попытайтесь создать 5-й - поймайте и выведите исключение.

(i) Удалите один экземпляр.

(j) Создайте 5-й экземпляр - должно сработать.

Пример использования:

```
# Создаем 4 базы
dbs = [DatabasePool(f"DB{i}") for i in range(4)]

# Попытка создать 5-ю
try:
    db5 = DatabasePool("DB5")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем одну
del dbs[0]

# Создаем 5-ю - успешно
db5 = DatabasePool("DB5")
print("Создана база:", db5.name)
```

13. Написать программу на Python, которая создает класс ‘FileHandlerPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 9.

    Инструкции:

    (a) Создайте класс ‘FileHandlerPool‘.

    (b) Добавьте атрибут класса ‘_handlers‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘HANDLER_MAX‘ и инициализируйте его значением 9.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_handlers) >= HANDLER_MAX‘, выбросьте ‘RuntimeError("Слишком много обработчиков файлов!")‘. Иначе, создайте экземпляр, добавьте в ‘_handlers‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_handlers‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘file_path‘ и устанавливает ‘self.path = file_path‘.

    (g) Создайте 9 экземпляров.

    (h) Попытайтесь создать 10-й - поймайте и выведите исключение.

    (i) Удалите четыре экземпляра.

    (j) Создайте 10-й, 11-й, 12-й, 13-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 9 обработчиков
handlers = [FileHandlerPool(f"/path/to/file{i}.txt") for i in range(9)]

# Попытка создать 10-й
try:
    h10 = FileHandlerPool("/path/to/newfile.txt")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 4
del handlers[0], handlers[1], handlers[2], handlers[3]
```

```
# Создаем 10-й, 11-й, 12-й, 13-й - успешно
new_handlers = [FileHandlerPool(f"/path/to/newfile{i}.txt") for i in range
    (10, 14)]
for h in new_handlers:
    print("Обработчик для:", h.path)
```

14. Написать программу на Python, которая создает класс ‘NetworkPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 11.

    Инструкции:

    (a) Создайте класс ‘NetworkPool‘.

    (b) Добавьте атрибут класса ‘_networks‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘NETWORK_CAP‘ и инициализируйте его значением 11.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_networks) >= NETWORK_CAP‘, выбросьте ‘RuntimeError("Сеть: превышен лимит!")‘. Иначе, создайте экземпляр, добавьте в ‘_networks‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_networks‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘network_id‘ и устанавливает ‘self.net_id = network_id‘.

    (g) Создайте 11 экземпляров.

    (h) Попытайтесь создать 12-й - поймайте и выведите исключение.

    (i) Удалите шесть экземпляров.

    (j) Создайте 12-й, 13-й, ..., 17-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 11 сетей
networks = [NetworkPool(f"Net{i}") for i in range(11)]

# Попытка создать 12-ю
try:
    n12 = NetworkPool("Net12")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 6
for i in range(6):
    del networks[0]

# Создаем 12-ю, 13-ю, ..., 17-ю - успешно
new_networks = [NetworkPool(f"Net{i}") for i in range(12, 18)]
for n in new_networks:
    print("Создана сеть:", n.net_id)
```

15. Написать программу на Python, которая создает класс ‘MemoryPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 25.

    Инструкции:

    (a) Создайте класс ‘MemoryPool‘.

    (b) Добавьте атрибут класса ‘_blocks‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘MEMORY_LIMIT‘ и инициализируйте его значением 25.

(d) Переопределите метод ‘__new__‘. Если ‘len(_blocks) >= MEMORY_LIMIT‘, выбросьте ‘RuntimeError("Память: лимит блоков превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_blocks‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_blocks‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘block_size‘ и устанавливает ‘self.size = block_size‘.

(g) Создайте 25 экземпляров.

(h) Попытайтесь создать 26-й - поймайте и выведите исключение.

(i) Удалите десять экземпляров.

(j) Создайте 26-й, 27-й, ..., 35-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 25 блоков
blocks = [MemoryPool(f"Size{i}") for i in range(25)]

# Попытка создать 26-й
try:
    b26 = MemoryPool("Size26")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 10
for i in range(10):
    del blocks[0]

# Создаем 26-й, 27-й, ..., 35-й - успешно
new_blocks = [MemoryPool(f"Size{i}") for i in range(26, 36)]
for b in new_blocks:
    print("Создан блок размером:", b.size)
```

16. Написать программу на Python, которая создает класс ‘ProcessPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 16.

Инструкции:

(a) Создайте класс ‘ProcessPool‘.

(b) Добавьте атрибут класса ‘_processes‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘PROCESS_MAX‘ и инициализируйте его значением 16.

(d) Переопределите метод ‘__new__‘. Если ‘len(_processes) >= PROCESS_MAX‘, выбросьте ‘RuntimeError("Процессы: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_processes‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_processes‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘process_name‘ и устанавливает ‘self.name = process_name‘.

(g) Создайте 16 экземпляров.

(h) Попытайтесь создать 17-й - поймайте и выведите исключение.

(i) Удалите восемь экземпляров.

(j) Создайте 17-й, 18-й, ..., 24-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 16 процессов
processes = [ProcessPool(f"Proc{i}") for i in range(16)]

# Попытка создать 17-й
try:
    p17 = ProcessPool("Proc17")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 8
for i in range(8):
    del processes[0]

# Создаем 17-й, 18-й, ..., 24-й - успешно
new_processes = [ProcessPool(f"Proc{i}") for i in range(17, 25)]
for p in new_processes:
    print("Запущен процесс:", p.name)
```

17. Написать программу на Python, которая создает класс ‘BufferPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 18.

    Инструкции:

    (a) Создайте класс ‘BufferPool‘.

    (b) Добавьте атрибут класса ‘_buffers‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘BUFFER_SIZE‘ и инициализируйте его значением 18.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_buffers) >= BUFFER_SIZE‘, выбросьте ‘RuntimeError("Буфер: переполнение!")‘. Иначе, создайте экземпляр, добавьте в ‘_buffers‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_buffers‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘buffer_id‘ и устанавливает ‘self.id = buffer_id‘.

    (g) Создайте 18 экземпляров.

    (h) Попытайтесь создать 19-й - поймайте и выведите исключение.

    (i) Удалите девять экземпляров.

    (j) Создайте 19-й, 20-й, ..., 27-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 18 буферов
buffers = [BufferPool(f"Buf{i}") for i in range(18)]

# Попытка создать 19-й
try:
    b19 = BufferPool("Buf19")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 9
for i in range(9):
    del buffers[0]
```

```
# Создаем 19-й, 20-й, ..., 27-й - успешно
new_buffers = [BufferPool(f"Buf{i}") for i in range(19, 28)]
for b in new_buffers:
    print("Создан буфер:", b.id)
```

18. Написать программу на Python, которая создает класс ‘ChannelPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 13.

    Инструкции:

    (a) Создайте класс ‘ChannelPool‘.

    (b) Добавьте атрибут класса ‘_channels‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘CHANNEL_LIMIT‘ и инициализируйте его значением 13.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_channels) >= CHANNEL_LIMIT‘, выбросьте ‘RuntimeError("Каналы: лимит исчерпан!")‘. Иначе, создайте экземпляр, добавьте в ‘_channels‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_channels‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘channel_name‘ и устанавливает ‘self.name = channel_name‘.

    (g) Создайте 13 экземпляров.

    (h) Попытайтесь создать 14-й - поймайте и выведите исключение.

    (i) Удалите три экземпляра.

    (j) Создайте 14-й, 15-й, 16-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 13 каналов
channels = [ChannelPool(f"Channel{i}") for i in range(13)]

# Попытка создать 14-й
try:
    c14 = ChannelPool("Channel14")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 3
del channels[0], channels[1], channels[2]

# Создаем 14-й, 15-й, 16-й - успешно
c14 = ChannelPool("Channel14")
c15 = ChannelPool("Channel15")
c16 = ChannelPool("Channel16")
print("Созданы каналы:", c14.name, c15.name, c16.name)
```

19. Написать программу на Python, которая создает класс ‘SocketPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 22.

    Инструкции:

    (a) Создайте класс ‘SocketPool‘.

    (b) Добавьте атрибут класса ‘_sockets‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘SOCKET_MAX‘ и инициализируйте его значением 22.

(d) Переопределите метод ‘__new__‘. Если ‘len(_sockets) >= SOCKET_MAX‘, выбросьте ‘RuntimeError("Сокеты: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_sockets‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_sockets‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘socket_port‘ и устанавливает ‘self.port = socket_port‘.

(g) Создайте 22 экземпляра.

(h) Попытайтесь создать 23-й - поймайте и выведите исключение.

(i) Удалите одиннадцать экземпляров.

(j) Создайте 23-й, 24-й, ..., 33-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 22 сокета
sockets = [SocketPool(8000 + i) for i in range(22)]

# Попытка создать 23-й
try:
    s23 = SocketPool(8022)
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 11
for i in range(11):
    del sockets[0]

# Создаем 23-й, 24-й, ..., 33-й - успешно
new_sockets = [SocketPool(8022 + i) for i in range(11)]
for s in new_sockets:
    print("Создан сокет на порту:", s.port)
```

20. Написать программу на Python, которая создает класс ‘LockPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 14.

Инструкции:

(a) Создайте класс ‘LockPool‘.

(b) Добавьте атрибут класса ‘_locks‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘LOCK_COUNT‘ и инициализируйте его значением 14.

(d) Переопределите метод ‘__new__‘. Если ‘len(_locks) >= LOCK_COUNT‘, выбросьте ‘RuntimeError("Замки: все заняты!")‘. Иначе, создайте экземпляр, добавьте в ‘_locks‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_locks‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘lock_name‘ и устанавливает ‘self.name = lock_name‘.

(g) Создайте 14 экземпляров.

(h) Попытайтесь создать 15-й - поймайте и выведите исключение.

(i) Удалите семь экземпляров.

(j) Создайте 15-й, 16-й, ..., 21-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 14 замков
locks = [LockPool(f"Lock{i}") for i in range(14)]

# Попытка создать 15-й
try:
    l15 = LockPool("Lock15")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 7
for i in range(7):
    del locks[0]

# Создаем 15-й, 16-й, ..., 21-й - успешно
new_locks = [LockPool(f"Lock{i}") for i in range(15, 22)]
for l in new_locks:
    print("Создан замок:", l.name)
```

21. Написать программу на Python, которая создает класс ‘QueuePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 19.

Инструкции:

(a) Создайте класс ‘QueuePool‘.

(b) Добавьте атрибут класса ‘_queues‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘QUEUE_COUNT‘ и инициализируйте его значением 19.

(d) Переопределите метод ‘__new__‘. Если ‘len(_queues) >= QUEUE_COUNT‘, выбросьте ‘RuntimeError("Очереди: лимит достигнут!")‘. Иначе, создайте экземпляр, добавьте в ‘_queues‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_queues‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘queue_name‘ и устанавливает ‘self.name = queue_name‘.

(g) Создайте 19 экземпляров.

(h) Попытайтесь создать 20-й - поймайте и выведите исключение.

(i) Удалите десять экземпляров.

(j) Создайте 20-й, 21-й, ..., 29-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 19 очередей
queues = [QueuePool(f"Queue{i}") for i in range(19)]

# Попытка создать 20-ю
try:
    q20 = QueuePool("Queue20")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 10
for i in range(10):
```

```
    del queues[0]

# Создаем 20-ю, 21-ю, ..., 29-ю - успешно
new_queues = [QueuePool(f"Queue{i}") for i in range(20, 30)]
for q in new_queues:
    print("Создана очередь:", q.name)
```

22. Написать программу на Python, которая создает класс ‘SemaphorePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 8.

    Инструкции:

    (a) Создайте класс ‘SemaphorePool‘.

    (b) Добавьте атрибут класса ‘_semaphores‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘SEMA_LIMIT‘ и инициализируйте его значением 8.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_semaphores) >= SEMA_LIMIT‘, выбросьте ‘RuntimeError("Семафоры: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_semaphores‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_semaphores‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘sema_id‘ и устанавливает ‘self.id = sema_id‘.

    (g) Создайте 8 экземпляров.

    (h) Попытайтесь создать 9-й - поймайте и выведите исключение.

    (i) Удалите четыре экземпляра.

    (j) Создайте 9-й, 10-й, 11-й, 12-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 8 семафоров
semas = [SemaphorePool(f"Sema{i}") for i in range(8)]

# Попытка создать 9-й
try:
    s9 = SemaphorePool("Sema9")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 4
del semas[0], semas[1], semas[2], semas[3]

# Создаем 9-й, 10-й, 11-й, 12-й - успешно
s9 = SemaphorePool("Sema9")
s10 = SemaphorePool("Sema10")
s11 = SemaphorePool("Sema11")
s12 = SemaphorePool("Sema12")
print("Созданы семафоры:", s9.id, s10.id, s11.id, s12.id)
```

23. Написать программу на Python, которая создает класс ‘TimerPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 21.

    Инструкции:

    (a) Создайте класс ‘TimerPool‘.

(b) Добавьте атрибут класса ‘_timers‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘TIMER_MAX‘ и инициализируйте его значением 21.

(d) Переопределите метод ‘__new__‘. Если ‘len(_timers) >= TIMER_MAX‘, выбросьте ‘RuntimeError("Таймеры: лимит исчерпан!")‘. Иначе, создайте экземпляр, добавьте в ‘_timers‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_timers‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘timer_duration‘ и устанавливает ‘self.duration = timer_duration‘.

(g) Создайте 21 экземпляр.

(h) Попытайтесь создать 22-й - поймайте и выведите исключение.

(i) Удалите одиннадцать экземпляров.

(j) Создайте 22-й, 23-й, ..., 32-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 21 таймер
timers = [TimerPool(i * 10) for i in range(21)]

# Попытка создать 22-й
try:
    t22 = TimerPool(220)
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 11
for i in range(11):
    del timers[0]

# Создаем 22-й, 23-й, ..., 32-й - успешно
new_timers = [TimerPool(i * 10) for i in range(22, 33)]
for t in new_timers:
    print("Создан таймер на:", t.duration, "сек")
```

24. Написать программу на Python, которая создает класс ‘WorkerPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 23.

Инструкции:

(a) Создайте класс ‘WorkerPool‘.

(b) Добавьте атрибут класса ‘_workers‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘WORKER_LIMIT‘ и инициализируйте его значением 23.

(d) Переопределите метод ‘__new__‘. Если ‘len(_workers) >= WORKER_LIMIT‘, выбросьте ‘RuntimeError("Рабочие: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_workers‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_workers‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘worker_id‘ и устанавливает ‘self.id = worker_id‘.

(g) Создайте 23 экземпляра.

(h) Попытайтесь создать 24-й - поймайте и выведите исключение.

(i) Удалите двенадцать экземпляров.

(j) Создайте 24-й, 25-й, ..., 35-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 23 рабочих
workers = [WorkerPool(f"Worker{i}") for i in range(23)]

# Попытка создать 24-го
try:
    w24 = WorkerPool("Worker24")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 12
for i in range(12):
    del workers[0]

# Создаем 24-го, 25-го, ..., 35-го - успешно
new_workers = [WorkerPool(f"Worker{i}") for i in range(24, 36)]
for w in new_workers:
    print("Создан рабочий:", w.id)
```

25. Написать программу на Python, которая создает класс ‘JobPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 26.

Инструкции:

(a) Создайте класс ‘JobPool‘.

(b) Добавьте атрибут класса ‘_jobs‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘JOB_CAP‘ и инициализируйте его значением 26.

(d) Переопределите метод ‘__new__‘. Если ‘len(_jobs) >= JOB_CAP‘, выбросьте ‘RuntimeError("Задания: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_jobs‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_jobs‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘job_name‘ и устанавливает ‘self.name = job_name‘.

(g) Создайте 26 экземпляров.

(h) Попытайтесь создать 27-й - поймайте и выведите исключение.

(i) Удалите тринадцать экземпляров.

(j) Создайте 27-й, 28-й, ..., 39-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 26 заданий
jobs = [JobPool(f"Job{i}") for i in range(26)]

# Попытка создать 27-е
try:
    j27 = JobPool("Job27")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 13
```

```
for i in range(13):
    del jobs[0]

# Создаем 27-е, 28-е, ..., 39-е - успешно
new_jobs = [JobPool(f"Job{i}") for i in range(27, 40)]
for j in new_jobs:
    print("Создано задание:", j.name)
```

26. Написать программу на Python, которая создает класс ‘RequestPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 27.

    Инструкции:

    (a) Создайте класс ‘RequestPool‘.

    (b) Добавьте атрибут класса ‘_requests‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘REQUEST_MAX‘ и инициализируйте его значением 27.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_requests) >= REQUEST_MAX‘, выбросьте ‘RuntimeError("Запросы: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_requests‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_requests‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘request_url‘ и устанавливает ‘self.url = request_url‘.

    (g) Создайте 27 экземпляров.

    (h) Попытайтесь создать 28-й - поймайте и выведите исключение.

    (i) Удалите четырнадцать экземпляров.

    (j) Создайте 28-й, 29-й, ..., 41-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 27 запросов
requests = [RequestPool(f"http://site{i}.com") for i in range(27)]

# Попытка создать 28-й
try:
    r28 = RequestPool("http://newsite.com")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 14
for i in range(14):
    del requests[0]

# Создаем 28-й, 29-й, ..., 41-й - успешно
new_requests = [RequestPool(f"http://newsite{i}.com") for i in range(28, 42)]
for r in new_requests:
    print("Создан запрос к:", r.url)
```

27. Написать программу на Python, которая создает класс ‘EventPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 28.

    Инструкции:

    (a) Создайте класс ‘EventPool‘.

(b) Добавьте атрибут класса ‘_events‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘EVENT_LIMIT‘ и инициализируйте его значением 28.

(d) Переопределите метод ‘__new__‘. Если ‘len(_events) >= EVENT_LIMIT‘, выбросьте ‘RuntimeError("События: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_events‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_events‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘event_type‘ и устанавливает ‘self.type = event_type‘.

(g) Создайте 28 экземпляров.

(h) Попытайтесь создать 29-е - поймайте и выведите исключение.

(i) Удалите пятнадцать экземпляров.

(j) Создайте 29-е, 30-е, ..., 43-е экземпляры - должно сработать.

Пример использования:

```
# Создаем 28 событий
events = [EventPool(f"Event{i}") for i in range(28)]

# Попытка создать 29-е
try:
    e29 = EventPool("Event29")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 15
for i in range(15):
    del events[0]

# Создаем 29-е, 30-е, ..., 43-е - успешно
new_events = [EventPool(f"Event{i}") for i in range(29, 44)]
for e in new_events:
    print("Создано событие типа:", e.type)
```

28. Написать программу на Python, которая создает класс ‘MessagePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 29.

Инструкции:

(a) Создайте класс ‘MessagePool‘.

(b) Добавьте атрибут класса ‘_messages‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘MSG_MAX‘ и инициализируйте его значением 29.

(d) Переопределите метод ‘__new__‘. Если ‘len(_messages) >= MSG_MAX‘, выбросьте ‘RuntimeError("Сообщения: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_messages‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_messages‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘message_text‘ и устанавливает ‘self.text = message_text‘.

(g) Создайте 29 экземпляров.

(h) Попытайтесь создать 30-й - поймайте и выведите исключение.

(i) Удалите шестнадцать экземпляров.

(j) Создайте 30-й, 31-й, ..., 45-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 29 сообщений
messages = [MessagePool(f"Message{i}") for i in range(29)]

# Попытка создать 30-е
try:
    m30 = MessagePool("Message30")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 16
for i in range(16):
    del messages[0]

# Создаем 30-е, 31-е, ..., 45-е - успешно
new_messages = [MessagePool(f"Message{i}") for i in range(30, 46)]
for m in new_messages:
    print("Создано сообщение:", m.text)
```

29. Написать программу на Python, которая создает класс ‘NotificationPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 31.

    Инструкции:

    (a) Создайте класс ‘NotificationPool‘.

    (b) Добавьте атрибут класса ‘_notifications‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘NOTIF_LIMIT‘ и инициализируйте его значением 31.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_notifications) >= NOTIF_LIMIT‘, выбросьте ‘RuntimeError("Уведомления: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_notifications‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_notifications‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘notification_title‘ и устанавливает ‘self.title = notification_title‘.

    (g) Создайте 31 экземпляр.

    (h) Попытайтесь создать 32-й - поймайте и выведите исключение.

    (i) Удалите семнадцать экземпляров.

    (j) Создайте 32-й, 33-й, ..., 48-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 31 уведомление
notifications = [NotificationPool(f"Notif{i}") for i in range(31)]

# Попытка создать 32-е
try:
    n32 = NotificationPool("Notif32")
except RuntimeError as e:
    print("Ошибка:", e)
```

```
# Удаляем 17
for i in range(17):
    del notifications[0]

# Создаем 32-е, 33-е, ..., 48-е - успешно
new_notifications = [NotificationPool(f"Notif{i}") for i in range(32, 49)]
for n in new_notifications:
    print("Создано уведомление:", n.title)
```

30. Написать программу на Python, которая создает класс ‘LoggerPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 5.

    Инструкции:

    (a) Создайте класс ‘LoggerPool‘.

    (b) Добавьте атрибут класса ‘_loggers‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘LOGGER_LIMIT‘ и инициализируйте его значением 5.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_loggers) >= LOGGER_LIMIT‘, выбросьте ‘RuntimeError("Логгеры: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_loggers‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_loggers‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘logger_name‘ и устанавливает ‘self.name = logger_name‘.

    (g) Создайте 5 экземпляров.

    (h) Попытайтесь создать 6-й - поймайте и выведите исключение.

    (i) Удалите два экземпляра.

    (j) Создайте 6-й и 7-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 5 логгеров
loggers = [LoggerPool(f"Logger{i}") for i in range(5)]

# Попытка создать 6-й
try:
    l6 = LoggerPool("Logger6")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 2
del loggers[0], loggers[1]

# Создаем 6-й и 7-й - успешно
l6 = LoggerPool("Logger6")
l7 = LoggerPool("Logger7")
print("Созданы логгеры:", l6.name, l7.name)
```

31. Написать программу на Python, которая создает класс ‘ConfigPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 12.

    Инструкции:

    (a) Создайте класс ‘ConfigPool‘.

    (b) Добавьте атрибут класса ‘_configs‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘CONFIG_MAX‘ и инициализируйте его значением 12.

(d) Переопределите метод ‘__new__‘. Если ‘len(_configs) >= CONFIG_MAX‘, выбросьте ‘RuntimeError("Конфигурации: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_configs‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_configs‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘config_name‘ и устанавливает ‘self.name = config_name‘.

(g) Создайте 12 экземпляров.

(h) Попытайтесь создать 13-й - поймайте и выведите исключение.

(i) Удалите шесть экземпляров.

(j) Создайте 13-й, 14-й, ..., 18-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 12 конфигураций
configs = [ConfigPool(f"Config{i}") for i in range(12)]

# Попытка создать 13-ю
try:
    c13 = ConfigPool("Config13")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 6
for i in range(6):
    del configs[0]

# Создаем 13-ю, 14-ю, ..., 18-ю - успешно
new_configs = [ConfigPool(f"Config{i}") for i in range(13, 19)]
for c in new_configs:
    print("Создана конфигурация:", c.name)
```

32. Написать программу на Python, которая создает класс ‘PluginPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 10.

Инструкции:

(a) Создайте класс ‘PluginPool‘.

(b) Добавьте атрибут класса ‘_plugins‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘PLUGIN_CAP‘ и инициализируйте его значением 10.

(d) Переопределите метод ‘__new__‘. Если ‘len(_plugins) >= PLUGIN_CAP‘, выбросьте ‘RuntimeError("Плагины: лимит исчерпан!")‘. Иначе, создайте экземпляр, добавьте в ‘_plugins‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_plugins‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘plugin_id‘ и устанавливает ‘self.id = plugin_id‘.

(g) Создайте 10 экземпляров.

(h) Попытайтесь создать 11-й - поймайте и выведите исключение.

(i) Удалите пять экземпляров.

(j) Создайте 11-й, 12-й, ..., 15-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 10 плагинов
plugins = [PluginPool(f"Plugin{i}") for i in range(10)]

# Попытка создать 11-й
try:
    p11 = PluginPool("Plugin11")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 5
for i in range(5):
    del plugins[0]

# Создаем 11-й, 12-й, ..., 15-й - успешно
new_plugins = [PluginPool(f"Plugin{i}") for i in range(11, 16)]
for p in new_plugins:
    print("Создан плагин:", p.id)
```

33. Написать программу на Python, которая создает класс ‘ServicePool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 8.

Инструкции:

(a) Создайте класс ‘ServicePool‘.

(b) Добавьте атрибут класса ‘_services‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘SERVICE_LIMIT‘ и инициализируйте его значением 8.

(d) Переопределите метод ‘__new__‘. Если ‘len(_services) >= SERVICE_LIMIT‘, выбросьте ‘RuntimeError("Сервисы: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_services‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_services‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘service_name‘ и устанавливает ‘self.name = service_name‘.

(g) Создайте 8 экземпляров.

(h) Попытайтесь создать 9-й - поймайте и выведите исключение.

(i) Удалите четыре экземпляра.

(j) Создайте 9-й, 10-й, 11-й, 12-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 8 сервисов
services = [ServicePool(f"Service{i}") for i in range(8)]

# Попытка создать 9-й
try:
    s9 = ServicePool("Service9")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 4
del services[0], services[1], services[2], services[3]
```

```
# Создаем 9-й, 10-й, 11-й, 12-й - успешно
s9 = ServicePool("Service9")
s10 = ServicePool("Service10")
s11 = ServicePool("Service11")
s12 = ServicePool("Service12")
print("Созданы сервисы:", s9.name, s10.name, s11.name, s12.name)
```

34. Написать программу на Python, которая создает класс ‘CacheEntryPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 15.

    Инструкции:

    (a) Создайте класс ‘CacheEntryPool‘.

    (b) Добавьте атрибут класса ‘_entries‘ и инициализируйте его пустым списком.

    (c) Добавьте атрибут класса ‘ENTRY_MAX‘ и инициализируйте его значением 15.

    (d) Переопределите метод ‘__new__‘. Если ‘len(_entries) >= ENTRY_MAX‘, выбросьте ‘RuntimeError("Кэш-записи: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_entries‘, верните.

    (e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_entries‘.

    (f) Переопределите метод ‘__init__‘, который принимает ‘entry_key‘ и устанавливает ‘self.key = entry_key‘.

    (g) Создайте 15 экземпляров.

    (h) Попытайтесь создать 16-й - поймайте и выведите исключение.

    (i) Удалите семь экземпляров.

    (j) Создайте 16-й, 17-й, ..., 22-й экземпляры - должно сработать.

    Пример использования:

```
# Создаем 15 записей
entries = [CacheEntryPool(f"Key{i}") for i in range(15)]

# Попытка создать 16-ю
try:
    e16 = CacheEntryPool("Key16")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 7
for i in range(7):
    del entries[0]

# Создаем 16-ю, 17-ю, ..., 22-ю - успешно
new_entries = [CacheEntryPool(f"Key{i}") for i in range(16, 23)]
for e in new_entries:
    print("Создана запись с ключом:", e.key)
```

35. Написать программу на Python, которая создает класс ‘ConnectionHandlerPool‘ с использованием метода ‘__new__‘ для ограничения количества создаваемых экземпляров до 20.

    Инструкции:

    (a) Создайте класс ‘ConnectionHandlerPool‘.

(b) Добавьте атрибут класса ‘_handlers‘ и инициализируйте его пустым списком.

(c) Добавьте атрибут класса ‘HANDLER_LIMIT‘ и инициализируйте его значением 20.

(d) Переопределите метод ‘__new__‘. Если ‘len(_handlers) >= HANDLER_LIMIT‘, выбросьте ‘RuntimeError("Обработчики соединений: лимит превышен!")‘. Иначе, создайте экземпляр, добавьте в ‘_handlers‘, верните.

(e) Переопределите метод ‘__del__‘, чтобы он удалял ‘self‘ из ‘_handlers‘.

(f) Переопределите метод ‘__init__‘, который принимает ‘handler_id‘ и устанавливает ‘self.id = handler_id‘.

(g) Создайте 20 экземпляров.

(h) Попытайтесь создать 21-й - поймайте и выведите исключение.

(i) Удалите десять экземпляров.

(j) Создайте 21-й, 22-й, ..., 30-й экземпляры - должно сработать.

Пример использования:

```
# Создаем 20 обработчиков
handlers = [ConnectionHandlerPool(f"H{i}") for i in range(20)]

# Попытка создать 21-й
try:
    h21 = ConnectionHandlerPool("H21")
except RuntimeError as e:
    print("Ошибка:", e)

# Удаляем 10
for i in range(10):
    del handlers[0]

# Создаем 21-й, 22-й, ..., 30-й - успешно
new_handlers = [ConnectionHandlerPool(f"H{i}") for i in range(21, 31)]
for h in new_handlers:
    print("Создан обработчик:", h.id)
```

2.4.3 Задача 3 (именование)

1. Написать программу на Python, которая создает класс ‘Vagon‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "vagon_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Vagon‘.

   (b) Добавьте атрибут класса ‘numbers‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘number‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "vagon_ выбросьте ‘ValueError("Имя должно начинаться с ’vagon_’")‘.

   (e) Извлеките номер вагона: ‘vagon_number = name[6:]‘ (удаляем "vagon_").

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте номер в словарь: ‘cls.numbers[vagon_number] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"vvagon_number number)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой: ‘def __init__(self, *args, **kwargs): pass‘.

(k) Создайте объект ‘v1‘ с именем "vagon_1"и номером 101.

(l) Создайте объект ‘v2‘ с именем "vagon_2"и номером 102.

(m) Попытайтесь создать объект с именем "car_3"— должно возникнуть исключение ‘ValueError‘.

(n) Выведите ‘v1.v1‘ и ‘v2.v2‘.

(o) Выведите ‘Vagon.numbers‘.

Пример использования:

```
v1 = Vagon("vagon_1", 101)
v2 = Vagon("vagon_2", 102)

try:
    v3 = Vagon("car_3", 103)
except ValueError as e:
    print("Ошибка:", e)

print("v1.v1:", v1.v1)  # 101
print("v2.v2:", v2.v2)  # 102
print("Vagon.numbers:", Vagon.numbers)
```

2. Написать программу на Python, которая создает класс ‘Room‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "room_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘Room‘.

(b) Добавьте атрибут класса ‘registry‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘capacity‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "room_ выбросьте ‘ValueError("Недопустимое имя комнаты")‘.

(e) Извлеките номер комнаты: ‘room_num = name[5:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте номер в словарь: ‘cls.registry[room_num] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"rroom_num capacity)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘r1‘ с именем "room_101"и вместимостью 50.

(l) Создайте объект ‘r2‘ с именем "room_202"и вместимостью 30.

(m) Попытайтесь создать объект с именем "hall_A"— поймайте исключение.

(n) Выведите ‘r1.r101‘ и ‘r2.r202‘.

(o) Выведите ‘Room.registry‘.

Пример использования:

```
r1 = Room("room_101", 50)
r2 = Room("room_202", 30)

try:
    r3 = Room("hall_A", 100)
except ValueError as e:
    print("Ошибка:", e)

print("r1.r101:", r1.r101)  # 50
print("r2.r202:", r2.r202)  # 30
print("Room.registry:", Room.registry)
```

3. Написать программу на Python, которая создает класс ‘Device‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "dev_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Device‘.

   (b) Добавьте атрибут класса ‘inventory‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘model‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "dev_ выбросьте ‘ValueError("Неверный префикс устройства")‘.

   (e) Извлеките ID устройства: ‘dev_id = name[4:]‘.

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте ID в словарь: ‘cls.inventory[dev_id] = instance‘.

   (h) Установите атрибут экземпляра: ‘setattr(instance, f"ddev_id model)‘.

   (i) Верните ‘instance‘.

   (j) Переопределите метод ‘__init__‘ как пустой.

   (k) Создайте объект ‘d1‘ с именем "dev_001"и моделью "X1".

   (l) Создайте объект ‘d2‘ с именем "dev_002"и моделью "Y2".

   (m) Попытайтесь создать объект с именем "sensor_01"— поймайте исключение.

   (n) Выведите ‘d1.d001‘ и ‘d2.d002‘.

   (o) Выведите ‘Device.inventory‘.

   Пример использования:

```
d1 = Device("dev_001", "X1")
d2 = Device("dev_002", "Y2")

try:
    d3 = Device("sensor_01", "Z3")
except ValueError as e:
    print("Ошибка:", e)

print("d1.d001:", d1.d001)  # X1
print("d2.d002:", d2.d002)  # Y2
print("Device.inventory:", Device.inventory)
```

4. Написать программу на Python, которая создает класс ‘Book‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "book_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Book‘.

   (b) Добавьте атрибут класса ‘catalog‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘author‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "book_ выбросьте ‘ValueError("Книга должна иметь префикс ’book_’")‘.

   (e) Извлеките ID книги: ‘book_id = name[5:]‘.

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте ID в словарь: ‘cls.catalog[book_id] = instance‘.

   (h) Установите атрибут экземпляра: ‘setattr(instance, f"bbook_id author)‘.

   (i) Верните ‘instance‘.

   (j) Переопределите метод ‘__init__‘ как пустой.

   (k) Создайте объект ‘b1‘ с именем "book_001"и автором "Толстой".

   (l) Создайте объект ‘b2‘ с именем "book_002"и автором "Достоевский".

   (m) Попытайтесь создать объект с именем "magazine_01"— поймайте исключение.

   (n) Выведите ‘b1.b001‘ и ‘b2.b002‘.

   (o) Выведите ‘Book.catalog‘.

   Пример использования:

```
b1 = Book("book_001", "Толстой")
b2 = Book("book_002", "Достоевский")

try:
    b3 = Book("magazine_01", "Пушкин")
except ValueError as e:
    print("Ошибка:", e)

print("b1.b001:", b1.b001)  # Толстой
print("b2.b002:", b2.b002)  # Достоевский
print("Book.catalog:", Book.catalog)
```

5. Написать программу на Python, которая создает класс ‘File‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "file_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘File‘.

   (b) Добавьте атрибут класса ‘index‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘size‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "file_ выбросьте ‘ValueError("Файл должен иметь префикс ’file_’")‘.

   (e) Извлеките ID файла: ‘file_id = name[5:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.index[file_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ffile_id size)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘f1‘ с именем "file_config"и размером 1024.

(l) Создайте объект ‘f2‘ с именем "file_data"и размером 2048.

(m) Попытайтесь создать объект с именем "document_1"— поймайте исключение.

(n) Выведите ‘f1.fconfig‘ и ‘f2.fdata‘.

(o) Выведите ‘File.index‘.

Пример использования:

```
f1 = File("file_config", 1024)
f2 = File("file_data", 2048)

try:
    f3 = File("document_1", 512)
except ValueError as e:
    print("Ошибка:", e)

print("f1.fconfig:", f1.fconfig)  # 1024
print("f2.fdata:", f2.fdata)      # 2048
print("File.index:", File.index)
```

6. Написать программу на Python, которая создает класс ‘User‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "user_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘User‘.

   (b) Добавьте атрибут класса ‘directory‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘email‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "user_ выбросьте ‘ValueError("Пользователь должен иметь префикс ’user_’")‘.

   (e) Извлеките ID пользователя: ‘user_id = name[5:]‘.

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте ID в словарь: ‘cls.directory[user_id] = instance‘.

   (h) Установите атрибут экземпляра: ‘setattr(instance, f"uuser_id email)‘.

   (i) Верните ‘instance‘.

   (j) Переопределите метод ‘__init__‘ как пустой.

   (k) Создайте объект ‘u1‘ с именем "user_alice"и email "alice@example.com".

   (l) Создайте объект ‘u2‘ с именем "user_bob"и email "bob@example.com".

   (m) Попытайтесь создать объект с именем "admin_john"— поймайте исключение.

   (n) Выведите ‘u1.ualice‘ и ‘u2.ubob‘.

(o) Выведите ‘User.directory‘.

Пример использования:

```
u1 = User("user_alice", "alice@example.com")
u2 = User("user_bob", "bob@example.com")

try:
    u3 = User("admin_john", "john@example.com")
except ValueError as e:
    print("Ошибка:", e)

print("u1.ualice:", u1.ualice)  # alice@example.com
print("u2.ubob:", u2.ubob)      # bob@example.com
print("User.directory:", User.directory)
```

7. Написать программу на Python, которая создает класс ‘Product‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "prod_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Product‘.

   (b) Добавьте атрибут класса ‘warehouse‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘price‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "prod_ выбросьте ‘ValueError("Продукт должен иметь префикс ’prod_’")‘.

   (e) Извлеките ID продукта: ‘prod_id = name[5:]‘.

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте ID в словарь: ‘cls.warehouse[prod_id] = instance‘.

   (h) Установите атрибут экземпляра: ‘setattr(instance, f"pprod_id price)‘.

   (i) Верните ‘instance‘.

   (j) Переопределите метод ‘__init__‘ как пустой.

   (k) Создайте объект ‘p1‘ с именем "prod_laptop"и ценой 999.

   (l) Создайте объект ‘p2‘ с именем "prod_mouse"и ценой 25.

   (m) Попытайтесь создать объект с именем "item_keyboard"— поймайте исключение.

   (n) Выведите ‘p1.plaptop‘ и ‘p2.pmouse‘.

   (o) Выведите ‘Product.warehouse‘.

   Пример использования:

```
p1 = Product("prod_laptop", 999)
p2 = Product("prod_mouse", 25)

try:
    p3 = Product("item_keyboard", 50)
except ValueError as e:
    print("Ошибка:", e)

print("p1.plaptop:", p1.plaptop)  # 999
print("p2.pmouse:", p2.pmouse)    # 25
print("Product.warehouse:", Product.warehouse)
```

8. Написать программу на Python, которая создает класс ‘Employee‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "emp_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Employee‘.

   (b) Добавьте атрибут класса ‘staff‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘department‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "emp_ выбросьте ‘ValueError("Сотрудник должен иметь префикс ’emp_’")‘.

   (e) Извлеките ID сотрудника: ‘emp_id = name[4:]‘.

   (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

   (g) Добавьте ID в словарь: ‘cls.staff[emp_id] = instance‘.

   (h) Установите атрибут экземпляра: ‘setattr(instance, f"eemp_id department)‘.

   (i) Верните ‘instance‘.

   (j) Переопределите метод ‘__init__‘ как пустой.

   (k) Создайте объект ‘e1‘ с именем "emp_001"и отделом "IT".

   (l) Создайте объект ‘e2‘ с именем "emp_002"и отделом "HR".

   (m) Попытайтесь создать объект с именем "worker_003"— поймайте исключение.

   (n) Выведите ‘e1.e001‘ и ‘e2.e002‘.

   (o) Выведите ‘Employee.staff‘.

   Пример использования:

```
e1 = Employee("emp_001", "IT")
e2 = Employee("emp_002", "HR")

try:
    e3 = Employee("worker_003", "Sales")
except ValueError as e:
    print("Ошибка:", e)

print("e1.e001:", e1.e001)  # IT
print("e2.e002:", e2.e002)  # HR
print("Employee.staff:", Employee.staff)
```

9. Написать программу на Python, которая создает класс ‘Order‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "order_". Метод ‘__init__‘ должен быть пустым.

   Инструкции:

   (a) Создайте класс ‘Order‘.

   (b) Добавьте атрибут класса ‘ledger‘ и инициализируйте его пустым словарем.

   (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘total‘.

   (d) В ‘__new__‘: если ‘name‘ не начинается с "order_ выбросьте ‘ValueError("Заказ должен иметь префикс ’order_’")‘.

   (e) Извлеките ID заказа: ‘order_id = name[6:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.ledger[order_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"oorder_id total)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘o1‘ с именем "order_1001"и суммой 150.0.

(l) Создайте объект ‘o2‘ с именем "order_1002"и суммой 89.99.

(m) Попытайтесь создать объект с именем "purchase_1003"— поймайте исключение.

(n) Выведите ‘o1.o1001‘ и ‘o2.o1002‘.

(o) Выведите ‘Order.ledger‘.

Пример использования:

```
o1 = Order("order_1001", 150.0)
o2 = Order("order_1002", 89.99)

try:
    o3 = Order("purchase_1003", 200.0)
except ValueError as e:
    print("Ошибка:", e)

print("o1.o1001:", o1.o1001)  # 150.0
print("o2.o1002:", o2.o1002)  # 89.99
print("Order.ledger:", Order.ledger)
```

10. Написать программу на Python, которая создает класс ‘Ticket‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "ticket_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Ticket‘.

    (b) Добавьте атрибут класса ‘database‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘priority‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "ticket_ выбросьте ‘ValueError("Тикет должен иметь префикс ’ticket_’")‘.

    (e) Извлеките ID тикета: ‘ticket_id = name[7:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.database[ticket_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"tticket_id priority)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘t1‘ с именем "ticket_001"и приоритетом "High".

    (l) Создайте объект ‘t2‘ с именем "ticket_002"и приоритетом "Low".

    (m) Попытайтесь создать объект с именем "issue_003"— поймайте исключение.

    (n) Выведите ‘t1.t001‘ и ‘t2.t002‘.

(o) Выведите ‘Ticket.database‘.

Пример использования:

```
t1 = Ticket("ticket_001", "High")
t2 = Ticket("ticket_002", "Low")

try:
    t3 = Ticket("issue_003", "Medium")
except ValueError as e:
    print("Ошибка:", e)

print("t1.t001:", t1.t001)  # High
print("t2.t002:", t2.t002)  # Low
print("Ticket.database:", Ticket.database)
```

11. Написать программу на Python, которая создает класс ‘Project‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "proj_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Project‘.

    (b) Добавьте атрибут класса ‘portfolio‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘status‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "proj_ выбросьте ‘ValueError("Проект должен иметь префикс ’proj_’")‘.

    (e) Извлеките ID проекта: ‘proj_id = name[5:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.portfolio[proj_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"prproj_id status)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘pr1‘ с именем "proj_alpha"и статусом "Active".

    (l) Создайте объект ‘pr2‘ с именем "proj_beta"и статусом "Inactive".

    (m) Попытайтесь создать объект с именем "task_gamma"— поймайте исключение.

    (n) Выведите ‘pr1.pralpha‘ и ‘pr2.prbeta‘.

    (o) Выведите ‘Project.portfolio‘.

    Пример использования:

```
pr1 = Project("proj_alpha", "Active")
pr2 = Project("proj_beta", "Inactive")

try:
    pr3 = Project("task_gamma", "Pending")
except ValueError as e:
    print("Ошибка:", e)

print("pr1.pralpha:", pr1.pralpha)  # Active
print("pr2.prbeta:", pr2.prbeta)    # Inactive
print("Project.portfolio:", Project.portfolio)
```

12. Написать программу на Python, которая создает класс ‘Sensor‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "sensor_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Sensor‘.

    (b) Добавьте атрибут класса ‘registry‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘type‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "sensor_ выбросьте ‘ValueError("Сенсор должен иметь префикс ’sensor_’")‘.

    (e) Извлеките ID сенсора: ‘sensor_id = name[7:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.registry[sensor_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"ssensor_id type)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘s1‘ с именем "sensor_temp"и типом "Temperature".

    (l) Создайте объект ‘s2‘ с именем "sensor_humid"и типом "Humidity".

    (m) Попытайтесь создать объект с именем "device_press"— поймайте исключение.

    (n) Выведите ‘s1.stemp‘ и ‘s2.shumid‘.

    (o) Выведите ‘Sensor.registry‘.

    Пример использования:

```
s1 = Sensor("sensor_temp", "Temperature")
s2 = Sensor("sensor_humid", "Humidity")

try:
    s3 = Sensor("device_press", "Pressure")
except ValueError as e:
    print("Ошибка:", e)

print("s1.stemp:", s1.stemp)     # Temperature
print("s2.shumid:", s2.shumid)  # Humidity
print("Sensor.registry:", Sensor.registry)
```

13. Написать программу на Python, которая создает класс ‘Vehicle‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "veh_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Vehicle‘.

    (b) Добавьте атрибут класса ‘fleet‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘model‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "veh_ выбросьте ‘ValueError("Транспортное средство должно иметь префикс ’veh_’")‘.

    (e) Извлеките ID транспорта: ‘veh_id = name[4:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.fleet[veh_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"vveh_id model)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘v1‘ с именем "veh_car1"и моделью "Sedan".

(l) Создайте объект ‘v2‘ с именем "veh_truck1"и моделью "Pickup".

(m) Попытайтесь создать объект с именем "bike_01"— поймайте исключение.

(n) Выведите ‘v1.vcar1‘ и ‘v2.vtruck1‘.

(o) Выведите ‘Vehicle.fleet‘.

Пример использования:

```
v1 = Vehicle("veh_car1", "Sedan")
v2 = Vehicle("veh_truck1", "Pickup")

try:
    v3 = Vehicle("bike_01", "Mountain")
except ValueError as e:
    print("Ошибка:", e)

print("v1.vcar1:", v1.vcar1)     # Sedan
print("v2.vtruck1:", v2.vtruck1) # Pickup
print("Vehicle.fleet:", Vehicle.fleet)
```

14. Написать программу на Python, которая создает класс ‘Animal‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "animal_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘Animal‘.

(b) Добавьте атрибут класса ‘zoo‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘species‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "animal_ выбросьте ‘ValueError("Животное должно иметь префикс ’animal_’")‘.

(e) Извлеките ID животного: ‘animal_id = name[7:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.zoo[animal_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"aanimal_id species)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘a1‘ с именем "animal_lion"и видом "Panthera leo".

(l) Создайте объект ‘a2‘ с именем "animal_elephant"и видом "Loxodonta".

(m) Попытайтесь создать объект с именем "creature_tiger"— поймайте исключение.

(n) Выведите ‘a1.alion‘ и ‘a2.aelephant‘.

(o) Выведите ‘Animal.zoo‘.

Пример использования:

```
a1 = Animal("animal_lion", "Panthera leo")
a2 = Animal("animal_elephant", "Loxodonta")

try:
    a3 = Animal("creature_tiger", "Panthera tigris")
except ValueError as e:
    print("Ошибка:", e)

print("a1.alion:", a1.alion)        # Panthera leo
print("a2.aelephant:", a2.aelephant) # Loxodonta
print("Animal.zoo:", Animal.zoo)
```

15. Написать программу на Python, которая создает класс ‘Plant‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "plant_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Plant‘.

    (b) Добавьте атрибут класса ‘greenhouse‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘family‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "plant_ выбросьте ‘ValueError("Растение должно иметь префикс ’plant_’")‘.

    (e) Извлеките ID растения: ‘plant_id = name[6:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.greenhouse[plant_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"plplant_id family)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘pl1‘ с именем "plant_rose"и семейством "Rosaceae".

    (l) Создайте объект ‘pl2‘ с именем "plant_oak"и семейством "Fagaceae".

    (m) Попытайтесь создать объект с именем "tree_pine"— поймайте исключение.

    (n) Выведите ‘pl1.plrose‘ и ‘pl2.ploak‘.

    (o) Выведите ‘Plant.greenhouse‘.

    Пример использования:

```
pl1 = Plant("plant_rose", "Rosaceae")
pl2 = Plant("plant_oak", "Fagaceae")

try:
    pl3 = Plant("tree_pine", "Pinaceae")
except ValueError as e:
    print("Ошибка:", e)

print("pl1.plrose:", pl1.plrose)  # Rosaceae
print("pl2.ploak:", pl2.ploak)    # Fagaceae
print("Plant.greenhouse:", Plant.greenhouse)
```

16. Написать программу на Python, которая создает класс ‘Planet‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "planet_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Planet‘.

    (b) Добавьте атрибут класса ‘solar_system‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘type‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "planet_ выбросьте ‘ValueError("Планета должна иметь префикс ’planet_’")‘.

    (e) Извлеките ID планеты: ‘planet_id = name[7:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.solar_system[planet_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"pnplanet_id type)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘pn1‘ с именем "planet_earth"и типом "Terrestrial".

    (l) Создайте объект ‘pn2‘ с именем "planet_jupiter"и типом "Gas Giant".

    (m) Попытайтесь создать объект с именем "star_sun"— поймайте исключение.

    (n) Выведите ‘pn1.pnearth‘ и ‘pn2.pnjupiter‘.

    (o) Выведите ‘Planet.solar_system‘.

    Пример использования:

```
pn1 = Planet("planet_earth", "Terrestrial")
pn2 = Planet("planet_jupiter", "Gas Giant")

try:
    pn3 = Planet("star_sun", "Star")
except ValueError as e:
    print("Ошибка:", e)

print("pn1.pnearth:", pn1.pnearth)     # Terrestrial
print("pn2.pnjupiter:", pn2.pnjupiter) # Gas Giant
print("Planet.solar_system:", Planet.solar_system)
```

17. Написать программу на Python, которая создает класс ‘Star‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "star_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Star‘.

    (b) Добавьте атрибут класса ‘galaxy‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘class_type‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "star_ выбросьте ‘ValueError("Звезда должна иметь префикс ’star_’")‘.

    (e) Извлеките ID звезды: ‘star_id = name[5:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.galaxy[star_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ststar_id class_type)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘st1‘ с именем "star_sol"и классом "G2V".

(l) Создайте объект ‘st2‘ с именем "star_proxima"и классом "M5.5V".

(m) Попытайтесь создать объект с именем "nova_1"— поймайте исключение.

(n) Выведите ‘st1.stsol‘ и ‘st2.stproxima‘.

(o) Выведите ‘Star.galaxy‘.

Пример использования:

```
st1 = Star("star_sol", "G2V")
st2 = Star("star_proxima", "M5.5V")

try:
    st3 = Star("nova_1", "Variable")
except ValueError as e:
    print("Ошибка:", e)

print("st1.stsol:", st1.stsol)      # G2V
print("st2.stproxima:", st2.stproxima) # M5.5V
print("Star.galaxy:", Star.galaxy)
```

18. Написать программу на Python, которая создает класс ‘Galaxy‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "galaxy_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘Galaxy‘.

(b) Добавьте атрибут класса ‘universe‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘type‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "galaxy_ выбросьте ‘ValueError("Галактика должна иметь префикс ’galaxy_’")‘.

(e) Извлеките ID галактики: ‘galaxy_id = name[7:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.universe[galaxy_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ggalaxy_id type)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘g1‘ с именем "galaxy_milkyway"и типом "Spiral".

(l) Создайте объект ‘g2‘ с именем "galaxy_andromeda"и типом "Spiral".

(m) Попытайтесь создать объект с именем "cluster_virgo"— поймайте исключение.

(n) Выведите ‘g1.gmilkyway‘ и ‘g2.gandromeda‘.

(o) Выведите ‘Galaxy.universe‘.

Пример использования:

```
g1 = Galaxy("galaxy_milkyway", "Spiral")
g2 = Galaxy("galaxy_andromeda", "Spiral")

try:
    g3 = Galaxy("cluster_virgo", "Cluster")
except ValueError as e:
    print("Ошибка:", e)

print("g1.gmilkyway:", g1.gmilkyway)    # Spiral
print("g2.gandromeda:", g2.gandromeda)  # Spiral
print("Galaxy.universe:", Galaxy.universe)
```

19. Написать программу на Python, которая создает класс ‘Constellation‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "const_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Constellation‘.

    (b) Добавьте атрибут класса ‘sky_map‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘stars‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "const_ выбросьте ‘ValueError("Созвездие должно иметь префикс ’const_’")‘.

    (e) Извлеките ID созвездия: ‘const_id = name[6:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.sky_map[const_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"cconst_id stars)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘c1‘ с именем "const_orion"и количеством звезд 81.

    (l) Создайте объект ‘c2‘ с именем "const_ursa"и количеством звезд 20.

    (m) Попытайтесь создать объект с именем "asterism_bigdipper"— поймайте исключение.

    (n) Выведите ‘c1.corion‘ и ‘c2.cursa‘.

    (o) Выведите ‘Constellation.sky_map‘.

    Пример использования:

```
c1 = Constellation("const_orion", 81)
c2 = Constellation("const_ursa", 20)

try:
    c3 = Constellation("asterism_bigdipper", 7)
except ValueError as e:
    print("Ошибка:", e)

print("c1.corion:", c1.corion)  # 81
print("c2.cursa:", c2.cursa)    # 20
print("Constellation.sky_map:", Constellation.sky_map)
```

20. Написать программу на Python, которая создает класс ‘Asteroid‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "ast_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Asteroid‘.

    (b) Добавьте атрибут класса ‘belt‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘diameter‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "ast_ выбросьте ‘ValueError("Астероид должен иметь префикс ’ast_’")‘.

    (e) Извлеките ID астероида: ‘ast_id = name[4:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.belt[ast_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"aast_id diameter)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘a1‘ с именем "ast_ceres"и диаметром 939.

    (l) Создайте объект ‘a2‘ с именем "ast_vesta"и диаметром 525.

    (m) Попытайтесь создать объект с именем "meteor_id8"— поймайте исключение.

    (n) Выведите ‘a1.aceres‘ и ‘a2.avesta‘.

    (o) Выведите ‘Asteroid.belt‘.

    Пример использования:

```
a1 = Asteroid("ast_ceres", 939)
a2 = Asteroid("ast_vesta", 525)

try:
    a3 = Asteroid("meteor_id8", 10)
except ValueError as e:
    print("Ошибка:", e)

print("a1.aceres:", a1.aceres)  # 939
print("a2.avesta:", a2.avesta)  # 525
print("Asteroid.belt:", Asteroid.belt)
```

21. Написать программу на Python, которая создает класс ‘Comet‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "comet_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Comet‘.

    (b) Добавьте атрибут класса ‘orbits‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘period‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "comet_ выбросьте ‘ValueError("Комета должна иметь префикс ’comet_’")‘.

    (e) Извлеките ID кометы: ‘comet_id = name[6:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.orbits[comet_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"cmcomet_id period)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘cm1‘ с именем "comet_halley"и периодом 76.

(l) Создайте объект ‘cm2‘ с именем "comet_encke"и периодом 3.3.

(m) Попытайтесь создать объект с именем "meteor_shower"— поймайте исключение.

(n) Выведите ‘cm1.cmhalley‘ и ‘cm2.cmencke‘.

(o) Выведите ‘Comet.orbits‘.

Пример использования:

```
cm1 = Comet("comet_halley", 76)
cm2 = Comet("comet_encke", 3.3)

try:
    cm3 = Comet("meteor_shower", 0.1)
except ValueError as e:
    print("Ошибка:", e)

print("cm1.cmhalley:", cm1.cmhalley)  # 76
print("cm2.cmencke:", cm2.cmencke)    # 3.3
print("Comet.orbits:", Comet.orbits)
```

22. Написать программу на Python, которая создает класс ‘Satellite‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "sat_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘Satellite‘.

(b) Добавьте атрибут класса ‘orbiters‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘planet‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "sat_ выбросьте ‘ValueError("Спутник должен иметь префикс ’sat_’")‘.

(e) Извлеките ID спутника: ‘sat_id = name[4:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.orbiters[sat_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ssat_id planet)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘s1‘ с именем "sat_moon"и планетой "Earth".

(l) Создайте объект ‘s2‘ с именем "sat_phobos"и планетой "Mars".

(m) Попытайтесь создать объект с именем "rover_curiosity"— поймайте исключение.

(n) Выведите ‘s1.smoon‘ и ‘s2.sphobos‘.

(o) Выведите ‘Satellite.orbiters‘.

Пример использования:

```
s1 = Satellite("sat_moon", "Earth")
s2 = Satellite("sat_phobos", "Mars")

try:
    s3 = Satellite("rover_curiosity", "Mars")
except ValueError as e:
    print("Ошибка:", e)

print("s1.smoon:", s1.smoon)    # Earth
print("s2.sphobos:", s2.sphobos) # Mars
print("Satellite.orbiters:", Satellite.orbiters)
```

23. Написать программу на Python, которая создает класс ‘Rocket‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "rocket_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Rocket‘.

    (b) Добавьте атрибут класса ‘launchpad‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘payload‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "rocket_ выбросьте ‘ValueError("Ракета должна иметь префикс ’rocket_’")‘.

    (e) Извлеките ID ракеты: ‘rocket_id = name[7:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.launchpad[rocket_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"rrocket_id payload)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘r1‘ с именем "rocket_falcon9"и полезной нагрузкой "Starlink".

    (l) Создайте объект ‘r2‘ с именем "rocket_atlas"и полезной нагрузкой "GPS".

    (m) Попытайтесь создать объект с именем "drone_delivery"— поймайте исключение.

    (n) Выведите ‘r1.rfalcon9‘ и ‘r2.ratlas‘.

    (o) Выведите ‘Rocket.launchpad‘.

    Пример использования:

```
r1 = Rocket("rocket_falcon9", "Starlink")
r2 = Rocket("rocket_atlas", "GPS")

try:
    r3 = Rocket("drone_delivery", "Package")
except ValueError as e:
    print("Ошибка:", e)

print("r1.rfalcon9:", r1.rfalcon9)  # Starlink
print("r2.ratlas:", r2.ratlas)      # GPS
print("Rocket.launchpad:", Rocket.launchpad)
```

24. Написать программу на Python, которая создает класс ‘Drone‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "drone_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Drone‘.

    (b) Добавьте атрибут класса ‘fleet‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘range‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "drone_ выбросьте ‘ValueError("Дрон должен иметь префикс ’drone_’")‘.

    (e) Извлеките ID дрона: ‘drone_id = name[6:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.fleet[drone_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"ddrone_id range)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘d1‘ с именем "drone_x1"и дальностью 5.

    (l) Создайте объект ‘d2‘ с именем "drone_x2"и дальностью 10.

    (m) Попытайтесь создать объект с именем "robot_r1"— поймайте исключение.

    (n) Выведите ‘d1.dx1‘ и ‘d2.dx2‘.

    (o) Выведите ‘Drone.fleet‘.

    Пример использования:

```
d1 = Drone("drone_x1", 5)
d2 = Drone("drone_x2", 10)

try:
    d3 = Drone("robot_r1", 2)
except ValueError as e:
    print("Ошибка:", e)

print("d1.dx1:", d1.dx1)  # 5
print("d2.dx2:", d2.dx2)  # 10
print("Drone.fleet:", Drone.fleet)
```

25. Написать программу на Python, которая создает класс ‘Robot‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "robot_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Robot‘.

    (b) Добавьте атрибут класса ‘factory‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘function‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "robot_ выбросьте ‘ValueError("Робот должен иметь префикс ’robot_’")‘.

    (e) Извлеките ID робота: ‘robot_id = name[6:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.factory[robot_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"rbrobot_id function)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘rb1‘ с именем "robot_arm"и функцией "Assembly".

(l) Создайте объект ‘rb2‘ с именем "robot_cleaner"и функцией "Cleaning".

(m) Попытайтесь создать объект с именем "android_unit"— поймайте исключение.

(n) Выведите ‘rb1.rbarm‘ и ‘rb2.rbcleaner‘.

(o) Выведите ‘Robot.factory‘.

Пример использования:

```
rb1 = Robot("robot_arm", "Assembly")
rb2 = Robot("robot_cleaner", "Cleaning")

try:
    rb3 = Robot("android_unit", "General")
except ValueError as e:
    print("Ошибка:", e)

print("rb1.rbarm:", rb1.rbarm)       # Assembly
print("rb2.rbcleaner:", rb2.rbcleaner) # Cleaning
print("Robot.factory:", Robot.factory)
```

26. Написать программу на Python, которая создает класс ‘AI‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "ai_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘AI‘.

(b) Добавьте атрибут класса ‘network‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘capability‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "ai_ выбросьте ‘ValueError("ИИ должен иметь префикс ’ai_’")‘.

(e) Извлеките ID ИИ: ‘ai_id = name[3:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.network[ai_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"aai_id capability)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘a1‘ с именем "ai_alpha"и возможностью "NLP".

(l) Создайте объект ‘a2‘ с именем "ai_beta"и возможностью "CV".

(m) Попытайтесь создать объект с именем "ml_model"— поймайте исключение.

(n) Выведите ‘a1.aalpha‘ и ‘a2.abeta‘.

(o) Выведите ‘AI.network‘.

Пример использования:

```
a1 = AI("ai_alpha", "NLP")
a2 = AI("ai_beta", "CV")

try:
    a3 = AI("ml_model", "Regression")
except ValueError as e:
    print("Ошибка:", e)

print("a1.aalpha:", a1.aalpha)  # NLP
print("a2.abeta:", a2.abeta)    # CV
print("AI.network:", AI.network)
```

27. Написать программу на Python, которая создает класс ‘MLModel‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "model_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘MLModel‘.

(b) Добавьте атрибут класса ‘repository‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘algorithm‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "model_ выбросьте ‘ValueError("Модель должна иметь префикс ’model_’")‘.

(e) Извлеките ID модели: ‘model_id = name[6:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.repository[model_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"mmodel_id algorithm)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘m1‘ с именем "model_logreg"и алгоритмом "Logistic Regression".

(l) Создайте объект ‘m2‘ с именем "model_svm"и алгоритмом "Support Vector Machine".

(m) Попытайтесь создать объект с именем "algo_randomforest"— поймайте исключение.

(n) Выведите ‘m1.mlogreg‘ и ‘m2.msvm‘.

(o) Выведите ‘MLModel.repository‘.

Пример использования:

```
m1 = MLModel("model_logreg", "Logistic Regression")
m2 = MLModel("model_svm", "Support Vector Machine")

try:
    m3 = MLModel("algo_randomforest", "Random Forest")
except ValueError as e:
    print("Ошибка:", e)

print("m1.mlogreg:", m1.mlogreg)  # Logistic Regression
print("m2.msvm:", m2.msvm)        # Support Vector Machine
print("MLModel.repository:", MLModel.repository)
```

28. Написать программу на Python, которая создает класс ‘Dataset‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "dataset_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Dataset‘.

    (b) Добавьте атрибут класса ‘catalog‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘size‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "dataset_ выбросьте ‘ValueError("Набор данных должен иметь префикс ’dataset_’")‘.

    (e) Извлеките ID набора данных: ‘dataset_id = name[8:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.catalog[dataset_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"dsdataset_id size)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘ds1‘ с именем "dataset_train"и размером 10000.

    (l) Создайте объект ‘ds2‘ с именем "dataset_test"и размером 2000.

    (m) Попытайтесь создать объект с именем "data_validation"— поймайте исключение.

    (n) Выведите ‘ds1.dstrain‘ и ‘ds2.dstest‘.

    (o) Выведите ‘Dataset.catalog‘.

    Пример использования:

```
ds1 = Dataset("dataset_train", 10000)
ds2 = Dataset("dataset_test", 2000)

try:
    ds3 = Dataset("data_validation", 2000)
except ValueError as e:
    print("Ошибка:", e)

print("ds1.dstrain:", ds1.dstrain)  # 10000
print("ds2.dstest:", ds2.dstest)    # 2000
print("Dataset.catalog:", Dataset.catalog)
```

29. Написать программу на Python, которая создает класс ‘Feature‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "feat_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Feature‘.

    (b) Добавьте атрибут класса ‘registry‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘type‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "feat_ выбросьте ‘ValueError("Признак должен иметь префикс ’feat_’")‘.

    (e) Извлеките ID признака: ‘feat_id = name[5:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.registry[feat_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ffeat_id type)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘f1‘ с именем "feat_age"и типом "Numeric".

(l) Создайте объект ‘f2‘ с именем "feat_gender"и типом "Categorical".

(m) Попытайтесь создать объект с именем "attr_income"— поймайте исключение.

(n) Выведите ‘f1.fage‘ и ‘f2.fgender‘.

(o) Выведите ‘Feature.registry‘.

Пример использования:

```
f1 = Feature("feat_age", "Numeric")
f2 = Feature("feat_gender", "Categorical")

try:
    f3 = Feature("attr_income", "Numeric")
except ValueError as e:
    print("Ошибка:", e)

print("f1.fage:", f1.fage)      # Numeric
print("f2.fgender:", f2.fgender) # Categorical
print("Feature.registry:", Feature.registry)
```

30. Написать программу на Python, которая создает класс ‘Label‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "label_". Метод ‘__init__‘ должен быть пустым.

Инструкции:

(a) Создайте класс ‘Label‘.

(b) Добавьте атрибут класса ‘index‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘class_name‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "label_ выбросьте ‘ValueError("Метка должна иметь префикс ’label_’")‘.

(e) Извлеките ID метки: ‘label_id = name[6:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.index[label_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"llabel_id class_name)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘l1‘ с именем "label_cat"и классом "Animal".

(l) Создайте объект ‘l2‘ с именем "label_car"и классом "Vehicle".

(m) Попытайтесь создать объект с именем "tag_dog"— поймайте исключение.

(n) Выведите ‘l1.lcat‘ и ‘l2.lcar‘.

(o) Выведите ‘Label.index‘.

Пример использования:

```
l1 = Label("label_cat", "Animal")
l2 = Label("label_car", "Vehicle")

try:
    l3 = Label("tag_dog", "Animal")
except ValueError as e:
    print("Ошибка:", e)

print("l1.lcat:", l1.lcat)  # Animal
print("l2.lcar:", l2.lcar)  # Vehicle
print("Label.index:", Label.index)
```

31. Написать программу на Python, которая создает класс ‘Layer‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "layer_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Layer‘.

    (b) Добавьте атрибут класса ‘stack‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘neurons‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "layer_ выбросьте ‘ValueError("Слой должен иметь префикс ’layer_’")‘.

    (e) Извлеките ID слоя: ‘layer_id = name[6:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.stack[layer_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"lylayer_id neurons)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘ly1‘ с именем "layer_input"и нейронами 784.

    (l) Создайте объект ‘ly2‘ с именем "layer_hidden"и нейронами 128.

    (m) Попытайтесь создать объект с именем "unit_output"— поймайте исключение.

    (n) Выведите ‘ly1.lyinput‘ и ‘ly2.lyhidden‘.

    (o) Выведите ‘Layer.stack‘.

    Пример использования:

```
ly1 = Layer("layer_input", 784)
ly2 = Layer("layer_hidden", 128)

try:
    ly3 = Layer("unit_output", 10)
except ValueError as e:
    print("Ошибка:", e)

print("ly1.lyinput:", ly1.lyinput)    # 784
print("ly2.lyhidden:", ly2.lyhidden)  # 128
print("Layer.stack:", Layer.stack)
```

32. Написать программу на Python, которая создает класс ‘Neuron‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "neuron_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Neuron‘.

    (b) Добавьте атрибут класса ‘brain‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘activation‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "neuron_ выбросьте ‘ValueError("Нейрон должен иметь префикс ’neuron_’")‘.

    (e) Извлеките ID нейрона: ‘neuron_id = name[7:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.brain[neuron_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"nneuron_id activation)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘n1‘ с именем "neuron_1"и активацией "ReLU".

    (l) Создайте объект ‘n2‘ с именем "neuron_2"и активацией "Sigmoid".

    (m) Попытайтесь создать объект с именем "cell_3"— поймайте исключение.

    (n) Выведите ‘n1.n1‘ и ‘n2.n2‘.

    (o) Выведите ‘Neuron.brain‘.

    Пример использования:

```
n1 = Neuron("neuron_1", "ReLU")
n2 = Neuron("neuron_2", "Sigmoid")

try:
    n3 = Neuron("cell_3", "Tanh")
except ValueError as e:
    print("Ошибка:", e)

print("n1.n1:", n1.n1)  # ReLU
print("n2.n2:", n2.n2)  # Sigmoid
print("Neuron.brain:", Neuron.brain)
```

33. Написать программу на Python, которая создает класс ‘Synapse‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "syn_". Метод ‘__init__‘ должен быть пустым.

    Инструкции:

    (a) Создайте класс ‘Synapse‘.

    (b) Добавьте атрибут класса ‘connections‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘weight‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "syn_ выбросьте ‘ValueError("Синапс должен иметь префикс ’syn_’")‘.

    (e) Извлеките ID синапса: ‘syn_id = name[4:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.connections[syn_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"ssyn_id weight)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘s1‘ с именем "syn_a1b1"и весом 0.5.

(l) Создайте объект ‘s2‘ с именем "syn_a2b2"и весом -0.3.

(m) Попытайтесь создать объект с именем "link_x1y1"— поймайте исключение.

(n) Выведите ‘s1.sa1b1‘ и ‘s2.sa2b2‘.

(o) Выведите ‘Synapse.connections‘.

Пример использования:

```
s1 = Synapse("syn_a1b1", 0.5)
s2 = Synapse("syn_a2b2", -0.3)

try:
    s3 = Synapse("link_x1y1", 0.8)
except ValueError as e:
    print("Ошибка:", e)

print("s1.sa1b1:", s1.sa1b1)  # 0.5
print("s2.sa2b2:", s2.sa2b2)  # -0.3
print("Synapse.connections:", Synapse.connections)
```

34. Написать программу на Python, которая создает класс ‘Container‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "container_". Метод ‘__init__‘ должен быть пустым. Инструкции:

(a) Создайте класс ‘Container‘.

(b) Добавьте атрибут класса ‘depot‘ и инициализируйте его пустым словарем.

(c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘volume‘.

(d) В ‘__new__‘: если ‘name‘ не начинается с "container_ выбросьте ‘ValueError("Контейнер должен иметь префикс ’container_’")‘.

(e) Извлеките ID контейнера: ‘container_id = name[9:]‘.

(f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

(g) Добавьте ID в словарь: ‘cls.depot[container_id] = instance‘.

(h) Установите атрибут экземпляра: ‘setattr(instance, f"cncontainer_id volume)‘.

(i) Верните ‘instance‘.

(j) Переопределите метод ‘__init__‘ как пустой.

(k) Создайте объект ‘cn1‘ с именем "container_20ft"и объемом 33.

(l) Создайте объект ‘cn2‘ с именем "container_40ft"и объемом 67.

(m) Попытайтесь создать объект с именем "box_small"— поймайте исключение.

(n) Выведите ‘cn1.cn20ft‘ и ‘cn2.cn40ft‘.

(o) Выведите ‘Container.depot‘.

Пример использования:

```
cn1 = Container("container_20ft", 33)
cn2 = Container("container_40ft", 67)
try:
    cn3 = Container("box_small", 1)
except ValueError as e:
    print("Ошибка:", e)
print("cn1.cn20ft:", cn1.cn20ft)  # 33
print("cn2.cn40ft:", cn2.cn40ft)  # 67
print("Container.depot:", Container.depot)
```

35. Написать программу на Python, которая создает класс ‘Module‘ с использованием метода ‘__new__‘ для контроля именования. Имена должны начинаться с "mod_". Метод ‘__init__‘ должен быть пустым. Инструкции:

    (a) Создайте класс ‘Module‘.

    (b) Добавьте атрибут класса ‘system‘ и инициализируйте его пустым словарем.

    (c) Переопределите метод ‘__new__‘, принимающий ‘cls‘, ‘name‘, ‘version‘.

    (d) В ‘__new__‘: если ‘name‘ не начинается с "mod_ выбросьте ‘ValueError("Модуль должен иметь префикс ’mod_’")‘.

    (e) Извлеките ID модуля: ‘mod_id = name[4:]‘.

    (f) Создайте экземпляр: ‘instance = super().__new__(cls)‘.

    (g) Добавьте ID в словарь: ‘cls.system[mod_id] = instance‘.

    (h) Установите атрибут экземпляра: ‘setattr(instance, f"mdmod_id version)‘.

    (i) Верните ‘instance‘.

    (j) Переопределите метод ‘__init__‘ как пустой.

    (k) Создайте объект ‘md1‘ с именем "mod_auth"и версией "1.2.0".

    (l) Создайте объект ‘md2‘ с именем "mod_payment"и версией "3.0.1".

    (m) Попытайтесь создать объект с именем "lib_utils"— поймайте исключение.

    (n) Выведите ‘md1.mdauth‘ и ‘md2.mdpayment‘.

    (o) Выведите ‘Module.system‘.

    Пример использования:

```
md1 = Module("mod_auth", "1.2.0")
md2 = Module("mod_payment", "3.0.1")
try:
    md3 = Module("lib_utils", "0.9.5")
except ValueError as e:
    print("Ошибка:", e)
print("md1.mdauth:", md1.mdauth)    # 1.2.0
print("md2.mdpayment:", md2.mdpayment) # 3.0.1
print("Module.system:", Module.system)
```

2.4.4 Задача 4

1 Написать программу на Python, которая создает класс `ShoppingCart` для представления корзины покупок. Класс должен содержать методы для добавления и удаления товаров, а также вычисления общего количества. Программа также должна создавать экземпляр класса `ShoppingCart`, добавлять товары в корзину, удалять товары из корзины и выводить информацию о корзине на экран.

- Создайте класс `ShoppingCart` с методом `__init__`, который создает пустой список товаров.
- Создайте метод `add_item`, который принимает название товара и количество в качестве аргументов и добавляет их в список товаров.
- Создайте метод `remove_item`, который удаляет товар из списка товаров по его названию.
- Создайте метод `calculate_total`, который вычисляет и возвращает общее количество всех товаров в корзине.
- Создайте экземпляр класса `ShoppingCart` и добавьте товары в корзину.
- Выведите информацию о текущих товарах в корзине на экран.
- Выведите общее количество всех товаров в корзине на экран.
- Удалите товар из корзины и выведите обновленную информацию о товарах в корзине на экран.
- Выведите общее количество всех товаров в корзине после удаления товара на экран.

Пример использования:

```
cart = ShoppingCart()
cart.add_item("Картофель", 100)
cart.add_item("Капуста", 200)
cart.add_item("Апельсин", 150)
print("Число товаров в корзине:")
for item in cart.items:
    print(item[0], "-", item[1])
total_qty = cart.calculate_total()
print("Общее количество:", total_qty)
cart.remove_item("Апельсин")
print("Обновление числа покупок в корзине после удаления апельсина:")
for item in cart.items:
    print(item[0], "-", item[1])
total_qty = cart.calculate_total()
print("Общее количество:", total_qty)
```

Вывод:

```
Число товаров в корзине:
Картофель - 100
Капуста - 200
```

```
Апельсин - 150
Общее количество: 450
Обновление числа покупок в корзине после удаления апельсина:
Картофель - 100
Капуста - 200
Общее количество: 300
```

2 Написать программу на Python, которая создает класс `BookCollection` для представления коллекции книг. Класс должен содержать методы для добавления и удаления книг, а также подсчета общего количества страниц. Программа также должна создавать экземпляр класса `BookCollection`, добавлять книги в коллекцию, удалять книги из коллекции и выводить информацию о коллекции на экран.

- Создайте класс `BookCollection` с методом `__init__`, который создает пустой список книг.
- Создайте метод `add_book`, который принимает название книги и количество страниц в качестве аргументов и добавляет их в список книг.
- Создайте метод `remove_book`, который удаляет книгу из списка по её названию.
- Создайте метод `total_pages`, который вычисляет и возвращает общее количество страниц всех книг в коллекции.
- Создайте экземпляр класса `BookCollection` и добавьте книги в коллекцию.
- Выведите информацию о текущих книгах в коллекции на экран.
- Выведите общее количество страниц всех книг на экран.
- Удалите книгу из коллекции и выведите обновленную информацию о книгах на экран.
- Выведите общее количество страниц после удаления книги на экран.

Пример использования:

```
collection = BookCollection()
collection.add_book("Война и мир", 1225)
collection.add_book("Преступление и наказание", 671)
collection.add_book("Мастер и Маргарита", 480)
print("Книги в коллекции:")
for book in collection.books:
    print(book[0], "-", book[1], "стр.")
total = collection.total_pages()
print("Общее количество страниц:", total)
collection.remove_book("Преступление и наказание")
print("Книги после удаления 'Преступления и наказания':")
for book in collection.books:
    print(book[0], "-", book[1], "стр.")
total = collection.total_pages()
print("Общее количество страниц:", total)
```

Вывод:

```
Книги в коллекции:
Война и мир - 1225 стр.
Преступление и наказание - 671 стр.
Мастер и Маргарита - 480 стр.
Общее количество страниц: 2376
Книги после удаления 'Преступления и наказания':
Война и мир - 1225 стр.
Мастер и Маргарита - 480 стр.
Общее количество страниц: 1705
```

3 Написать программу на Python, которая создает класс `Inventory` для представления складского запаса. Класс должен содержать методы для добавления и удаления предметов, а также вычисления общего количества единиц товара. Программа также должна создавать экземпляр класса `Inventory`, добавлять предметы на склад, удалять предметы со склада и выводить информацию о запасах на экран.

- Создайте класс `Inventory` с методом `__init__`, который создает пустой список предметов.
- Создайте метод `add_item`, который принимает название предмета и количество единиц в качестве аргументов и добавляет их в список.
- Создайте метод `remove_item`, который удаляет предмет из списка по его названию.
- Создайте метод `total_count`, который вычисляет и возвращает общее количество всех единиц товара на складе.
- Создайте экземпляр класса `Inventory` и добавьте предметы на склад.
- Выведите информацию о текущих предметах на складе на экран.
- Выведите общее количество единиц товара на экран.
- Удалите предмет со склада и выведите обновленную информацию о предметах на экран.
- Выведите общее количество единиц товара после удаления предмета на экран.

Пример использования:

```
inv = Inventory()
inv.add_item("Молотки", 50)
inv.add_item("Отвертки", 120)
inv.add_item("Гвозди", 1000)
print("Предметы на складе:")
for item in inv.items:
    print(item[0], "-", item[1])
total = inv.total_count()
print("Общее количество:", total)
inv.remove_item("Отвертки")
print("Предметы после удаления отверток:")
for item in inv.items:
    print(item[0], "-", item[1])
total = inv.total_count()
print("Общее количество:", total)
```

Вывод:

```
Предметы на складе:
Молотки - 50
Отвертки - 120
Гвозди - 1000
Общее количество: 1170
Предметы после удаления отверток:
Молотки - 50
Гвозди - 1000
Общее количество: 1050
```

4 Написать программу на Python, которая создает класс `Playlist` для представления музыкального плейлиста. Класс должен содержать методы для добавления и удаления треков, а также подсчета общего времени воспроизведения. Программа также должна создавать экземпляр класса `Playlist`, добавлять треки в плейлист, удалять треки из плейлиста и выводить информацию о плейлисте на экран.

- Создайте класс `Playlist` с методом `__init__`, который создает пустой список треков.
- Создайте метод `add_track`, который принимает название трека и его длительность (в секундах) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_track`, который удаляет трек из списка по его названию.
- Создайте метод `total_duration`, который вычисляет и возвращает общую длительность всех треков в плейлисте (в секундах).
- Создайте экземпляр класса `Playlist` и добавьте треки в плейлист.
- Выведите информацию о текущих треках в плейлисте на экран.
- Выведите общую длительность всех треков на экран.
- Удалите трек из плейлиста и выведите обновленную информацию о треках на экран.
- Выведите общую длительность после удаления трека на экран.

Пример использования:

```
pl = Playlist()
pl.add_track("Bohemian Rhapsody", 354)
pl.add_track("Imagine", 183)
pl.add_track("Smells Like Teen Spirit", 301)
print("Треки в плейлисте:")
for track in pl.tracks:
    print(track[0], "-", track[1], "сек.")
total = pl.total_duration()
print("Общая длительность:", total, "сек.")
pl.remove_track("Imagine")
print("Треки после удаления 'Imagine':")
for track in pl.tracks:
    print(track[0], "-", track[1], "сек.")
total = pl.total_duration()
print("Общая длительность:", total, "сек.")
```

Вывод:

```
Треки в плейлисте:
Bohemian Rhapsody - 354 сек.
Imagine - 183 сек.
Smells Like Teen Spirit - 301 сек.
Общая длительность: 838 сек.
Треки после удаления 'Imagine':
Bohemian Rhapsody - 354 сек.
Smells Like Teen Spirit - 301 сек.
Общая длительность: 655 сек.
```

5 Написать программу на Python, которая создает класс `StudentGrades` для представления оценок студента. Класс должен содержать методы для добавления и удаления оценок, а также вычисления среднего балла. Программа также должна создавать экземпляр класса `StudentGrades`, добавлять оценки, удалять оценки и выводить информацию об успеваемости на экран.

- Создайте класс `StudentGrades` с методом `__init__`, который создает пустой список оценок.
- Создайте метод `add_grade`, который принимает название предмета и оценку в качестве аргументов и добавляет их в список.
- Создайте метод `remove_grade`, который удаляет оценку по названию предмета.
- Создайте метод `average_grade`, который вычисляет и возвращает средний балл по всем предметам.
- Создайте экземпляр класса `StudentGrades` и добавьте оценки по разным предметам.
- Выведите информацию о текущих оценках на экран.
- Выведите средний балл на экран.
- Удалите оценку по одному из предметов и выведите обновленную информацию.
- Выведите средний балл после удаления оценки на экран.

Пример использования:

```
grades = StudentGrades()
grades.add_grade("Математика", 5)
grades.add_grade("Физика", 4)
grades.add_grade("Информатика", 5)
print("Оценки студента:")
for subject, grade in grades.grades:
    print(subject, "-", grade)
avg = grades.average_grade()
print("Средний балл:", round(avg, 2))
grades.remove_grade("Физика")
print("Оценки после удаления Физики:")
for subject, grade in grades.grades:
    print(subject, "-", grade)
avg = grades.average_grade()
print("Средний балл:", round(avg, 2))
```

Вывод:

```
Оценки студента:
Математика - 5
Физика - 4
Информатика - 5
Средний балл: 4.67
Оценки после удаления Физики:
Математика - 5
Информатика - 5
Средний балл: 5.0
```

6 Написать программу на Python, которая создает класс `TaskList` для представления списка задач. Класс должен содержать методы для добавления и удаления задач, а также подсчета общего количества задач. Программа также должна создавать экземпляр класса `TaskList`, добавлять задачи, удалять задачи и выводить информацию о списке задач на экран.

- Создайте класс `TaskList` с методом `__init__`, который создает пустой список задач.
- Создайте метод `add_task`, который принимает описание задачи и приоритет (целое число) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_task`, который удаляет задачу из списка по её описанию.
- Создайте метод `task_count`, который возвращает общее количество задач в списке.
- Создайте экземпляр класса `TaskList` и добавьте несколько задач.
- Выведите информацию о текущих задачах на экран.
- Выведите общее количество задач на экран.
- Удалите одну из задач и выведите обновленный список задач.
- Выведите общее количество задач после удаления на экран.

Пример использования:

```
tasks = TaskList()
tasks.add_task("Написать отчет", 1)
tasks.add_task("Проверить почту", 3)
tasks.add_task("Подготовить презентацию", 2)
print("Список задач:")
for desc, priority in tasks.tasks:
    print(desc, "(приоритет", priority, ")")
count = tasks.task_count()
print("Всего задач:", count)
tasks.remove_task("Проверить почту")
print("Список задач после удаления 'Проверить почту':")
for desc, priority in tasks.tasks:
    print(desc, "(приоритет", priority, ")")
count = tasks.task_count()
print("Всего задач:", count)
```

Вывод:

```
Список задач:
Написать отчет (приоритет 1 )
Проверить почту (приоритет 3 )
Подготовить презентацию (приоритет 2 )
Всего задач: 3
Список задач после удаления 'Проверить почту':
Написать отчет (приоритет 1 )
Подготовить презентацию (приоритет 2 )
Всего задач: 2
```

7 Написать программу на Python, которая создает класс `BankAccount` для представления банковского счета. Класс должен содержать методы для добавления и снятия средств, а также получения текущего баланса. Программа также должна создавать экземпляр класса `BankAccount`, выполнять операции пополнения и снятия, и выводить информацию о балансе на экран.

- Создайте класс `BankAccount` с методом `__init__`, который инициализирует баланс нулём.
- Создайте метод `deposit`, который принимает сумму и увеличивает баланс на неё.
- Создайте метод `withdraw`, который принимает сумму и уменьшает баланс на неё (если достаточно средств).
- Создайте метод `get_balance`, который возвращает текущий баланс.
- Создайте экземпляр класса `BankAccount`.
- Выполните несколько операций пополнения счета.
- Выведите текущий баланс на экран.
- Выполните операцию снятия средств и выведите обновленный баланс.
- Выведите окончательный баланс на экран.

Пример использования:

```
account = BankAccount()
account.deposit(1000)
account.deposit(500)
print("Баланс после пополнений:", account.get_balance())
account.withdraw(300)
print("Баланс после снятия 300:", account.get_balance())
account.withdraw(200)
print("Окончательный баланс:", account.get_balance())
```

Вывод:

```
Баланс после пополнений: 1500
Баланс после снятия 300: 1200
Окончательный баланс: 1000
```

8 Написать программу на Python, которая создает класс `Library` для представления библиотеки. Класс должен содержать методы для добавления и удаления книг, а также подсчета общего количества книг. Программа также должна создавать экземпляр класса `Library`, добавлять книги, удалять книги и выводить информацию о фонде на экран.

- Создайте класс `Library` с методом `__init__`, который создает пустой список книг.
- Создайте метод `add_book`, который принимает название книги и автора в качестве аргументов и добавляет их в список.
- Создайте метод `remove_book`, который удаляет книгу из списка по её названию.
- Создайте метод `book_count`, который возвращает общее количество книг в библиотеке.
- Создайте экземпляр класса `Library` и добавьте несколько книг.
- Выведите информацию о текущих книгах на экран.
- Выведите общее количество книг на экран.
- Удалите одну из книг и выведите обновленный список.
- Выведите общее количество книг после удаления на экран.

Пример использования:

```
lib = Library()
lib.add_book("1984", "Джордж Оруэлл")
lib.add_book("Гарри Поттер", "Дж.К. Роулинг")
lib.add_book("Гордость и предубеждение", "Джейн Остин")
print("Книги в библиотеке:")
for title, author in lib.books:
    print(title, "-", author)
count = lib.book_count()
print("Всего книг:", count)
lib.remove_book("Гарри Поттер")
print("Книги после удаления 'Гарри Поттера':")
for title, author in lib.books:
    print(title, "-", author)
count = lib.book_count()
print("Всего книг:", count)
```

Вывод:

```
Книги в библиотеке:
1984 - Джордж Оруэлл
Гарри Поттер - Дж.К. Роулинг
Гордость и предубеждение - Джейн Остин
Всего книг: 3
Книги после удаления 'Гарри Поттера':
1984 - Джордж Оруэлл
Гордость и предубеждение - Джейн Остин
Всего книг: 2
```

9 Написать программу на Python, которая создает класс `GroceryList` для представления списка покупок. Класс должен содержать методы для добавления и удаления продуктов, а также подсчета общего количества позиций. Программа также должна создавать экземпляр класса `GroceryList`, добавлять продукты, удалять продукты и выводить информацию о списке на экран.

- Создайте класс `GroceryList` с методом `__init__`, который создает пустой список продуктов.
- Создайте метод `add_product`, который принимает название продукта и количество в качестве аргументов и добавляет их в список.
- Создайте метод `remove_product`, который удаляет продукт из списка по его названию.
- Создайте метод `total_items`, который возвращает общее количество различных продуктов в списке.
- Создайте экземпляр класса `GroceryList` и добавьте несколько продуктов.
- Выведите информацию о текущих продуктах на экран.
- Выведите общее количество позиций на экран.
- Удалите один из продуктов и выведите обновленный список.
- Выведите общее количество позиций после удаления на экран.

Пример использования:

```
grocery = GroceryList()
grocery.add_product("Молоко", 2)
grocery.add_product("Хлеб", 1)
grocery.add_product("Яйца", 12)
print("Список покупок:")
for name, qty in grocery.products:
    print(name, "-", qty)
count = grocery.total_items()
print("Всего позиций:", count)
grocery.remove_product("Хлеб")
print("Список после удаления хлеба:")
for name, qty in grocery.products:
    print(name, "-", qty)
count = grocery.total_items()
print("Всего позиций:", count)
```

Вывод:

```
Список покупок:
Молоко - 2
Хлеб - 1
Яйца - 12
Всего позиций: 3
Список после удаления хлеба:
Молоко - 2
Яйца - 12
Всего позиций: 2
```

10 Написать программу на Python, которая создает класс `ContactList` для представления списка контактов. Класс должен содержать методы для добавления и удаления контактов, а также подсчета общего количества контактов. Программа также должна создавать экземпляр класса `ContactList`, добавлять контакты, удалять контакты и выводить информацию о списке на экран.

- Создайте класс `ContactList` с методом `__init__`, который создает пустой список контактов.
- Создайте метод `add_contact`, который принимает имя и номер телефона в качестве аргументов и добавляет их в список.
- Создайте метод `remove_contact`, который удаляет контакт из списка по имени.
- Создайте метод `contact_count`, который возвращает общее количество контактов в списке.
- Создайте экземпляр класса `ContactList` и добавьте несколько контактов.
- Выведите информацию о текущих контактах на экран.
- Выведите общее количество контактов на экран.
- Удалите один из контактов и выведите обновленный список.
- Выведите общее количество контактов после удаления на экран.

Пример использования:

```
contacts = ContactList()
contacts.add_contact("Анна", "+79001234567")
contacts.add_contact("Борис", "+79007654321")
contacts.add_contact("Виктория", "+79001112233")
print("Контакты:")
for name, phone in contacts.contacts:
    print(name, "-", phone)
count = contacts.contact_count()
print("Всего контактов:", count)
contacts.remove_contact("Борис")
print("Контакты после удаления Бориса:")
for name, phone in contacts.contacts:
    print(name, "-", phone)
count = contacts.contact_count()
print("Всего контактов:", count)
```

Вывод:

```
Контакты:
Анна - +79001234567
Борис - +79007654321
Виктория - +79001112233
Всего контактов: 3
Контакты после удаления Бориса:
Анна - +79001234567
Виктория - +79001112233
Всего контактов: 2
```

11 Написать программу на Python, которая создает класс `MovieCollection` для представления коллекции фильмов. Класс должен содержать методы для добавления и удаления фильмов, а также подсчета общего количества фильмов. Программа также должна создавать экземпляр класса `MovieCollection`, добавлять фильмы, удалять фильмы и выводить информацию о коллекции на экран.

- Создайте класс `MovieCollection` с методом `__init__`, который создает пустой список фильмов.
- Создайте метод `add_movie`, который принимает название фильма и год выпуска в качестве аргументов и добавляет их в список.
- Создайте метод `remove_movie`, который удаляет фильм из списка по его названию.
- Создайте метод `movie_count`, который возвращает общее количество фильмов в коллекции.
- Создайте экземпляр класса `MovieCollection` и добавьте несколько фильмов.
- Выведите информацию о текущих фильмах на экран.
- Выведите общее количество фильмов на экран.
- Удалите один из фильмов и выведите обновленный список.
- Выведите общее количество фильмов после удаления на экран.

Пример использования:

```
movies = MovieCollection()
movies.add_movie("Крёстный отец", 1972)
movies.add_movie("Побег из Шоушенка", 1994)
movies.add_movie("Тёмный рыцарь", 2008)
print("Фильмы в коллекции:")
for title, year in movies.movies:
    print(title, "(", year, ")")
count = movies.movie_count()
print("Всего фильмов:", count)
movies.remove_movie("Побег из Шоушенка")
print("Фильмы после удаления 'Побега из Шоушенка':")
for title, year in movies.movies:
    print(title, "(", year, ")")
count = movies.movie_count()
print("Всего фильмов:", count)
```

Вывод:

```
Фильмы в коллекции:
Крёстный отец ( 1972 )
Побег из Шоушенка ( 1994 )
Тёмный рыцарь ( 2008 )
Всего фильмов: 3
Фильмы после удаления 'Побега из Шоушенка':
Крёстный отец ( 1972 )
Тёмный рыцарь ( 2008 )
Всего фильмов: 2
```

12 Написать программу на Python, которая создает класс `RecipeBook` для представления кулинарной книги. Класс должен содержать методы для добавления и удаления рецептов, а также подсчета общего количества рецептов. Программа также должна создавать экземпляр класса `RecipeBook`, добавлять рецепты, удалять рецепты и выводить информацию о книге на экран.

- Создайте класс `RecipeBook` с методом `__init__`, который создает пустой список рецептов.
- Создайте метод `add_recipe`, который принимает название блюда и время приготовления (в минутах) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_recipe`, который удаляет рецепт из списка по названию блюда.
- Создайте метод `recipe_count`, который возвращает общее количество рецептов в книге.
- Создайте экземпляр класса `RecipeBook` и добавьте несколько рецептов.
- Выведите информацию о текущих рецептах на экран.
- Выведите общее количество рецептов на экран.
- Удалите один из рецептов и выведите обновленный список.
- Выведите общее количество рецептов после удаления на экран.

Пример использования:

```
recipes = RecipeBook()
recipes.add_recipe("Борщ", 60)
recipes.add_recipe("Омлет", 10)
recipes.add_recipe("Паста", 20)
print("Рецепты в книге:")
for dish, time in recipes.recipes:
    print(dish, "-", time, "мин.")
count = recipes.recipe_count()
print("Всего рецептов:", count)
recipes.remove_recipe("Омлет")
print("Рецепты после удаления омлета:")
for dish, time in recipes.recipes:
    print(dish, "-", time, "мин.")
count = recipes.recipe_count()
print("Всего рецептов:", count)
```

Вывод:

```
Рецепты в книге:
Борщ - 60 мин.
Омлет - 10 мин.
Паста - 20 мин.
Всего рецептов: 3
Рецепты после удаления омлета:
Борщ - 60 мин.
Паста - 20 мин.
Всего рецептов: 2
```

13 Написать программу на Python, которая создает класс `CarGarage` для представления автосервиса. Класс должен содержать методы для добавления и удаления автомобилей, а также подсчета общего количества машин. Программа также должна создавать экземпляр класса `CarGarage`, добавлять автомобили, удалять автомобили и выводить информацию о гараже на экран.

- Создайте класс `CarGarage` с методом `__init__`, который создает пустой список автомобилей.
- Создайте метод `add_car`, который принимает марку и модель автомобиля в качестве аргументов и добавляет их в список.
- Создайте метод `remove_car`, который удаляет автомобиль из списка по марке.
- Создайте метод `car_count`, который возвращает общее количество автомобилей в гараже.
- Создайте экземпляр класса `CarGarage` и добавьте несколько автомобилей.
- Выведите информацию о текущих автомобилях на экран.
- Выведите общее количество машин на экран.
- Удалите один из автомобилей и выведите обновленный список.
- Выведите общее количество машин после удаления на экран.

Пример использования:

```
garage = CarGarage()
garage.add_car("Toyota", "Camry")
garage.add_car("BMW", "X5")
garage.add_car("Ford", "Focus")
print("Автомобили в гараже:")
for brand, model in garage.cars:
    print(brand, model)
count = garage.car_count()
print("Всего автомобилей:", count)
garage.remove_car("BMW")
print("Автомобили после удаления BMW:")
for brand, model in garage.cars:
    print(brand, model)
count = garage.car_count()
print("Всего автомобилей:", count)
```

Вывод:

```
Автомобили в гараже:
Toyota Camry
BMW X5
Ford Focus
Всего автомобилей: 3
Автомобили после удаления BMW:
Toyota Camry
Ford Focus
Всего автомобилей: 2
```

14 Написать программу на Python, которая создает класс `PetStore` для представления зоомагазина. Класс должен содержать методы для добавления и удаления животных, а также подсчета общего количества питомцев. Программа также должна создавать экземпляр класса `PetStore`, добавлять животных, удалять животных и выводить информацию о магазине на экран.

- Создайте класс `PetStore` с методом `__init__`, который создает пустой список животных.
- Создайте метод `add_pet`, который принимает вид животного и количество в качестве аргументов и добавляет их в список.
- Создайте метод `remove_pet`, который удаляет животное из списка по виду.
- Создайте метод `total_pets`, который возвращает общее количество всех питомцев в магазине.
- Создайте экземпляр класса `PetStore` и добавьте несколько видов животных.
- Выведите информацию о текущих животных на экран.
- Выведите общее количество питомцев на экран.
- Удалите один из видов животных и выведите обновленный список.
- Выведите общее количество питомцев после удаления на экран.

Пример использования:

```
store = PetStore()
store.add_pet("Кошки", 5)
store.add_pet("Собаки", 3)
store.add_pet("Попугаи", 10)
print("Животные в магазине:")
for species, count in store.pets:
    print(species, "-", count)
total = store.total_pets()
print("Всего питомцев:", total)
store.remove_pet("Собаки")
print("Животные после удаления собак:")
for species, count in store.pets:
    print(species, "-", count)
total = store.total_pets()
print("Всего питомцев:", total)
```

Вывод:

```
Животные в магазине:
Кошки - 5
Собаки - 3
Попугаи - 10
Всего питомцев: 18
Животные после удаления собак:
Кошки - 5
Попугаи - 10
Всего питомцев: 15
```

15 Написать программу на Python, которая создает класс `CourseRoster` для представления списка студентов на курсе. Класс должен содержать методы для добавления и удаления студентов, а также подсчета общего количества учащихся. Программа также должна создавать экземпляр класса `CourseRoster`, добавлять студентов, удалять студентов и выводить информацию о курсе на экран.

- Создайте класс `CourseRoster` с методом `__init__`, который создает пустой список студентов.
- Создайте метод `enroll_student`, который принимает имя студента и его ID в качестве аргументов и добавляет их в список.
- Создайте метод `drop_student`, который удаляет студента из списка по имени.
- Создайте метод `student_count`, который возвращает общее количество студентов на курсе.
- Создайте экземпляр класса `CourseRoster` и добавьте несколько студентов.
- Выведите информацию о текущих студентах на экран.
- Выведите общее количество студентов на экран.
- Удалите одного из студентов и выведите обновленный список.
- Выведите общее количество студентов после удаления на экран.

Пример использования:

```
roster = CourseRoster()
roster.enroll_student("Иван", 101)
roster.enroll_student("Мария", 102)
roster.enroll_student("Алексей", 103)
print("Студенты на курсе:")
for name, sid in roster.students:
    print(name, "(ID:", sid, ")")
count = roster.student_count()
print("Всего студентов:", count)
roster.drop_student("Мария")
print("Студенты после отчисления Марии:")
for name, sid in roster.students:
    print(name, "(ID:", sid, ")")
count = roster.student_count()
print("Всего студентов:", count)
```

Вывод:

```
Студенты на курсе:
Иван (ID: 101 )
Мария (ID: 102 )
Алексей (ID: 103 )
Всего студентов: 3
Студенты после отчисления Марии:
Иван (ID: 101 )
Алексей (ID: 103 )
Всего студентов: 2
```

16 Написать программу на Python, которая создает класс `TravelItinerary` для представления туристического маршрута. Класс должен содержать методы для добавления и удаления мест, а также подсчета общего количества пунктов назначения. Программа также должна создавать экземпляр класса `TravelItinerary`, добавлять места, удалять места и выводить информацию о маршруте на экран.

- Создайте класс `TravelItinerary` с методом `__init__`, который создает пустой список мест.
- Создайте метод `add_destination`, который принимает название города и количество дней пребывания в качестве аргументов и добавляет их в список.
- Создайте метод `remove_destination`, который удаляет место из списка по названию города.
- Создайте метод `destination_count`, который возвращает общее количество пунктов назначения в маршруте.
- Создайте экземпляр класса `TravelItinerary` и добавьте несколько городов.
- Выведите информацию о текущих местах на экран.
- Выведите общее количество пунктов назначения на экран.
- Удалите один из городов и выведите обновленный маршрут.
- Выведите общее количество пунктов назначения после удаления на экран.

Пример использования:

```
itinerary = TravelItinerary()
itinerary.add_destination("Париж", 4)
itinerary.add_destination("Рим", 3)
itinerary.add_destination("Барселона", 5)
print("Маршрут путешествия:")
for city, days in itinerary.destinations:
    print(city, "-", days, "дней")
count = itinerary.destination_count()
print("Всего пунктов:", count)
itinerary.remove_destination("Рим")
print("Маршрут после удаления Рима:")
for city, days in itinerary.destinations:
    print(city, "-", days, "дней")
count = itinerary.destination_count()
print("Всего пунктов:", count)
```

Вывод:

```
Маршрут путешествия:
Париж - 4 дней
Рим - 3 дней
Барселона - 5 дней
Всего пунктов: 3
Маршрут после удаления Рима:
Париж - 4 дней
Барселона - 5 дней
Всего пунктов: 2
```

17 Написать программу на Python, которая создает класс `FitnessTracker` для представления тренировочного плана. Класс должен содержать методы для добавления и удаления упражнений, а также подсчета общего количества подходов. Программа также должна создавать экземпляр класса `FitnessTracker`, добавлять упражнения, удалять упражнения и выводить информацию о плане на экран.

- Создайте класс `FitnessTracker` с методом `__init__`, который создает пустой список упражнений.
- Создайте метод `add_exercise`, который принимает название упражнения и количество подходов в качестве аргументов и добавляет их в список.
- Создайте метод `remove_exercise`, который удаляет упражнение из списка по его названию.
- Создайте метод `total_sets`, который возвращает общее количество подходов по всем упражнениям.
- Создайте экземпляр класса `FitnessTracker` и добавьте несколько упражнений.
- Выведите информацию о текущих упражнениях на экран.
- Выведите общее количество подходов на экран.
- Удалите одно из упражнений и выведите обновленный план.
- Выведите общее количество подходов после удаления на экран.

Пример использования:

```
tracker = FitnessTracker()
tracker.add_exercise("Приседания", 4)
tracker.add_exercise("Отжимания", 3)
tracker.add_exercise("Подтягивания", 5)
print("Тренировочный план:")
for ex, sets in tracker.exercises:
    print(ex, "-", sets, "подходов")
total = tracker.total_sets()
print("Всего подходов:", total)
tracker.remove_exercise("Отжимания")
print("План после удаления отжиманий:")
for ex, sets in tracker.exercises:
    print(ex, "-", sets, "подходов")
total = tracker.total_sets()
print("Всего подходов:", total)
```

Вывод:

```
Тренировочный план:
Приседания - 4 подходов
Отжимания - 3 подходов
Подтягивания - 5 подходов
Всего подходов: 12
План после удаления отжиманий:
Приседания - 4 подходов
Подтягивания - 5 подходов
Всего подходов: 9
```

18 Написать программу на Python, которая создает класс `ExpenseTracker` для представления расходов. Класс должен содержать методы для добавления и удаления трат, а также подсчета общей суммы расходов. Программа также должна создавать экземпляр класса `ExpenseTracker`, добавлять расходы, удалять расходы и выводить информацию о тратах на экран.

- Создайте класс `ExpenseTracker` с методом `__init__`, который создает пустой список расходов.
- Создайте метод `add_expense`, который принимает категорию и сумму в качестве аргументов и добавляет их в список.
- Создайте метод `remove_expense`, который удаляет расход из списка по категории.
- Создайте метод `total_expenses`, который возвращает общую сумму всех расходов.
- Создайте экземпляр класса `ExpenseTracker` и добавьте несколько расходов.
- Выведите информацию о текущих тратах на экран.
- Выведите общую сумму расходов на экран.
- Удалите один из расходов и выведите обновленный список.
- Выведите общую сумму расходов после удаления на экран.

Пример использования:

```
expenses = ExpenseTracker()
expenses.add_expense("Продукты", 2500)
expenses.add_expense("Транспорт", 800)
expenses.add_expense("Развлечения", 1200)
print("Расходы:")
for cat, amount in expenses.expenses:
    print(cat, "-", amount, "руб.")
total = expenses.total_expenses()
print("Общая сумма расходов:", total, "руб.")
expenses.remove_expense("Транспорт")
print("Расходы после удаления транспорта:")
for cat, amount in expenses.expenses:
    print(cat, "-", amount, "руб.")
total = expenses.total_expenses()
print("Общая сумма расходов:", total, "руб.")
```

Вывод:

```
Расходы:
Продукты - 2500 руб.
Транспорт - 800 руб.
Развлечения - 1200 руб.
Общая сумма расходов: 4500 руб.
Расходы после удаления транспорта:
Продукты - 2500 руб.
Развлечения - 1200 руб.
Общая сумма расходов: 3700 руб.
```

19 Написать программу на Python, которая создает класс `ProjectTasks` для представления задач проекта. Класс должен содержать методы для добавления и удаления задач, а также подсчета общего количества задач. Программа также должна создавать экземпляр класса `ProjectTasks`, добавлять задачи, удалять задачи и выводить информацию о проекте на экран.

- Создайте класс `ProjectTasks` с методом `__init__`, который создает пустой список задач.
- Создайте метод `add_task`, который принимает описание задачи и срок выполнения (в днях) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_task`, который удаляет задачу из списка по её описанию.
- Создайте метод `task_count`, который возвращает общее количество задач в проекте.
- Создайте экземпляр класса `ProjectTasks` и добавьте несколько задач.
- Выведите информацию о текущих задачах на экран.
- Выведите общее количество задач на экран.
- Удалите одну из задач и выведите обновленный список.
- Выведите общее количество задач после удаления на экран.

Пример использования:

```
project = ProjectTasks()
project.add_task("Разработка интерфейса", 5)
project.add_task("Тестирование", 3)
project.add_task("Документация", 2)
print("Задачи проекта:")
for desc, days in project.tasks:
    print(desc, "-", days, "дней")
count = project.task_count()
print("Всего задач:", count)
project.remove_task("Тестирование")
print("Задачи после удаления тестирования:")
for desc, days in project.tasks:
    print(desc, "-", days, "дней")
count = project.task_count()
print("Всего задач:", count)
```

Вывод:

```
Задачи проекта:
Разработка интерфейса - 5 дней
Тестирование - 3 дней
Документация - 2 дней
Всего задач: 3
Задачи после удаления тестирования:
Разработка интерфейса - 5 дней
Документация - 2 дней
Всего задач: 2
```

20 Написать программу на Python, которая создает класс `EventSchedule` для представления расписания мероприятий. Класс должен содержать методы для добавления и удаления событий, а также подсчета общего количества мероприятий. Программа также должна создавать экземпляр класса `EventSchedule`, добавлять события, удалять события и выводить информацию о расписании на экран.

- Создайте класс `EventSchedule` с методом `__init__`, который создает пустой список мероприятий.
- Создайте метод `add_event`, который принимает название мероприятия и дату проведения в качестве аргументов и добавляет их в список.
- Создайте метод `remove_event`, который удаляет мероприятие из списка по его названию.
- Создайте метод `event_count`, который возвращает общее количество мероприятий в расписании.
- Создайте экземпляр класса `EventSchedule` и добавьте несколько мероприятий.
- Выведите информацию о текущих мероприятиях на экран.
- Выведите общее количество мероприятий на экран.
- Удалите одно из мероприятий и выведите обновленное расписание.
- Выведите общее количество мероприятий после удаления на экран.

Пример использования:

```
schedule = EventSchedule()
schedule.add_event("Конференция", "15.05.2024")
schedule.add_event("Воркшоп", "20.05.2024")
schedule.add_event("Выставка", "25.05.2024")
print("Расписание мероприятий:")
for name, date in schedule.events:
    print(name, "-", date)
count = schedule.event_count()
print("Всего мероприятий:", count)
schedule.remove_event("Воркшоп")
print("Расписание после удаления воркшопа:")
for name, date in schedule.events:
    print(name, "-", date)
count = schedule.event_count()
print("Всего мероприятий:", count)
```

Вывод:

```
Расписание мероприятий:
Конференция - 15.05.2024
Воркшоп - 20.05.2024
Выставка - 25.05.2024
Всего мероприятий: 3
Расписание после удаления воркшопа:
Конференция - 15.05.2024
Выставка - 25.05.2024
Всего мероприятий: 2
```

21 Написать программу на Python, которая создает класс `GardenPlanner` для представления садового участка. Класс должен содержать методы для добавления и удаления растений, а также подсчета общего количества видов растений. Программа также должна создавать экземпляр класса `GardenPlanner`, добавлять растения, удалять растения и выводить информацию о саде на экран.

- Создайте класс `GardenPlanner` с методом `__init__`, который создает пустой список растений.
- Создайте метод `add_plant`, который принимает название растения и количество экземпляров в качестве аргументов и добавляет их в список.
- Создайте метод `remove_plant`, который удаляет растение из списка по его названию.
- Создайте метод `plant_count`, который возвращает общее количество различных видов растений в саду.
- Создайте экземпляр класса `GardenPlanner` и добавьте несколько растений.
- Выведите информацию о текущих растениях на экран.
- Выведите общее количество видов растений на экран.
- Удалите одно из растений и выведите обновленный список.
- Выведите общее количество видов растений после удаления на экран.

Пример использования:

```
garden = GardenPlanner()
garden.add_plant("Розы", 10)
garden.add_plant("Тюльпаны", 20)
garden.add_plant("Лаванда", 5)
print("Растения в саду:")
for name, qty in garden.plants:
    print(name, "-", qty)
count = garden.plant_count()
print("Всего видов растений:", count)
garden.remove_plant("Тюльпаны")
print("Растения после удаления тюльпанов:")
for name, qty in garden.plants:
    print(name, "-", qty)
count = garden.plant_count()
print("Всего видов растений:", count)
```

Вывод:

```
Растения в саду:
Розы - 10
Тюльпаны - 20
Лаванда - 5
Всего видов растений: 3
Растения после удаления тюльпанов:
Розы - 10
Лаванда - 5
Всего видов растений: 2
```

22 Написать программу на Python, которая создает класс `Warehouse` для представления склада товаров. Класс должен содержать методы для добавления и удаления товаров, а также подсчета общего количества типов товаров. Программа также должна создавать экземпляр класса `Warehouse`, добавлять товары, удалять товары и выводить информацию о складе на экран.

- Создайте класс `Warehouse` с методом `__init__`, который создает пустой список товаров.
- Создайте метод `add_product`, который принимает название товара и количество единиц в качестве аргументов и добавляет их в список.
- Создайте метод `remove_product`, который удаляет товар из списка по его названию.
- Создайте метод `product_types`, который возвращает общее количество различных типов товаров на складе.
- Создайте экземпляр класса `Warehouse` и добавьте несколько товаров.
- Выведите информацию о текущих товарах на экран.
- Выведите общее количество типов товаров на экран.
- Удалите один из товаров и выведите обновленный список.
- Выведите общее количество типов товаров после удаления на экран.

Пример использования:

```
warehouse = Warehouse()
warehouse.add_product("Стулья", 50)
warehouse.add_product("Столы", 20)
warehouse.add_product("Лампы", 100)
print("Товары на складе:")
for name, qty in warehouse.products:
    print(name, "-", qty)
types = warehouse.product_types()
print("Всего типов товаров:", types)
warehouse.remove_product("Столы")
print("Товары после удаления столов:")
for name, qty in warehouse.products:
    print(name, "-", qty)
types = warehouse.product_types()
print("Всего типов товаров:", types)
```

Вывод:

```
Товары на складе:
Стулья - 50
Столы - 20
Лампы - 100
Всего типов товаров: 3
Товары после удаления столов:
Стулья - 50
Лампы - 100
Всего типов товаров: 2
```

23 Написать программу на Python, которая создает класс `GameInventory` для представления инвентаря игрока. Класс должен содержать методы для добавления и удаления предметов, а также подсчета общего количества типов предметов. Программа также должна создавать экземпляр класса `GameInventory`, добавлять предметы, удалять предметы и выводить информацию об инвентаре на экран.

- Создайте класс `GameInventory` с методом `__init__`, который создает пустой список предметов.
- Создайте метод `add_item`, который принимает название предмета и количество в качестве аргументов и добавляет их в список.
- Создайте метод `remove_item`, который удаляет предмет из списка по его названию.
- Создайте метод `item_types`, который возвращает общее количество различных типов предметов в инвентаре.
- Создайте экземпляр класса `GameInventory` и добавьте несколько предметов.
- Выведите информацию о текущих предметах на экран.
- Выведите общее количество типов предметов на экран.
- Удалите один из предметов и выведите обновленный инвентарь.
- Выведите общее количество типов предметов после удаления на экран.

Пример использования:

```
inventory = GameInventory()
inventory.add_item("Меч", 1)
inventory.add_item("Зелье", 5)
inventory.add_item("Щит", 1)
print("Инвентарь игрока:")
for name, qty in inventory.items:
    print(name, "-", qty)
types = inventory.item_types()
print("Всего типов предметов:", types)
inventory.remove_item("Зелье")
print("Инвентарь после удаления зелий:")
for name, qty in inventory.items:
    print(name, "-", qty)
types = inventory.item_types()
print("Всего типов предметов:", types)
```

Вывод:

```
Инвентарь игрока:
Меч - 1
Зелье - 5
Щит - 1
Всего типов предметов: 3
Инвентарь после удаления зелий:
Меч - 1
Щит - 1
Всего типов предметов: 2
```

24 Написать программу на Python, которая создает класс `MusicAlbum` для представления музыкального альбома. Класс должен содержать методы для добавления и удаления треков, а также подсчета общего количества треков. Программа также должна создавать экземпляр класса `MusicAlbum`, добавлять треки, удалять треки и выводить информацию об альбоме на экран.

- Создайте класс `MusicAlbum` с методом `__init__`, который создает пустой список треков.
- Создайте метод `add_track`, который принимает название трека и его длительность (в секундах) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_track`, который удаляет трек из списка по его названию.
- Создайте метод `track_count`, который возвращает общее количество треков в альбоме.
- Создайте экземпляр класса `MusicAlbum` и добавьте несколько треков.
- Выведите информацию о текущих треках на экран.
- Выведите общее количество треков на экран.
- Удалите один из треков и выведите обновленный список.
- Выведите общее количество треков после удаления на экран.

Пример использования:

```
album = MusicAlbum()
album.add_track("Yesterday", 125)
album.add_track("Hey Jude", 431)
album.add_track("Let It Be", 243)
print("Треки в альбоме:")
for name, duration in album.tracks:
    print(name, "-", duration, "сек.")
count = album.track_count()
print("Всего треков:", count)
album.remove_track("Hey Jude")
print("Треки после удаления 'Hey Jude':")
for name, duration in album.tracks:
    print(name, "-", duration, "сек.")
count = album.track_count()
print("Всего треков:", count)
```

Вывод:

```
Треки в альбоме:
Yesterday - 125 сек.
Hey Jude - 431 сек.
Let It Be - 243 сек.
Всего треков: 3
Треки после удаления 'Hey Jude':
Yesterday - 125 сек.
Let It Be - 243 сек.
Всего треков: 2
```

25 Написать программу на Python, которая создает класс `EmployeeRoster` для представления списка сотрудников. Класс должен содержать методы для добавления и удаления сотрудников, а также подсчета общего количества работников. Программа также должна создавать экземпляр класса `EmployeeRoster`, добавлять сотрудников, удалять сотрудников и выводить информацию о персонале на экран.

- Создайте класс `EmployeeRoster` с методом `__init__`, который создает пустой список сотрудников.
- Создайте метод `hire_employee`, который принимает имя сотрудника и его должность в качестве аргументов и добавляет их в список.
- Создайте метод `fire_employee`, который удаляет сотрудника из списка по имени.
- Создайте метод `employee_count`, который возвращает общее количество сотрудников.
- Создайте экземпляр класса `EmployeeRoster` и добавьте несколько сотрудников.
- Выведите информацию о текущих сотрудниках на экран.
- Выведите общее количество работников на экран.
- Удалите одного из сотрудников и выведите обновленный список.
- Выведите общее количество работников после удаления на экран.

Пример использования:

```
roster = EmployeeRoster()
roster.hire_employee("Елена", "Менеджер")
roster.hire_employee("Дмитрий", "Разработчик")
roster.hire_employee("Ольга", "Дизайнер")
print("Сотрудники компании:")
for name, position in roster.employees:
    print(name, "-", position)
count = roster.employee_count()
print("Всего сотрудников:", count)
roster.fire_employee("Дмитрий")
print("Сотрудники после увольнения Дмитрия:")
for name, position in roster.employees:
    print(name, "-", position)
count = roster.employee_count()
print("Всего сотрудников:", count)
```

Вывод:

```
Сотрудники компании:
Елена - Менеджер
Дмитрий - Разработчик
Ольга - Дизайнер
Всего сотрудников: 3
Сотрудники после увольнения Дмитрия:
Елена - Менеджер
Ольга - Дизайнер
Всего сотрудников: 2
```

26 Написать программу на Python, которая создает класс `ShoppingWishlist` для представления списка желаний покупателя. Класс должен содержать методы для добавления и удаления товаров, а также подсчета общего количества позиций. Программа также должна создавать экземпляр класса `ShoppingWishlist`, добавлять товары, удалять товары и выводить информацию о списке желаний на экран.

- Создайте класс `ShoppingWishlist` с методом `__init__`, который создает пустой список товаров.
- Создайте метод `add_item`, который принимает название товара и его приоритет (от 1 до 5) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_item`, который удаляет товар из списка по его названию.
- Создайте метод `item_count`, который возвращает общее количество товаров в списке желаний.
- Создайте экземпляр класса `ShoppingWishlist` и добавьте несколько товаров.
- Выведите информацию о текущих товарах на экран.
- Выведите общее количество позиций на экран.
- Удалите один из товаров и выведите обновленный список.
- Выведите общее количество позиций после удаления на экран.

Пример использования:

```
wishlist = ShoppingWishlist()
wishlist.add_item("Наушники", 5)
wishlist.add_item("Книга", 3)
wishlist.add_item("Флешка", 2)
print("Список желаний:")
for name, priority in wishlist.items:
    print(name, "(приоритет", priority, ")")
count = wishlist.item_count()
print("Всего позиций:", count)
wishlist.remove_item("Книга")
print("Список после удаления книги:")
for name, priority in wishlist.items:
    print(name, "(приоритет", priority, ")")
count = wishlist.item_count()
print("Всего позиций:", count)
```

Вывод:

```
Список желаний:
Наушники (приоритет 5 )
Книга (приоритет 3 )
Флешка (приоритет 2 )
Всего позиций: 3
Список после удаления книги:
Наушники (приоритет 5 )
Флешка (приоритет 2 )
Всего позиций: 2
```

27 Написать программу на Python, которая создает класс `DietPlan` для представления плана питания. Класс должен содержать методы для добавления и удаления блюд, а также подсчета общего количества приемов пищи. Программа также должна создавать экземпляр класса `DietPlan`, добавлять блюда, удалять блюда и выводить информацию о плане на экран.

- Создайте класс `DietPlan` с методом `__init__`, который создает пустой список блюд.
- Создайте метод `add_meal`, который принимает название блюда и количество калорий в качестве аргументов и добавляет их в список.
- Создайте метод `remove_meal`, который удаляет блюдо из списка по его названию.
- Создайте метод `meal_count`, который возвращает общее количество блюд в плане.
- Создайте экземпляр класса `DietPlan` и добавьте несколько блюд.
- Выведите информацию о текущих блюдах на экран.
- Выведите общее количество приемов пищи на экран.
- Удалите одно из блюд и выведите обновленный план.
- Выведите общее количество приемов пищи после удаления на экран.

Пример использования:

```
diet = DietPlan()
diet.add_meal("Овсянка", 300)
diet.add_meal("Салат", 150)
diet.add_meal("Курица", 400)
print("План питания:")
for dish, calories in diet.meals:
    print(dish, "-", calories, "ккал")
count = diet.meal_count()
print("Всего приемов пищи:", count)
diet.remove_meal("Салат")
print("План после удаления салата:")
for dish, calories in diet.meals:
    print(dish, "-", calories, "ккал")
count = diet.meal_count()
print("Всего приемов пищи:", count)
```

Вывод:

```
План питания:
Овсянка - 300 ккал
Салат - 150 ккал
Курица - 400 ккал
Всего приемов пищи: 3
План после удаления салата:
Овсянка - 300 ккал
Курица - 400 ккал
Всего приемов пищи: 2
```

28 Написать программу на Python, которая создает класс `PhotoAlbum` для представления фотоальбома. Класс должен содержать методы для добавления и удаления фотографий, а также подсчета общего количества снимков. Программа также должна создавать экземпляр класса `PhotoAlbum`, добавлять фотографии, удалять фотографии и выводить информацию об альбоме на экран.

- Создайте класс `PhotoAlbum` с методом `__init__`, который создает пустой список фотографий.
- Создайте метод `add_photo`, который принимает название фотографии и дату съемки в качестве аргументов и добавляет их в список.
- Создайте метод `remove_photo`, который удаляет фотографию из списка по её названию.
- Создайте метод `photo_count`, который возвращает общее количество фотографий в альбоме.
- Создайте экземпляр класса `PhotoAlbum` и добавьте несколько фотографий.
- Выведите информацию о текущих фотографиях на экран.
- Выведите общее количество снимков на экран.
- Удалите одну из фотографий и выведите обновленный альбом.
- Выведите общее количество снимков после удаления на экран.

Пример использования:

```
album = PhotoAlbum()
album.add_photo("Пляж", "2023-07-15")
album.add_photo("Горы", "2023-08-20")
album.add_photo("Семья", "2023-12-25")
print("Фотографии в альбоме:")
for name, date in album.photos:
    print(name, "-", date)
count = album.photo_count()
print("Всего фотографий:", count)
album.remove_photo("Горы")
print("Фотографии после удаления 'Горы':")
for name, date in album.photos:
    print(name, "-", date)
count = album.photo_count()
print("Всего фотографий:", count)
```

Вывод:

```
Фотографии в альбоме:
Пляж - 2023-07-15
Горы - 2023-08-20
Семья - 2023-12-25
Всего фотографий: 3
Фотографии после удаления 'Горы':
Пляж - 2023-07-15
Семья - 2023-12-25
Всего фотографий: 2
```

29 Написать программу на Python, которая создает класс `StudyMaterials` для представления учебных материалов. Класс должен содержать методы для добавления и удаления материалов, а также подсчета общего количества ресурсов. Программа также должна создавать экземпляр класса `StudyMaterials`, добавлять материалы, удалять материалы и выводить информацию о ресурсах на экран.

- Создайте класс `StudyMaterials` с методом `__init__`, который создает пустой список материалов.
- Создайте метод `add_material`, который принимает название материала и тип (например, "книга "видео "статья") в качестве аргументов и добавляет их в список.
- Создайте метод `remove_material`, который удаляет материал из списка по его названию.
- Создайте метод `material_count`, который возвращает общее количество учебных материалов.
- Создайте экземпляр класса `StudyMaterials` и добавьте несколько материалов.
- Выведите информацию о текущих материалах на экран.
- Выведите общее количество ресурсов на экран.
- Удалите один из материалов и выведите обновленный список.
- Выведите общее количество ресурсов после удаления на экран.

Пример использования:

```
materials = StudyMaterials()
materials.add_material("Алгоритмы", "книга")
materials.add_material("Python для начинающих", "видео")
materials.add_material("Структуры данных", "статья")
print("Учебные материалы:")
for name, mtype in materials.materials:
    print(name, "-", mtype)
count = materials.material_count()
print("Всего материалов:", count)
materials.remove_material("Python для начинающих")
print("Материалы после удаления видео:")
for name, mtype in materials.materials:
    print(name, "-", mtype)
count = materials.material_count()
print("Всего материалов:", count)
```

Вывод:

```
Учебные материалы:
Алгоритмы - книга
Python для начинающих - видео
Структуры данных - статья
Всего материалов: 3
Материалы после удаления видео:
Алгоритмы - книга
Структуры данных - статья
Всего материалов: 2
```

30 Написать программу на Python, которая создает класс `ArtCollection` для представления коллекции произведений искусства. Класс должен содержать методы для добавления и удаления работ, а также подсчета общего количества экспонатов. Программа также должна создавать экземпляр класса `ArtCollection`, добавлять работы, удалять работы и выводить информацию о коллекции на экран.

- Создайте класс `ArtCollection` с методом `__init__`, который создает пустой список произведений.
- Создайте метод `add_artwork`, который принимает название работы и имя художника в качестве аргументов и добавляет их в список.
- Создайте метод `remove_artwork`, который удаляет работу из списка по её названию.
- Создайте метод `artwork_count`, который возвращает общее количество произведений в коллекции.
- Создайте экземпляр класса `ArtCollection` и добавьте несколько работ.
- Выведите информацию о текущих произведениях на экран.
- Выведите общее количество экспонатов на экран.
- Удалите одну из работ и выведите обновленный список.
- Выведите общее количество экспонатов после удаления на экран.

Пример использования:

```
art = ArtCollection()
art.add_artwork("Звёздная ночь", "Ван Гог")
art.add_artwork("Мона Лиза", "Леонардо да Винчи")
art.add_artwork("Крик", "Мунк")
print("Произведения в коллекции:")
for title, artist in art.artworks:
    print(title, "-", artist)
count = art.artwork_count()
print("Всего экспонатов:", count)
art.remove_artwork("Мона Лиза")
print("Произведения после удаления 'Моны Лизы':")
for title, artist in art.artworks:
    print(title, "-", artist)
count = art.artwork_count()
print("Всего экспонатов:", count)
```

Вывод:

```
Произведения в коллекции:
Звёздная ночь - Ван Гог
Мона Лиза - Леонардо да Винчи
Крик - Мунк
Всего экспонатов: 3
Произведения после удаления 'Моны Лизы':
Звёздная ночь - Ван Гог
Крик - Мунк
Всего экспонатов: 2
```

31 Написать программу на Python, которая создает класс `FlightSchedule` для представления расписания рейсов. Класс должен содержать методы для добавления и удаления рейсов, а также подсчета общего количества перелетов. Программа также должна создавать экземпляр класса `FlightSchedule`, добавлять рейсы, удалять рейсы и выводить информацию о расписании на экран.

- Создайте класс `FlightSchedule` с методом `__init__`, который создает пустой список рейсов.
- Создайте метод `add_flight`, который принимает номер рейса и пункт назначения в качестве аргументов и добавляет их в список.
- Создайте метод `remove_flight`, который удаляет рейс из списка по его номеру.
- Создайте метод `flight_count`, который возвращает общее количество рейсов в расписании.
- Создайте экземпляр класса `FlightSchedule` и добавьте несколько рейсов.
- Выведите информацию о текущих рейсах на экран.
- Выведите общее количество перелетов на экран.
- Удалите один из рейсов и выведите обновленное расписание.
- Выведите общее количество перелетов после удаления на экран.

Пример использования:

```
flights = FlightSchedule()
flights.add_flight("SU123", "Париж")
flights.add_flight("SU456", "Лондон")
flights.add_flight("SU789", "Рим")
print("Рейсы:")
for num, dest in flights.flights:
    print(num, "-", dest)
count = flights.flight_count()
print("Всего рейсов:", count)
flights.remove_flight("SU456")
print("Рейсы после отмены SU456:")
for num, dest in flights.flights:
    print(num, "-", dest)
count = flights.flight_count()
print("Всего рейсов:", count)
```

Вывод:

```
Рейсы:
SU123 - Париж
SU456 - Лондон
SU789 - Рим
Всего рейсов: 3
Рейсы после отмены SU456:
SU123 - Париж
SU789 - Рим
Всего рейсов: 2
```

32 Написать программу на Python, которая создает класс `RecipeIngredients` для представления ингредиентов рецепта. Класс должен содержать методы для добавления и удаления ингредиентов, а также подсчета общего количества компонентов. Программа также должна создавать экземпляр класса `RecipeIngredients`, добавлять ингредиенты, удалять ингредиенты и выводить информацию о рецепте на экран.

- Создайте класс `RecipeIngredients` с методом `__init__`, который создает пустой список ингредиентов.
- Создайте метод `add_ingredient`, который принимает название ингредиента и количество (в граммах или штуках) в качестве аргументов и добавляет их в список.
- Создайте метод `remove_ingredient`, который удаляет ингредиент из списка по его названию.
- Создайте метод `ingredient_count`, который возвращает общее количество ингредиентов в рецепте.
- Создайте экземпляр класса `RecipeIngredients` и добавьте несколько ингредиентов.
- Выведите информацию о текущих ингредиентах на экран.
- Выведите общее количество компонентов на экран.
- Удалите один из ингредиентов и выведите обновленный список.
- Выведите общее количество компонентов после удаления на экран.

Пример использования:

```
recipe = RecipeIngredients()
recipe.add_ingredient("Мука", 200)
recipe.add_ingredient("Сахар", 100)
recipe.add_ingredient("Яйца", 2)
print("Ингредиенты рецепта:")
for name, qty in recipe.ingredients:
    print(name, "-", qty)
count = recipe.ingredient_count()
print("Всего ингредиентов:", count)
recipe.remove_ingredient("Сахар")
print("Ингредиенты после удаления сахара:")
for name, qty in recipe.ingredients:
    print(name, "-", qty)
count = recipe.ingredient_count()
print("Всего ингредиентов:", count)
```

Вывод:

```
Ингредиенты рецепта:
Мука - 200
Сахар - 100
Яйца - 2
Всего ингредиентов: 3
Ингредиенты после удаления сахара:
```

```
Мука - 200
Яйца - 2
Всего ингредиентов: 2
```

33 Написать программу на Python, которая создает класс `WorkoutPlan` для представления плана тренировок. Класс должен содержать методы для добавления и удаления упражнений, а также подсчета общего количества упражнений. Программа также должна создавать экземпляр класса `WorkoutPlan`, добавлять упражнения, удалять упражнения и выводить информацию о плане на экран.

- Создайте класс `WorkoutPlan` с методом `__init__`, который создает пустой список упражнений.
- Создайте метод `add_exercise`, который принимает название упражнения и количество повторений в качестве аргументов и добавляет их в список.
- Создайте метод `remove_exercise`, который удаляет упражнение из списка по его названию.
- Создайте метод `exercise_count`, который возвращает общее количество упражнений в плане.
- Создайте экземпляр класса `WorkoutPlan` и добавьте несколько упражнений.
- Выведите информацию о текущих упражнениях на экран.
- Выведите общее количество упражнений на экран.
- Удалите одно из упражнений и выведите обновленный план.
- Выведите общее количество упражнений после удаления на экран.

Пример использования:

```
workout = WorkoutPlan()
workout.add_exercise("Бег", 30)
workout.add_exercise("Планка", 3)
workout.add_exercise("Приседания", 20)
print("План тренировки:")
for name, reps in workout.exercises:
    print(name, "-", reps)
count = workout.exercise_count()
print("Всего упражнений:", count)
workout.remove_exercise("Планка")
print("План после удаления планки:")
for name, reps in workout.exercises:
    print(name, "-", reps)
count = workout.exercise_count()
print("Всего упражнений:", count)
```

Вывод:

```
План тренировки:
Бег - 30
Планка - 3
```

```
Приседания - 20
Всего упражнений: 3
План после удаления планки:
Бег - 30
Приседания - 20
Всего упражнений: 2
```

34 Написать программу на Python, которая создает класс `InventoryManager` для представления управления запасами. Класс должен содержать методы для добавления и удаления товаров, а также подсчета общего количества типов товаров. Программа также должна создавать экземпляр класса `InventoryManager`, добавлять товары, удалять товары и выводить информацию о запасах на экран.

- Создайте класс `InventoryManager` с методом `__init__`, который создает пустой список товаров.
- Создайте метод `add_product`, который принимает название товара и количество единиц в качестве аргументов и добавляет их в список.
- Создайте метод `remove_product`, который удаляет товар из списка по его названию.
- Создайте метод `product_types`, который возвращает общее количество различных типов товаров.
- Создайте экземпляр класса `InventoryManager` и добавьте несколько товаров.
- Выведите информацию о текущих товарах на экран.
- Выведите общее количество типов товаров на экран.
- Удалите один из товаров и выведите обновленный список.
- Выведите общее количество типов товаров после удаления на экран.

Пример использования:

```
inv = InventoryManager()
inv.add_product("Мыло", 100)
inv.add_product("Шампунь", 50)
inv.add_product("Зубная паста", 75)
print("Товары на складе:")
for name, qty in inv.products:
    print(name, "-", qty)
types = inv.product_types()
print("Всего типов товаров:", types)
inv.remove_product("Шампунь")
print("Товары после удаления шампуня:")
for name, qty in inv.products:
    print(name, "-", qty)
types = inv.product_types()
print("Всего типов товаров:", types)
```

Вывод:

```
Товары на складе:
Мыло - 100
Шампунь - 50
Зубная паста - 75
Всего типов товаров: 3
Товары после удаления шампуня:
Мыло - 100
Зубная паста - 75
Всего типов товаров: 2
```

35 Написать программу на Python, которая создает класс `EventGuestList` для представления списка гостей мероприятия. Класс должен содержать методы для добавления и удаления гостей, а также подсчета общего количества приглашенных. Программа также должна создавать экземпляр класса `EventGuestList`, добавлять гостей, удалять гостей и выводить информацию о списке на экран.

- Создайте класс `EventGuestList` с методом `__init__`, который создает пустой список гостей.
- Создайте метод `add_guest`, который принимает имя гостя и его статус (например, "подтвержден "ожидает") в качестве аргументов и добавляет их в список.
- Создайте метод `remove_guest`, который удаляет гостя из списка по имени.
- Создайте метод `guest_count`, который возвращает общее количество гостей в списке.
- Создайте экземпляр класса `EventGuestList` и добавьте несколько гостей.
- Выведите информацию о текущих гостях на экран.
- Выведите общее количество приглашенных на экран.
- Удалите одного из гостей и выведите обновленный список.
- Выведите общее количество приглашенных после удаления на экран.

Пример использования:

```
guests = EventGuestList()
guests.add_guest("Андрей", "подтвержден")
guests.add_guest("Светлана", "ожидает")
guests.add_guest("Михаил", "подтвержден")
print("Список гостей:")
for name, status in guests.guests:
    print(name, "-", status)
count = guests.guest_count()
print("Всего гостей:", count)
guests.remove_guest("Светлана")
print("Список после отмены Светланы:")
for name, status in guests.guests:
    print(name, "-", status)
count = guests.guest_count()
print("Всего гостей:", count)
```

Вывод:

```
Список гостей:
Андрей - подтвержден
Светлана - ожидает
Михаил - подтвержден
Всего гостей: 3
Список после отмены Светланы:
Андрей - подтвержден
Михаил - подтвержден
Всего гостей: 2
```

2.4.5 Задача 5

1 Написать программу на Python, которая создает класс `Bank`, представляющий банк. Класс должен содержать методы для создания учетных записей клиентов, внесения депозитов, снятия средств и проверки баланса. Программа также должна создавать экземпляр класса `Bank`, создавать учетные записи клиентов, вносить депозиты, снимать средства и проверять баланс.

- Создайте класс `Bank` с методом `__init__`, который создает пустой словарь клиентов.
- Создайте метод `create_account`, который принимает номер счета и начальный баланс в качестве аргументов. Метод должен проверять, существует ли уже номер счета в словаре клиентов. Если это так, он должен выводить сообщение об ошибке. В противном случае, он должен добавить номер счета в словарь клиентов с начальным балансом в качестве значения.
- Создайте метод `make_deposit`, который принимает номер счета и сумму в качестве аргументов. Метод должен проверять, существует ли номер счета в словаре клиентов. Если это так, он должен добавить сумму к текущему балансу счета. Если номер счета не существует, он должен вывести сообщение об ошибке.
- Создайте метод `make_withdrawal`, который принимает номер счета и сумму в качестве аргументов. Метод должен проверять, существует ли номер счета в словаре клиентов. Если это так, он должен проверить, достаточно ли средств на счете для снятия. Если это так, он должен вычесть сумму из текущего баланса счета. В противном случае, он должен вывести сообщение об ошибке, указывающее на недостаточность средств. Если номер счета не существует, он должен вывести сообщение об ошибке.
- Создайте метод `check_balance`, который принимает номер счета в качестве аргумента. Метод должен проверять, существует ли номер счета в словаре клиентов. Если это так, он должен извлечь и вывести текущий баланс счета. Если номер счета не существует, он должен вывести сообщение об ошибке.
- Создайте экземпляр класса `Bank` и создайте учетные записи клиентов.
- Вносите депозиты на счета клиентов.
- Снимайте средства со счетов клиентов.
- Проверяйте баланс счетов клиентов.

Пример использования:

```
bank = Bank()
acno1 = "SB-123"
damt1 = 1000
print("Новый номер счета: ", acno1, " Внесенная сумма: ", damt1)
bank.create_account(acno1, damt1)
acno2 = "SB-124"
damt2 = 1500
print("Новый номер счета: ", acno2, " Внесенная сумма: ", damt2)
bank.create_account(acno2, damt2)
wamt1 = 600
print("\nДепозит средств: ", wamt1, " на счет № ", acno1)
bank.make_deposit(acno1, wamt1)
wamt2 = 350
print("Вывод средств: ", wamt2, " со счета № ", acno2)
bank.make_withdrawal(acno2, wamt2)
print("Номер расчетного счета: ", acno1)
bank.check_balance(acno1)
print("Номер расчетного счета: ", acno2)
bank.check_balance(acno2)
wamt3 = 1200
print("Вывод средств: ", wamt3, " со счета № ", acno2)
bank.make_withdrawal(acno2, wamt3)
acno3 = "SB-134"
print("Проверка баланса счета № ", acno3)
bank.check_balance(acno3)  # Non-existent account number
```

2 Написать программу на Python, которая создает класс `CreditUnion`, представляющий кредитный союз. Класс должен содержать методы для открытия счетов участников, пополнения баланса, снятия денег и запроса текущего состояния счета. Программа также должна создавать экземпляр класса `CreditUnion`, открывать счета, выполнять операции и проверять балансы.

- Создайте класс `CreditUnion` с методом `__init__`, инициализирующим пустой словарь счетов.
- Создайте метод `open_account`, принимающий идентификатор счета и стартовый остаток. Если счет уже существует, выведите ошибку; иначе — добавьте запись.
- Создайте метод `deposit`, принимающий идентификатор счета и сумму. Если счет существует, увеличьте баланс; иначе — сообщите об ошибке.
- Создайте метод `withdraw`, принимающий идентификатор счета и сумму. Если счет существует и средств достаточно, уменьшите баланс; иначе — выведите соответствующую ошибку.
- Создайте метод `get_balance`, принимающий идентификатор счета. Если счет существует, выведите его баланс; иначе — сообщите об ошибке.
- Создайте экземпляр `CreditUnion`.
- Откройте несколько счетов.
- Выполните пополнения.
- Выполните снятия.

- Проверьте балансы.

Пример использования:

```
cu = CreditUnion()
cu.open_account("CU-001", 2000)
cu.open_account("CU-002", 500)
cu.deposit("CU-001", 300)
cu.withdraw("CU-002", 200)
cu.get_balance("CU-001")
cu.get_balance("CU-002")
cu.withdraw("CU-002", 400)  # недостаточно средств
cu.get_balance("CU-999")     # несуществующий счет
```

3 Написать программу на Python, которая создает класс `SavingsBank`, моделирующий сберегательный банк. Класс должен поддерживать создание счетов, внесение вкладов, снятие средств и проверку баланса. Программа должна демонстрировать работу всех методов на примере нескольких счетов.

- Создайте класс `SavingsBank` с методом `__init__`, инициализирующим пустой словарь `accounts`.
- Метод `add_account` принимает номер счета и начальный депозит. Если счет уже есть — ошибка; иначе — добавление.
- Метод `credit` принимает номер счета и сумму. При наличии счета — пополнение; иначе — ошибка.
- Метод `debit` принимает номер счета и сумму. При наличии счета и достаточном балансе — снятие; иначе — ошибка.
- Метод `show_balance` принимает номер счета и выводит баланс или сообщение об ошибке.
- Создайте экземпляр `SavingsBank`.
- Добавьте два счета.
- Пополните один из них.
- Снимите средства с другого.
- Проверьте балансы обоих и несуществующего счета.

Пример использования:

```
sb = SavingsBank()
sb.add_account("SAV-101", 1000)
sb.add_account("SAV-102", 800)
sb.credit("SAV-101", 200)
sb.debit("SAV-102", 300)
sb.show_balance("SAV-101")
sb.show_balance("SAV-102")
sb.debit("SAV-102", 600)  # недостаточно
sb.show_balance("SAV-999") # несуществует
```

4 Написать программу на Python, которая создает класс `DigitalWallet`, представляющий цифровой кошелек. Класс должен поддерживать регистрацию кошельков, пополнение, списание и проверку баланса.

- Создайте класс `DigitalWallet` с методом `__init__`, создающим пустой словарь `wallets`.
- Метод `register_wallet` принимает ID кошелька и начальный баланс. Если ID занят — ошибка; иначе — регистрация.
- Метод `top_up` принимает ID и сумму. При существовании кошелька — пополнение; иначе — ошибка.
- Метод `spend` принимает ID и сумму. При наличии кошелька и достаточном балансе — списание; иначе — ошибка.
- Метод `get_wallet_balance` принимает ID и выводит баланс или сообщение об ошибке.
- Создайте экземпляр `DigitalWallet`.
- Зарегистрируйте два кошелька.
- Пополните один.
- Потратьте с другого.
- Проверьте балансы и попытайтесь проверить несуществующий.

Пример использования:

```
dw = DigitalWallet()
dw.register_wallet("WAL-01", 500)
dw.register_wallet("WAL-02", 300)
dw.top_up("WAL-01", 100)
dw.spend("WAL-02", 150)
dw.get_wallet_balance("WAL-01")
dw.get_wallet_balance("WAL-02")
dw.spend("WAL-02", 200)  # недостаточно
dw.get_wallet_balance("WAL-99")  # несуществует
```

5 Написать программу на Python, которая создает класс `PaymentSystem`, моделирующий систему платежей. Класс должен поддерживать создание счетов, зачисление средств, списание и проверку баланса.

- Создайте класс `PaymentSystem` с методом `__init__`, инициализирующим пустой словарь `accounts`.
- Метод `create_user_account` принимает идентификатор и начальный баланс. Если уже есть — ошибка; иначе — создание.
- Метод `credit_account` принимает ID и сумму. При наличии счета — зачисление; иначе — ошибка.
- Метод `debit_account` принимает ID и сумму. При наличии счета и достаточном балансе — списание; иначе — ошибка.
- Метод `check_account_balance` принимает ID и выводит баланс или ошибку.

- Создайте экземпляр `PaymentSystem`.
- Создайте два счета.
- Зачислите средства на один.
- Спишите с другого.
- Проверьте балансы и несуществующий счет.

Пример использования:

```
ps = PaymentSystem()
ps.create_user_account("USR-1", 1200)
ps.create_user_account("USR-2", 700)
ps.credit_account("USR-1", 300)
ps.debit_account("USR-2", 200)
ps.check_account_balance("USR-1")
ps.check_account_balance("USR-2")
ps.debit_account("USR-2", 600)  # недостаточно
ps.check_account_balance("USR-999")  # несуществует
```

6 Написать программу на Python, которая создает класс `MicroFinance`, представляющий микрофинансовую организацию. Класс должен поддерживать открытие счетов, пополнение, снятие и проверку баланса.

- Создайте класс `MicroFinance` с методом `__init__`, создающим пустой словарь `clients`.
- Метод `open_client_account` принимает номер счета и стартовый баланс. Если счет существует — ошибка; иначе — открытие.
- Метод `fund_account` принимает номер счета и сумму. При наличии счета — пополнение; иначе — ошибка.
- Метод `withdraw_funds` принимает номер счета и сумму. При наличии счета и достаточном балансе — снятие; иначе — ошибка.
- Метод `view_balance` принимает номер счета и выводит баланс или сообщение об ошибке.
- Создайте экземпляр `MicroFinance`.
- Откройте два счета.
- Пополните один.
- Снимите с другого.
- Проверьте балансы и несуществующий счет.

Пример использования:

```
mf = MicroFinance()
mf.open_client_account("MF-201", 900)
mf.open_client_account("MF-202", 400)
mf.fund_account("MF-201", 100)
mf.withdraw_funds("MF-202", 150)
```

```
mf.view_balance("MF-201")
mf.view_balance("MF-202")
mf.withdraw_funds("MF-202", 300)  # недостаточно
mf.view_balance("MF-999")  # несуществует
```

7 Написать программу на Python, которая создает класс `OnlineBank`, моделирующий онлайн-банк. Класс должен поддерживать регистрацию счетов, депозиты, выводы и проверку баланса.

- Создайте класс `OnlineBank` с методом `__init__`, инициализирующим пустой словарь `accounts`.
- Метод `register_account` принимает ID и начальный баланс. Если ID занят — ошибка; иначе — регистрация.
- Метод `deposit_funds` принимает ID и сумму. При наличии счета — пополнение; иначе — ошибка.
- Метод `withdraw_funds` принимает ID и сумму. При наличии счета и достаточном балансе — снятие; иначе — ошибка.
- Метод `check_current_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `OnlineBank`.
- Зарегистрируйте два счета.
- Пополните один.
- Снимите с другого.
- Проверьте балансы и несуществующий счет.

Пример использования:

```
ob = OnlineBank()
ob.register_account("ONB-501", 1500)
ob.register_account("ONB-502", 600)
ob.deposit_funds("ONB-501", 200)
ob.withdraw_funds("ONB-502", 250)
ob.check_current_balance("ONB-501")
ob.check_current_balance("ONB-502")
ob.withdraw_funds("ONB-502", 400)  # недостаточно
ob.check_current_balance("ONB-999")  # несуществует
```

8 Написать программу на Python, которая создает класс `FinTechApp`, представляющий финтех-приложение. Класс должен поддерживать создание аккаунтов, пополнение, снятие и проверку баланса.

- Создайте класс `FinTechApp` с методом `__init__`, создающим пустой словарь `users`.
- Метод `create_user` принимает логин и начальный баланс. Если логин занят — ошибка; иначе — создание.
- Метод `add_money` принимает логин и сумму. При наличии аккаунта — пополнение; иначе — ошибка.

- Метод `remove_money` принимает логин и сумму. При наличии аккаунта и достаточном балансе — снятие; иначе — ошибка.
- Метод `get_user_balance` принимает логин и выводит баланс или ошибку.
- Создайте экземпляр `FinTechApp`.
- Создайте двух пользователей.
- Пополните одного.
- Снимите у другого.
- Проверьте балансы и несуществующего пользователя.

Пример использования:

```
ft = FinTechApp()
ft.create_user("alice", 2000)
ft.create_user("bob", 800)
ft.add_money("alice", 300)
ft.remove_money("bob", 200)
ft.get_user_balance("alice")
ft.get_user_balance("bob")
ft.remove_money("bob", 700)  # недостаточно
ft.get_user_balance("charlie")  # несуществует
```

9 Написать программу на Python, которая создает класс `CryptoWallet`, моделирующий криптовалютный кошелек. Класс должен поддерживать создание кошельков, пополнение, перевод и проверку баланса.

- Создайте класс `CryptoWallet` с методом `__init__`, инициализирующим пустой словарь `wallets`.
- Метод `generate_wallet` принимает адрес и начальный баланс. Если адрес уже есть — ошибка; иначе — создание.
- Метод `receive_coins` принимает адрес и сумму. При наличии кошелька — пополнение; иначе — ошибка.
- Метод `send_coins` принимает адрес и сумму. При наличии кошелька и достаточном балансе — списание; иначе — ошибка.
- Метод `check_wallet_balance` принимает адрес и выводит баланс или ошибку.
- Создайте экземпляр `CryptoWallet`.
- Создайте два кошелька.
- Пополните один.
- Отправьте с другого.
- Проверьте балансы и несуществующий адрес.

Пример использования:

```
cw = CryptoWallet()
cw.generate_wallet("0x1a2b", 10.5)
cw.generate_wallet("0x3c4d", 5.0)
cw.receive_coins("0x1a2b", 2.0)
cw.send_coins("0x3c4d", 1.5)
cw.check_wallet_balance("0x1a2b")
cw.check_wallet_balance("0x3c4d")
cw.send_coins("0x3c4d", 4.0)  # недостаточно
cw.check_wallet_balance("0x9999")  # несуществует
```

10 Написать программу на Python, которая создает класс `StudentFund`, представляющий студенческий фонд. Класс должен поддерживать создание счетов студентов, внесение средств, снятие и проверку баланса.

- Создайте класс `StudentFund` с методом `__init__`, создающим пустой словарь `students`.
- Метод `enroll_student` принимает ID студента и начальный грант. Если ID уже есть — ошибка; иначе — зачисление.
- Метод `add_grant` принимает ID и сумму. При наличии студента — пополнение; иначе — ошибка.
- Метод `use_funds` принимает ID и сумму. При наличии студента и достаточном балансе — списание; иначе — ошибка.
- Метод `view_student_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `StudentFund`.
- Зачислите двух студентов.
- Пополните одного.
- Снимите у другого.
- Проверьте балансы и несуществующего студента.

Пример использования:

```
sf = StudentFund()
sf.enroll_student("STU-01", 5000)
sf.enroll_student("STU-02", 3000)
sf.add_grant("STU-01", 1000)
sf.use_funds("STU-02", 800)
sf.view_student_balance("STU-01")
sf.view_student_balance("STU-02")
sf.use_funds("STU-02", 2500)  # недостаточно
sf.view_student_balance("STU-99")  # несуществует
```

11 Написать программу на Python, которая создает класс `GameCurrency`, моделирующий внутриигровую валюту. Класс должен поддерживать создание аккаунтов игроков, начисление монет, трату и проверку баланса.

- Создайте класс `GameCurrency` с методом `__init__`, инициализирующим пустой словарь `players`.

- Метод `create_player` принимает ник и начальный баланс. Если ник занят — ошибка; иначе — создание.
- Метод `award_coins` принимает ник и сумму. При наличии игрока — начисление; иначе — ошибка.
- Метод `spend_coins` принимает ник и сумму. При наличии игрока и достаточном балансе — списание; иначе — ошибка.
- Метод `get_player_balance` принимает ник и выводит баланс или ошибку.
- Создайте экземпляр `GameCurrency`.
- Создайте двух игроков.
- Начислите одному.
- Потратьте у другого.
- Проверьте балансы и несуществующего игрока.

Пример использования:

```
gc = GameCurrency()
gc.create_player("hero1", 100)
gc.create_player("hero2", 75)
gc.award_coins("hero1", 25)
gc.spend_coins("hero2", 30)
gc.get_player_balance("hero1")
gc.get_player_balance("hero2")
gc.spend_coins("hero2", 50)  # недостаточно
gc.get_player_balance("hero99")  # несуществует
```

12 Написать программу на Python, которая создает класс `CharityFund`, представляющий благотворительный фонд. Класс должен поддерживать создание счетов доноров, получение пожертвований, выдачу средств и проверку баланса.

- Создайте класс `CharityFund` с методом `__init__`, создающим пустой словарь `donors`.
- Метод `register_donor` принимает ID и начальный взнос. Если ID есть — ошибка; иначе — регистрация.
- Метод `accept_donation` принимает ID и сумму. При наличии донора — пополнение; иначе — ошибка.
- Метод `distribute_funds` принимает ID и сумму. При наличии донора и достаточном балансе — списание; иначе — ошибка.
- Метод `check_donor_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `CharityFund`.
- Зарегистрируйте двух доноров.
- Примите пожертвование от одного.
- Распределите средства от другого.
- Проверьте балансы и несуществующего донора.

Пример использования:

```
cf = CharityFund()
cf.register_donor("DON-1", 2000)
cf.register_donor("DON-2", 1500)
cf.accept_donation("DON-1", 500)
cf.distribute_funds("DON-2", 600)
cf.check_donor_balance("DON-1")
cf.check_donor_balance("DON-2")
cf.distribute_funds("DON-2", 1000)  # недостаточно
cf.check_donor_balance("DON-99")  # несуществует
```

13 Написать программу на Python, которая создает класс `TravelWallet`, моделирующий кошелек для путешествий. Класс должен поддерживать создание профилей, пополнение, оплату и проверку баланса.

- Создайте класс `TravelWallet` с методом `__init__`, инициализирующим пустой словарь `profiles`.
- Метод `create_profile` принимает имя профиля и начальный бюджет. Если профиль существует — ошибка; иначе — создание.
- Метод `load_funds` принимает имя профиля и сумму. При наличии профиля — пополнение; иначе — ошибка.
- Метод `pay_expense` принимает имя профиля и сумму. При наличии профиля и достаточном балансе — списание; иначе — ошибка.
- Метод `check_budget` принимает имя профиля и выводит баланс или ошибку.
- Создайте экземпляр `TravelWallet`.
- Создайте два профиля.
- Пополните один.
- Оплатите по другому.
- Проверьте балансы и несуществующий профиль.

Пример использования:

```
tw = TravelWallet()
tw.create_profile("ParisTrip", 3000)
tw.create_profile("TokyoTrip", 2500)
tw.load_funds("ParisTrip", 500)
tw.pay_expense("TokyoTrip", 700)
tw.check_budget("ParisTrip")
tw.check_budget("TokyoTrip")
tw.pay_expense("TokyoTrip", 2000)  # недостаточно
tw.check_budget("LondonTrip")  # несуществует
```

14 Написать программу на Python, которая создает класс `SchoolFund`, представляющий школьный фонд. Класс должен поддерживать создание счетов классов, внесение средств, расход и проверку баланса.

- Создайте класс `SchoolFund` с методом `__init__`, создающим пустой словарь `classes`.

- Метод `add_class` принимает номер класса и начальный бюджет. Если класс уже есть — ошибка; иначе — добавление.
- Метод `collect_money` принимает номер класса и сумму. При наличии класса — пополнение; иначе — ошибка.
- Метод `spend_money` принимает номер класса и сумму. При наличии класса и достаточном бюджете — списание; иначе — ошибка.
- Метод `get_class_balance` принимает номер класса и выводит баланс или ошибку.
- Создайте экземпляр `SchoolFund`.
- Добавьте два класса.
- Соберите средства у одного.
- Потратьте у другого.
- Проверьте балансы и несуществующий класс.

Пример использования:

```
sf = SchoolFund()
sf.add_class("10A", 1200)
sf.add_class("11B", 900)
sf.collect_money("10A", 300)
sf.spend_money("11B", 400)
sf.get_class_balance("10A")
sf.get_class_balance("11B")
sf.spend_money("11B", 600)  # недостаточно
sf.get_class_balance("12C")  # несуществует
```

15 Написать программу на Python, которая создает класс `ClubAccount`, моделирующий счет клуба. Класс должен поддерживать создание счетов участников, пополнение взносами, снятие на мероприятия и проверку баланса.

- Создайте класс `ClubAccount` с методом `__init__`, инициализирующим пустой словарь `members`.
- Метод `join_club` принимает ID участника и вступительный взнос. Если ID есть — ошибка; иначе — добавление.
- Метод `pay_dues` принимает ID и сумму. При наличии участника — пополнение; иначе — ошибка.
- Метод `request_funds` принимает ID и сумму. При наличии участника и достаточном балансе — списание; иначе — ошибка.
- Метод `check_member_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `ClubAccount`.
- Зарегистрируйте двух участников.
- Внесите взносы за одного.
- Запросите средства у другого.
- Проверьте балансы и несуществующего участника.

Пример использования:

```
ca = ClubAccount()
ca.join_club("MEM-01", 500)
ca.join_club("MEM-02", 400)
ca.pay_dues("MEM-01", 100)
ca.request_funds("MEM-02", 150)
ca.check_member_balance("MEM-01")
ca.check_member_balance("MEM-02")
ca.request_funds("MEM-02", 300)  # недостаточно
ca.check_member_balance("MEM-99")  # несуществует
```

16 Написать программу на Python, которая создает класс `ProjectBudget`, представляющий бюджет проекта. Класс должен поддерживать создание проектов, выделение средств, расход и проверку остатка.

- Создайте класс `ProjectBudget` с методом `__init__`, создающим пустой словарь `projects`.
- Метод `initiate_project` принимает код проекта и начальный бюджет. Если проект существует — ошибка; иначе — инициализация.
- Метод `allocate_funds` принимает код проекта и сумму. При наличии проекта — пополнение; иначе — ошибка.
- Метод `expend_funds` принимает код проекта и сумму. При наличии проекта и достаточном бюджете — списание; иначе — ошибка.
- Метод `check_project_balance` принимает код проекта и выводит баланс или ошибку.
- Создайте экземпляр `ProjectBudget`.
- Инициируйте два проекта.
- Выделите средства одному.
- Потратьте у другого.
- Проверьте балансы и несуществующий проект.

Пример использования:

```
pb = ProjectBudget()
pb.initiate_project("PRJ-Alpha", 10000)
pb.initiate_project("PRJ-Beta", 8000)
pb.allocate_funds("PRJ-Alpha", 2000)
pb.expend_funds("PRJ-Beta", 3000)
pb.check_project_balance("PRJ-Alpha")
pb.check_project_balance("PRJ-Beta")
pb.expend_funds("PRJ-Beta", 6000)  # недостаточно
pb.check_project_balance("PRJ-Gamma")  # несуществует
```

17 Написать программу на Python, которая создает класс `EventFund`, моделирующий фонд мероприятия. Класс должен поддерживать создание событий, сбор средств, оплату расходов и проверку баланса.

- Создайте класс `EventFund` с методом `__init__`, инициализирующим пустой словарь `events`.
- Метод `create_event` принимает название события и стартовый бюджет. Если событие есть — ошибка; иначе — создание.
- Метод `collect_sponsorship` принимает название и сумму. При наличии события — пополнение; иначе — ошибка.
- Метод `pay_vendor` принимает название и сумму. При наличии события и достаточном бюджете — списание; иначе — ошибка.
- Метод `view_event_balance` принимает название и выводит баланс или ошибку.
- Создайте экземпляр `EventFund`.
- Создайте два события.
- Соберите спонсорские средства для одного.
- Оплатите поставщика для другого.
- Проверьте балансы и несуществующее событие.

Пример использования:

```
ef = EventFund()
ef.create_event("Conference", 5000)
ef.create_event("Workshop", 3000)
ef.collect_sponsorship("Conference", 1500)
ef.pay_vendor("Workshop", 1000)
ef.view_event_balance("Conference")
ef.view_event_balance("Workshop")
ef.pay_vendor("Workshop", 2500)  # недостаточно
ef.view_event_balance("Seminar")  # несуществует
```

18 Написать программу на Python, которая создает класс `PersonalFinance`, представляющий личные финансы. Класс должен поддерживать создание категорий, пополнение доходами, списание расходами и проверку баланса.

- Создайте класс `PersonalFinance` с методом `__init__`, создающим пустой словарь `categories`.
- Метод `add_category` принимает название категории и начальный баланс. Если категория есть — ошибка; иначе — добавление.
- Метод `record_income` принимает название и сумму. При наличии категории — пополнение; иначе — ошибка.
- Метод `record_expense` принимает название и сумму. При наличии категории и достаточном балансе — списание; иначе — ошибка.
- Метод `check_category_balance` принимает название и выводит баланс или ошибку.
- Создайте экземпляр `PersonalFinance`.
- Добавьте две категории.
- Запишите доход в одну.

- Запишите расход в другую.
- Проверьте балансы и несуществующую категорию.

Пример использования:

```
pf = PersonalFinance()
pf.add_category("Salary", 25000)
pf.add_category("Entertainment", 2000)
pf.record_income("Salary", 5000)
pf.record_expense("Entertainment", 800)
pf.check_category_balance("Salary")
pf.check_category_balance("Entertainment")
pf.record_expense("Entertainment", 1500)  # недостаточно
pf.check_category_balance("Travel")  # несуществует
```

19 Написать программу на Python, которая создает класс `InvestmentAccount`, моделирующий инвестиционный счет. Класс должен поддерживать создание счетов, внесение капитала, снятие прибыли и проверку баланса.

- Создайте класс `InvestmentAccount` с методом `__init__`, инициализирующим пустой словарь `accounts`.
- Метод `open_investment` принимает ID счета и начальный капитал. Если счет есть — ошибка; иначе — открытие.
- Метод `invest_more` принимает ID и сумму. При наличии счета — пополнение; иначе — ошибка.
- Метод `withdraw_profit` принимает ID и сумму. При наличии счета и достаточном балансе — списание; иначе — ошибка.
- Метод `check_investment_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `InvestmentAccount`.
- Откройте два счета.
- Инвестируйте дополнительно в один.
- Снимите прибыль с другого.
- Проверьте балансы и несуществующий счет.

Пример использования:

```
ia = InvestmentAccount()
ia.open_investment("INV-01", 10000)
ia.open_investment("INV-02", 7000)
ia.invest_more("INV-01", 2000)
ia.withdraw_profit("INV-02", 1500)
ia.check_investment_balance("INV-01")
ia.check_investment_balance("INV-02")
ia.withdraw_profit("INV-02", 6000)  # недостаточно
ia.check_investment_balance("INV-99")  # несуществует
```

20 Написать программу на Python, которая создает класс `FamilyBudget`, представляющий семейный бюджет. Класс должен поддерживать создание членов семьи, пополнение общими доходами, списание личными расходами и проверку баланса.

- Создайте класс `FamilyBudget` с методом `__init__`, создающим пустой словарь `members`.
- Метод `add_family_member` принимает имя и начальный вклад. Если имя есть — ошибка; иначе — добавление.
- Метод `add_income` принимает имя и сумму. При наличии члена — пополнение; иначе — ошибка.
- Метод `deduct_expense` принимает имя и сумму. При наличии члена и достаточном балансе — списание; иначе — ошибка.
- Метод `check_member_balance` принимает имя и выводит баланс или ошибку.
- Создайте экземпляр `FamilyBudget`.
- Добавьте двух членов семьи.
- Добавьте доход одному.
- Спишите расход у другого.
- Проверьте балансы и несуществующего члена.

Пример использования:

```
fb = FamilyBudget()
fb.add_family_member("Mother", 20000)
fb.add_family_member("Father", 25000)
fb.add_income("Mother", 5000)
fb.deduct_expense("Father", 3000)
fb.check_member_balance("Mother")
fb.check_member_balance("Father")
fb.deduct_expense("Father", 23000)  # недостаточно
fb.check_member_balance("Child")  # несуществует
```

21 Написать программу на Python, которая создает класс `StartupFund`, моделирующий фонд стартапа. Класс должен поддерживать создание стартапов, привлечение инвестиций, оплату расходов и проверку баланса.

- Создайте класс `StartupFund` с методом `__init__`, инициализирующим пустой словарь `startups`.
- Метод `launch_startup` принимает название и начальный капитал. Если стартап есть — ошибка; иначе — запуск.
- Метод `attract_investment` принимает название и сумму. При наличии стартапа — пополнение; иначе — ошибка.
- Метод `cover_costs` принимает название и сумму. При наличии стартапа и достаточном балансе — списание; иначе — ошибка.
- Метод `check_startup_balance` принимает название и выводит баланс или ошибку.

- Создайте экземпляр `StartupFund`.
- Запустите два стартапа.
- Привлеките инвестиции в один.
- Покройте расходы другого.
- Проверьте балансы и несуществующий стартап.

Пример использования:

```
sf = StartupFund()
sf.launch_startup("TechApp", 50000)
sf.launch_startup("EcoShop", 30000)
sf.attract_investment("TechApp", 20000)
sf.cover_costs("EcoShop", 10000)
sf.check_startup_balance("TechApp")
sf.check_startup_balance("EcoShop")
sf.cover_costs("EcoShop", 25000)  # недостаточно
sf.check_startup_balance("FoodDelivery")  # несуществует
```

22 Написать программу на Python, которая создает класс `NonProfitAccount`, представляющий счет некоммерческой организации. Класс должен поддерживать создание проектов, получение грантов, оплату деятельности и проверку баланса.

- Создайте класс `NonProfitAccount` с методом `__init__`, создающим пустой словарь `projects`.
- Метод `initiate_nonprofit_project` принимает ID и начальный грант. Если проект есть — ошибка; иначе — инициализация.
- Метод `receive_grant` принимает ID и сумму. При наличии проекта — пополнение; иначе — ошибка.
- Метод `pay_operational_costs` принимает ID и сумму. При наличии проекта и достаточном балансе — списание; иначе — ошибка.
- Метод `check_project_funds` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `NonProfitAccount`.
- Инициируйте два проекта.
- Получите грант на один.
- Оплатите расходы другого.
- Проверьте балансы и несуществующий проект.

Пример использования:

```
np = NonProfitAccount()
np.initiate_nonprofit_project("EDU-01", 15000)
np.initiate_nonprofit_project("HEALTH-02", 12000)
np.receive_grant("EDU-01", 5000)
np.pay_operational_costs("HEALTH-02", 4000)
np.check_project_funds("EDU-01")
np.check_project_funds("HEALTH-02")
np.pay_operational_costs("HEALTH-02", 9000)  # недостаточно
np.check_project_funds("ENV-99")  # несуществует
```

23 Написать программу на Python, которая создает класс `FreelancerWallet`, моделирующий кошелек фрилансера. Класс должен поддерживать создание профилей, получение оплаты, оплату налогов и проверку баланса.

- Создайте класс `FreelancerWallet` с методом `__init__`, инициализирующим пустой словарь `freelancers`.
- Метод `register_freelancer` принимает ник и начальный баланс. Если ник есть — ошибка; иначе — регистрация.
- Метод `receive_payment` принимает ник и сумму. При наличии фрилансера — пополнение; иначе — ошибка.
- Метод `pay_taxes` принимает ник и сумму. При наличии фрилансера и достаточном балансе — списание; иначе — ошибка.
- Метод `check_freelancer_balance` принимает ник и выводит баланс или ошибку.
- Создайте экземпляр `FreelancerWallet`.
- Зарегистрируйте двух фрилансеров.
- Получите оплату для одного.
- Оплатите налоги для другого.
- Проверьте балансы и несуществующего фрилансера.

Пример использования:

```
fw = FreelancerWallet()
fw.register_freelancer("dev_alex", 0)
fw.register_freelancer("design_maria", 0)
fw.receive_payment("dev_alex", 10000)
fw.receive_payment("design_maria", 2500)
fw.pay_taxes("design_maria", 2000)
fw.check_freelancer_balance("dev_alex")
fw.check_freelancer_balance("design_maria")
fw.pay_taxes("design_maria", 1000)  # недостаточно
fw.check_freelancer_balance("writer_john")  # несуществует
```

24 Написать программу на Python, которая создает класс `RentalIncome`, представляющий доход от аренды. Класс должен поддерживать создание объектов недвижимости, получение арендной платы, оплату расходов и проверку баланса.

- Создайте класс `RentalIncome` с методом `__init__`, создающим пустой словарь `properties`.
- Метод `add_property` принимает адрес и начальный баланс. Если адрес есть — ошибка; иначе — добавление.
- Метод `collect_rent` принимает адрес и сумму. При наличии объекта — пополнение; иначе — ошибка.
- Метод `pay_maintenance` принимает адрес и сумму. При наличии объекта и достаточном балансе — списание; иначе — ошибка.
- Метод `check_property_balance` принимает адрес и выводит баланс или ошибку.

- Создайте экземпляр `RentalIncome`.
- Добавьте два объекта.
- Соберите арендную плату с одного.
- Оплатите обслуживание другого.
- Проверьте балансы и несуществующий адрес.

Пример использования:

```
ri = RentalIncome()
ri.add_property("123 Main St", 0)
ri.add_property("456 Oak Ave", 0)
ri.collect_rent("123 Main St", 2000)
ri.collect_rent("456 Oak Ave", 700)
ri.pay_maintenance("456 Oak Ave", 300)
ri.check_property_balance("123 Main St")
ri.check_property_balance("456 Oak Ave")
ri.pay_maintenance("456 Oak Ave", 500)  # недостаточно
ri.check_property_balance("789 Pine Rd")  # несуществует
```

25 Написать программу на Python, которая создает класс `ScholarshipFund`, моделирующий стипендиальный фонд. Класс должен поддерживать создание получателей, выдачу стипендий, возврат средств и проверку баланса.

- Создайте класс `ScholarshipFund` с методом `__init__`, инициализирующим пустой словарь `recipients`.
- Метод `enroll_recipient` принимает ID и начальную стипендию. Если ID есть — ошибка; иначе — зачисление.
- Метод `award_scholarship` принимает ID и сумму. При наличии получателя — пополнение; иначе — ошибка.
- Метод `return_funds` принимает ID и сумму. При наличии получателя и достаточном балансе — списание; иначе — ошибка.
- Метод `check_recipient_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `ScholarshipFund`.
- Зачислите двух получателей.
- Выдайте стипендию одному.
- Примите возврат от другого.
- Проверьте балансы и несуществующего получателя.

Пример использования:

```
sf = ScholarshipFund()
sf.enroll_recipient("SCH-01", 5000)
sf.enroll_recipient("SCH-02", 4000)
sf.award_scholarship("SCH-01", 1000)
sf.return_funds("SCH-02", 500)
```

```
sf.check_recipient_balance("SCH-01")
sf.check_recipient_balance("SCH-02")
sf.return_funds("SCH-02", 4000)  # недостаточно
sf.check_recipient_balance("SCH-99")  # несуществует
```

26 Написать программу на Python, которая создает класс `Crowdfunding`, представляющий краудфандинговую платформу. Класс должен поддерживать создание кампаний, сбор средств, возврат пожертвований и проверку баланса.

- Создайте класс `Crowdfunding` с методом `__init__`, создающим пустой словарь `campaigns`.
- Метод `start_campaign` принимает название и начальный баланс. Если кампания есть — ошибка; иначе — создание.
- Метод `donate` принимает название и сумму. При наличии кампании — пополнение; иначе — ошибка.
- Метод `refund` принимает название и сумму. При наличии кампании и достаточном балансе — списание; иначе — ошибка.
- Метод `check_campaign_balance` принимает название и выводит баланс или ошибку.
- Создайте экземпляр `Crowdfunding`.
- Запустите две кампании.
- Пожертвуйте в одну.
- Верните средства из другой.
- Проверьте балансы и несуществующую кампанию.

Пример использования:

```
cf = Crowdfunding()
cf.start_campaign("BookPublish", 10000)
cf.start_campaign("ArtExhibit", 8000)
cf.donate("BookPublish", 3000)
cf.refund("ArtExhibit", 500)
cf.check_campaign_balance("BookPublish")
cf.check_campaign_balance("ArtExhibit")
cf.refund("ArtExhibit", 8000)  # недостаточно
cf.check_campaign_balance("FilmProject")  # несуществует
```

27 Написать программу на Python, которая создает класс `PiggyBank`, моделирующий копилку. Класс должен поддерживать создание копилок, добавление монет, извлечение средств и проверку баланса.

- Создайте класс `PiggyBank` с методом `__init__`, инициализирующим пустой словарь `banks`.
- Метод `create_piggy` принимает имя и начальную сумму. Если имя есть — ошибка; иначе — создание.

- Метод `add_coins` принимает имя и сумму. При наличии копилки — пополнение; иначе — ошибка.
- Метод `break_piggy` принимает имя и сумму. При наличии копилки и достаточном балансе — списание; иначе — ошибка.
- Метод `check_piggy_balance` принимает имя и выводит баланс или ошибку.
- Создайте экземпляр `PiggyBank`.
- Создайте две копилки.
- Добавьте монеты в одну.
- Разбейте другую частично.
- Проверьте балансы и несуществующую копилку.

Пример использования:

```
pb = PiggyBank()
pb.create_piggy("Vacation", 200)
pb.create_piggy("Gadget", 150)
pb.add_coins("Vacation", 100)
pb.break_piggy("Gadget", 50)
pb.check_piggy_balance("Vacation")
pb.check_piggy_balance("Gadget")
pb.break_piggy("Gadget", 120)  # недостаточно
pb.check_piggy_balance("Car")  # несуществует
```

28 Написать программу на Python, которая создает класс `BusinessAccount`, представляющий бизнес-счет. Класс должен поддерживать создание компаний, зачисление выручки, оплату счетов и проверку баланса.

- Создайте класс `BusinessAccount` с методом `__init__`, создающим пустой словарь `companies`.
- Метод `register_business` принимает название и начальный капитал. Если компания есть — ошибка; иначе — регистрация.
- Метод `record_revenue` принимает название и сумму. При наличии компании — пополнение; иначе — ошибка.
- Метод `pay_bills` принимает название и сумму. При наличии компании и достаточном балансе — списание; иначе — ошибка.
- Метод `check_business_balance` принимает название и выводит баланс или ошибку.
- Создайте экземпляр `BusinessAccount`.
- Зарегистрируйте две компании.
- Запишите выручку одной.
- Оплатите счета другой.
- Проверьте балансы и несуществующую компанию.

Пример использования:

```
ba = BusinessAccount()
ba.register_business("TechCorp", 50000)
ba.register_business("CafeLtd", 20000)
ba.record_revenue("TechCorp", 15000)
ba.pay_bills("CafeLtd", 3000)
ba.check_business_balance("TechCorp")
ba.check_business_balance("CafeLtd")
ba.pay_bills("CafeLtd", 18000)  # недостаточно
ba.check_business_balance("ShopInc")  # несуществует
```

29 Написать программу на Python, которая создает класс `GrantManager`, моделирующий управление грантами. Класс должен поддерживать создание грантов, выделение средств, отчетность и проверку баланса.

- Создайте класс `GrantManager` с методом `__init__`, инициализирующим пустой словарь `grants`.
- Метод `issue_grant` принимает код и сумму. Если код есть — ошибка; иначе — создание.
- Метод `disburse_funds` принимает код и сумму. При наличии гранта и достаточном балансе — списание (выдача средств); иначе — ошибка.
- Метод `submit_report` принимает код и сумму. При наличии гранта и достаточном балансе — списание; иначе — ошибка.
- Метод `check_grant_status` принимает код и выводит баланс или ошибку.
- Создайте экземпляр `GrantManager`.
- Выдайте два гранта.
- Распределите средства по одному.
- Подайте отчет по другому.
- Проверьте статусы и несуществующий грант.

Пример использования:

```
gm = GrantManager()
gm.issue_grant("GR-2024-01", 10000)
gm.issue_grant("GR-2024-02", 8000)
gm.disburse_funds("GR-2024-01", 4000)
gm.submit_report("GR-2024-02", 2000)
gm.check_grant_status("GR-2024-01")
gm.check_grant_status("GR-2024-02")
gm.submit_report("GR-2024-02", 7000)  # недостаточно
gm.check_grant_status("GR-2024-99")  # несуществует
```

30 Написать программу на Python, которая создает класс `SubscriptionService`, представляющий сервис подписок. Класс должен поддерживать создание пользователей, оплату подписок, возврат средств и проверку баланса.

- Создайте класс `SubscriptionService` с методом `__init__`, создающим пустой словарь `subscribers`.

- Метод `subscribe_user` принимает email и начальный баланс. Если email есть — ошибка; иначе — подписка.
- Метод `charge_payment` принимает email и сумму. При наличии пользователя — пополнение; иначе — ошибка.
- Метод `refund_payment` принимает email и сумму. При наличии пользователя и достаточном балансе — списание; иначе — ошибка.
- Метод `check_subscription_balance` принимает email и выводит баланс или ошибку.
- Создайте экземпляр `SubscriptionService`.
- Подпишите двух пользователей.
- Спишите оплату с одного.
- Верните средства другому.
- Проверьте балансы и несуществующий email.

Пример использования:

```
ss = SubscriptionService()
ss.subscribe_user("user1@example.com", 100)
ss.subscribe_user("user2@example.com", 80)
ss.charge_payment("user1@example.com", 20)
ss.refund_payment("user2@example.com", 10)
ss.check_subscription_balance("user1@example.com")
ss.check_subscription_balance("user2@example.com")
ss.refund_payment("user2@example.com", 80)  # недостаточно
ss.check_subscription_balance("user3@example.com")  # несуществует
```

31 Написать программу на Python, которая создает класс `LoyaltyProgram`, моделирующий программу лояльности. Класс должен поддерживать создание участников, начисление баллов, списание за вознаграждения и проверку баланса.

- Создайте класс `LoyaltyProgram` с методом `__init__`, инициализирующим пустой словарь `members`.
- Метод `enroll_member` принимает ID и начальные баллы. Если ID есть — ошибка; иначе — зачисление.
- Метод `earn_points` принимает ID и количество. При наличии участника — пополнение; иначе — ошибка.
- Метод `redeem_points` принимает ID и количество. При наличии участника и достаточном балансе — списание; иначе — ошибка.
- Метод `check_points_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `LoyaltyProgram`.
- Зачислите двух участников.
- Начислите баллы одному.
- Спишите у другого.
- Проверьте балансы и несуществующего участника.

Пример использования:

```
lp = LoyaltyProgram()
lp.enroll_member("MEM-101", 500)
lp.enroll_member("MEM-102", 300)
lp.earn_points("MEM-101", 200)
lp.redeem_points("MEM-102", 100)
lp.check_points_balance("MEM-101")
lp.check_points_balance("MEM-102")
lp.redeem_points("MEM-102", 250)  # недостаточно
lp.check_points_balance("MEM-999")  # несуществует
```

32 Написать программу на Python, которая создает класс `UtilityBill`, представляющий оплату коммунальных услуг. Класс должен поддерживать создание лицевых счетов, внесение платежей, списание задолженностей и проверку баланса.

- Создайте класс `UtilityBill` с методом `__init__`, создающим пустой словарь `accounts`.
- Метод `create_utility_account` принимает номер и начальный долг. Если номер есть — ошибка; иначе — создание.
- Метод `make_payment` принимает номер и сумму. При наличии счета — пополнение; иначе — ошибка.
- Метод `apply_charges` принимает номер и сумму. При наличии счета и достаточном балансе — списание; иначе — ошибка.
- Метод `check_account_status` принимает номер и выводит баланс или ошибку.
- Создайте экземпляр `UtilityBill`.
- Создайте два счета.
- Внесите платеж по одному.
- Начислите плату по другому.
- Проверьте статусы и несуществующий счет.

Пример использования:

```
ub = UtilityBill()
ub.create_utility_account("UTIL-01", 0)
ub.create_utility_account("UTIL-02", 0)
ub.make_payment("UTIL-01", 1500)
ub.make_payment("UTIL-02", 1200)
ub.apply_charges("UTIL-02", 800)
ub.check_account_status("UTIL-01")
ub.check_account_status("UTIL-02")
ub.apply_charges("UTIL-02", 1000)  # недостаточно
ub.check_account_status("UTIL-99")  # несуществует
```

33 Написать программу на Python, которая создает класс `InsuranceFund`, моделирующий страховой фонд. Класс должен поддерживать создание полисов, уплату премий, выплату возмещений и проверку баланса.

- Создайте класс `InsuranceFund` с методом `__init__`, инициализирующим пустой словарь `policies`.
- Метод `issue_policy` принимает номер полиса и начальный взнос. Если полис есть — ошибка; иначе — выдача.
- Метод `pay_premium` принимает номер и сумму. При наличии полиса — пополнение; иначе — ошибка.
- Метод `process_claim` принимает номер и сумму. При наличии полиса и достаточном балансе — списание; иначе — ошибка.
- Метод `check_policy_balance` принимает номер и выводит баланс или ошибку.
- Создайте экземпляр `InsuranceFund`.
- Выдайте два полиса.
- Уплатите премию по одному.
- Обработайте заявку по другому.
- Проверьте балансы и несуществующий полис.

Пример использования:

```
ifund = InsuranceFund()
ifund.issue_policy("POL-501", 10000)
ifund.issue_policy("POL-502", 8000)
ifund.pay_premium("POL-501", 2000)
ifund.process_claim("POL-502", 3000)
ifund.check_policy_balance("POL-501")
ifund.check_policy_balance("POL-502")
ifund.process_claim("POL-502", 6000)  # недостаточно
ifund.check_policy_balance("POL-999")  # несуществует
```

34 Написать программу на Python, которая создает класс `DonationBox`, представляющий ящик для пожертвований. Класс должен поддерживать создание ящиков, сбор средств, выдачу помощи и проверку баланса.

- Создайте класс `DonationBox` с методом `__init__`, создающим пустой словарь `boxes`.
- Метод `install_box` принимает локацию и начальный сбор. Если локация есть — ошибка; иначе — установка.
- Метод `collect_donations` принимает локацию и сумму. При наличии ящика — пополнение; иначе — ошибка.
- Метод `distribute_aid` принимает локацию и сумму. При наличии ящика и достаточном балансе — списание; иначе — ошибка.
- Метод `check_box_balance` принимает локацию и выводит баланс или ошибку.
- Создайте экземпляр `DonationBox`.
- Установите два ящика.
- Соберите пожертвования в один.
- Распределите помощь из другого.

- Проверьте балансы и несуществующую локацию.

Пример использования:

```
db = DonationBox()
db.install_box("Hospital", 0)
db.install_box("School", 0)
db.collect_donations("Hospital", 5000)
db.collect_donations("School", 1500)
db.distribute_aid("School", 1000)
db.check_box_balance("Hospital")
db.check_box_balance("School")
db.distribute_aid("School", 2000)  # недостаточно
db.check_box_balance("Park")  # несуществует
```

35 Написать программу на Python, которая создает класс `RewardWallet`, моделирующий кошелек вознаграждений. Класс должен поддерживать создание кошельков, начисление бонусов, списание за покупки и проверку баланса.

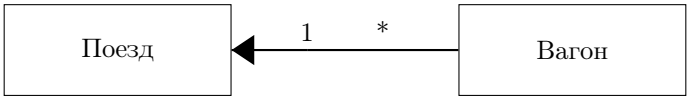
- Создайте класс `RewardWallet` с методом `__init__`, инициализирующим пустой словарь `wallets`.
- Метод `activate_wallet` принимает ID и начальные бонусы. Если ID есть — ошибка; иначе — активация.
- Метод `award_bonus` принимает ID и сумму. При наличии кошелька — пополнение; иначе — ошибка.
- Метод `redeem_reward` принимает ID и сумму. При наличии кошелька и достаточном балансе — списание; иначе — ошибка.
- Метод `check_reward_balance` принимает ID и выводит баланс или ошибку.
- Создайте экземпляр `RewardWallet`.
- Активируйте два кошелька.
- Начислите бонусы одному.
- Потратьте у другого.
- Проверьте балансы и несуществующий ID.

Пример использования:

```
rw = RewardWallet()
rw.activate_wallet("RW-001", 1000)
rw.activate_wallet("RW-002", 800)
rw.award_bonus("RW-001", 200)
rw.redeem_reward("RW-002", 300)
rw.check_reward_balance("RW-001")
rw.check_reward_balance("RW-002")
rw.redeem_reward("RW-002", 600)  # недостаточно
rw.check_reward_balance("RW-999")  # несуществует
```

2.5 Семинар «Композиция» (2 часа)

Теория



Композиция — это концепция, позволяющая моделировать отношения между классами в программе. Она представляет собой один из способов организации взаимодействия классов. Композиция позволяет создавать сложные типы, комбинируя объекты других типов. Это означает, что один класс («целое») может содержать экземпляр другого класса («часть») как своё поле.

Классы, содержащие объекты другхих классов, обычно называются *композитами* (*composites*), а классы, используемые для создания более сложных типов, называются *компонентами* (*components*).

Композиция позволяет повторно использовать код путём включения объектов других классов, избегая при этом наследования.

Вопросы для подготовки к защите

- Что такое композиция?
- Сравните композицию и наследование.
- Опишите достоинства и недостатки указания типов аргументов и результатов у функций.

В ходе выполнения используйте свойство инкапсуляции (даже если в формулировке задания это не учтено).

2.5.1 Задача 1

1 **Формирование состава груженых контейнеров**

(a) Создайте класс `Container`, который будет представлять собой контейнер с грузом. В конструкторе класса `Container` инициализируйте значения контейнера и груза из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список контейнеров (не менее 14):

```
['Контейнер_1', 'Контейнер_2', ..., 'Контейнер_14']
```

`MasList: list[str]` — это список грузов для контейнеров (не менее 4):

```
['Электроника', 'Мебель', 'Одежда', 'Продукты']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TrainOfContainers`, который будет представлять собой состав, состоящий из моделей контейнеров. В конструкторе класса `TrainOfContainers` инициализируйте список контейнеров `self.train: list[Container]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TrainOfContainers`, который будет перемешивать контейнеры в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Container`, который будет возвращать i-й контейнер и груз из списка `self.train`.

(e) Создайте экземпляр класса `TrainOfContainers` и вызовите метод `shuffle` для перемешивания контейнеров.

(f) Создайте цикл, который будет запрашивать у пользователя номер контейнера из состава и выводить информацию о выбранном контейнере.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все контейнеры или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора контейнеров.

(i) Убедитесь, что пользователь вводит корректные номера контейнеров и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров контейнеров и грузов.

2 **Формирование состава почтовых посылок**

(a) Создайте класс `Parcel`, который будет представлять собой посылку с содержимым. В конструкторе класса `Parcel` инициализируйте значения посылки и содержимого из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список посылок (не менее 14):

`['Посылка_1', 'Посылка_2', ..., 'Посылка_14']`

`MasList: list[str]` — это список содержимого посылок (не менее 4):

`['Книги', 'Игрушки', 'Косметика', 'Спортивный инвентарь']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TrainOfParcels`, который будет представлять собой состав, состоящий из моделей посылок. В конструкторе класса `TrainOfParcels` инициализируйте список посылок `self.train: list[Parcel]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TrainOfParcels`, который будет перемешивать посылки в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Parcel`, который будет возвращать i-ю посылку и её содержимое из списка `self.train`.

(e) Создайте экземпляр класса `TrainOfParcels` и вызовите метод `shuffle` для перемешивания посылок.

(f) Создайте цикл, который будет запрашивать у пользователя номер посылки из состава и выводить информацию о выбранной посылке.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все посылки или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора посылок.

(i) Убедитесь, что пользователь вводит корректные номера посылок и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров посылок и содержимого.

3 **Формирование автовоза с автомобилями**

(a) Создайте класс `Car`, который будет представлять собой автомобиль на автовозе. В конструкторе класса `Car` инициализируйте значения автомобиля и его марки из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список автомобилей (не менее 14):

`['Автомобиль_1', 'Автомобиль_2', ..., 'Автомобиль_14']`

`MasList: list[str]` — это список марок автомобилей (не менее 4):

`['Toyota', 'BMW', 'Lada', 'Tesla']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `CarCarrier`, который будет представлять собой автовоз, состоящий из моделей автомобилей. В конструкторе класса `CarCarrier` инициализируйте список автомобилей `self.train: list[Car]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `CarCarrier`, который будет перемешивать автомобили в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Car`, который будет возвращать i-й автомобиль и его марку из списка `self.train`.

(e) Создайте экземпляр класса `CarCarrier` и вызовите метод `shuffle` для перемешивания автомобилей.

(f) Создайте цикл, который будет запрашивать у пользователя номер автомобиля на автовозе и выводить информацию о выбранном автомобиле.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все автомобили или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора автомобилей.

(i) Убедитесь, что пользователь вводит корректные номера автомобилей и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров автомобилей и марок.

4 **Формирование багажного состава из чемоданов**

(a) Создайте класс `Suitcase`, который будет представлять собой чемодан с владельцем. В конструкторе класса `Suitcase` инициализируйте значения чемодана и владельца из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список чемоданов (не менее 14):

`['Чемодан_1', 'Чемодан_2', ..., 'Чемодан_14']`

`MasList: list[str]` — это список владельцев (не менее 4):

`['Иванов', 'Петров', 'Сидоров', 'Кузнецов']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `BaggageTrain`, который будет представлять собой багажный состав, состоящий из моделей чемоданов. В конструкторе класса `BaggageTrain` инициализируйте список чемоданов `self.train: list[Suitcase]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `BaggageTrain`, который будет перемешивать чемоданы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Suitcase`, который будет возвращать i-й чемодан и его владельца из списка `self.train`.

(e) Создайте экземпляр класса `BaggageTrain` и вызовите метод `shuffle` для перемешивания чемоданов.

(f) Создайте цикл, который будет запрашивать у пользователя номер чемодана из состава и выводить информацию о выбранном чемодане.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все чемоданы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора чемоданов.

(i) Убедитесь, что пользователь вводит корректные номера чемоданов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров чемоданов и владельцев.

5 **Формирование складского состава из ящиков**

(a) Создайте класс `Box`, который будет представлять собой ящик с содержимым. В конструкторе класса `Box` инициализируйте значения ящика и содержимого из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список ящиков (не менее 14):

`['Ящик_1', 'Ящик_2', ..., 'Ящик_14']`

`MasList: list[str]` — это список типов содержимого (не менее 4):

`['Инструменты', 'Запчасти', 'Химикаты', 'Упаковка']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `WarehouseTrain`, который будет представлять собой состав, состоящий из моделей ящиков. В конструкторе класса `WarehouseTrain` инициализируйте список ящиков `self.train: list[Box]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `WarehouseTrain`, который будет перемешивать ящики в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Box`, который будет возвращать i-й ящик и его содержимое из списка `self.train`.

(e) Создайте экземпляр класса `WarehouseTrain` и вызовите метод `shuffle` для перемешивания ящиков.

(f) Создайте цикл, который будет запрашивать у пользователя номер ящика из состава и выводить информацию о выбранном ящике.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все ящики или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора ящиков.

(i) Убедитесь, что пользователь вводит корректные номера ящиков и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров ящиков и содержимого.

6 **Формирование состава морских судов с грузом**

(a) Создайте класс `Ship`, который будет представлять собой судно с грузом. В конструкторе класса `Ship` инициализируйте значения судна и груза из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список судов (не менее 14):

`['Судно_1', 'Судно_2', ..., 'Судно_14']`

`MasList: list[str]` — это список грузов (не менее 4):

`['Нефть', 'Уголь', 'Зерно', 'Лес']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `Fleet`, который будет представлять собой флотилию, состоящую из моделей судов. В конструкторе класса `Fleet` инициализируйте список судов `self.train: list[Ship]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `Fleet`, который будет перемешивать суда в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Ship`, который будет возвращать i-е судно и его груз из списка `self.train`.

(e) Создайте экземпляр класса `Fleet` и вызовите метод `shuffle` для перемешивания судов.

(f) Создайте цикл, который будет запрашивать у пользователя номер судна из флотилии и выводить информацию о выбранном судне.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все суда или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора судов.

(i) Убедитесь, что пользователь вводит корректные номера судов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров судов и грузов.

7 Формирование состава ракет-носителей

(a) Создайте класс `Rocket`, который будет представлять собой ракету с полезной нагрузкой. В конструкторе класса `Rocket` инициализируйте значения ракеты и нагрузки из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список ракет (не менее 14):

`['Ракета_1', 'Ракета_2', ..., 'Ракета_14']`

`MasList: list[str]` — это список типов нагрузки (не менее 4):

`['Спутник', 'Грузовой модуль', 'Экипаж', 'Научное оборудование']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `RocketTrain`, который будет представлять собой состав ракет. В конструкторе класса `RocketTrain` инициализируйте список ракет `self.train: list[Rocket]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `RocketTrain`, который будет перемешивать ракеты в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Rocket`, который будет возвращать i-ю ракету и её нагрузку из списка `self.train`.

(e) Создайте экземпляр класса `RocketTrain` и вызовите метод `shuffle` для перемешивания ракет.

(f) Создайте цикл, который будет запрашивать у пользователя номер ракеты и выводить информацию о выбранной ракете.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все ракеты или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора ракет.

(i) Убедитесь, что пользователь вводит корректные номера ракет и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров ракет и нагрузок.

8 **Формирование состава дронов с грузом**

(a) Создайте класс `Drone`, который будет представлять собой дрон с миссией. В конструкторе класса `Drone` инициализируйте значения дрона и миссии из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список дронов (не менее 14):

`['Дрон_1', 'Дрон_2', ..., 'Дрон_14']`

`MasList: list[str]` — это список миссий (не менее 4):

`['Фотосъёмка', 'Доставка', 'Разведка', 'Мониторинг']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `DroneSquadron`, который будет представлять собой эскадрилью, состоящую из моделей дронов. В конструкторе класса `DroneSquadron` инициализируйте список дронов `self.train: list[Drone]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `DroneSquadron`, который будет перемешивать дроны в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Drone`, который будет возвращать i-й дрон и его миссию из списка `self.train`.

(e) Создайте экземпляр класса `DroneSquadron` и вызовите метод `shuffle` для перемешивания дронов.

(f) Создайте цикл, который будет запрашивать у пользователя номер дрона и выводить информацию о выбранном дроне.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все дроны или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора дронов.

(i) Убедитесь, что пользователь вводит корректные номера дронов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров дронов и миссий.

9 **Формирование состава тележек в супермаркете**

(a) Создайте класс `Trolley`, который будет представлять собой тележку с типом покупателя. В конструкторе класса `Trolley` инициализируйте значения тележки и типа покупателя из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список тележек (не менее 14):

`['Тележка_1', 'Тележка_2', ..., 'Тележка_14']`

`MasList: list[str]` — это список типов покупателей (не менее 4):

`['Семья', 'Студент', 'Пенсионер', 'Турист']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TrolleyTrain`, который будет представлять собой состав тележек. В конструкторе класса `TrolleyTrain` инициализируйте список тележек `self.train: list[Trolley]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TrolleyTrain`, который будет перемешивать тележки в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Trolley`, который будет возвращать i-ю тележку и тип её покупателя из списка `self.train`.

(e) Создайте экземпляр класса `TrolleyTrain` и вызовите метод `shuffle` для перемешивания тележек.

(f) Создайте цикл, который будет запрашивать у пользователя номер тележки и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все тележки или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора тележек.

(i) Убедитесь, что пользователь вводит корректные номера тележек и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров тележек и типов покупателей.

10 **Формирование состава камер хранения**

(a) Создайте класс `Locker`, который будет представлять собой камеру хранения с содержимым. В конструкторе класса `Locker` инициализируйте значения камеры и содержимого из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список камер (не менее 14):

`['Камера_1', 'Камера_2', ..., 'Камера_14']`

`MasList: list[str]` — это список типов содержимого (не менее 4):

`['Велосипед', 'Чемодан', 'Инструменты', 'Спортинвентарь']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `StorageTrain`, который будет представлять собой состав камер хранения. В конструкторе класса `StorageTrain` инициализируйте список камер `self.train: list[Locker]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `StorageTrain`, который будет перемешивать камеры в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Locker`, который будет возвращать i-ю камеру и её содержимое из списка `self.train`.

(e) Создайте экземпляр класса `StorageTrain` и вызовите метод `shuffle` для перемешивания камер.

(f) Создайте цикл, который будет запрашивать у пользователя номер камеры и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все камеры или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора камер.

(i) Убедитесь, что пользователь вводит корректные номера камер и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров камер и содержимого.

11 **Формирование состава самолётов с бортами**

(a) Создайте класс `Aircraft`, который будет представлять собой самолёт с типом рейса. В конструкторе класса `Aircraft` инициализируйте значения самолёта и типа рейса из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список самолётов (не менее 14):

`['Борт_1', 'Борт_2', ..., 'Борт_14']`

`MasList: list[str]` — это список типов рейсов (не менее 4):

`['Пассажирский', 'Грузовой', 'Военный', 'Санитарный']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `AirFleet`, который будет представлять собой воздушный флот, состоящий из моделей самолётов. В конструкторе класса `AirFleet` инициализируйте список самолётов `self.train: list[Aircraft]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `AirFleet`, который будет перемешивать самолёты в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Aircraft`, который будет возвращать i-й самолёт и его тип рейса из списка `self.train`.

(e) Создайте экземпляр класса `AirFleet` и вызовите метод `shuffle` для перемешивания самолётов.

(f) Создайте цикл, который будет запрашивать у пользователя номер самолёта и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все самолёты или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора самолётов.

(i) Убедитесь, что пользователь вводит корректные номера самолётов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров самолётов и типов рейсов.

12 **Формирование состава танкеров с жидкостями**

(a) Создайте класс `Tanker`, который будет представлять собой танкер с жидкостью. В конструкторе класса `Tanker` инициализируйте значения танкера и жидкости из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список танкеров (не менее 14):

`['Танкер_1', 'Танкер_2', ..., 'Танкер_14']`

`MasList: list[str]` — это список жидкостей (не менее 4):

`['Вода', 'Молоко', 'Топливо', 'Химикаты']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TankerConvoy`, который будет представлять собой конвой танкеров. В конструкторе класса `TankerConvoy` инициализируйте список танкеров `self.train: list[Tanker]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TankerConvoy`, который будет перемешивать танкеры в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Tanker`, который будет возвращать i-й танкер и его жидкость из списка `self.train`.

(e) Создайте экземпляр класса `TankerConvoy` и вызовите метод `shuffle` для перемешивания танкеров.

(f) Создайте цикл, который будет запрашивать у пользователя номер танкера и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все танкеры или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора танкеров.

(i) Убедитесь, что пользователь вводит корректные номера танкеров и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров танкеров и жидкостей.

13 Формирование состава паллет на складе

(a) Создайте класс `Pallet`, который будет представлять собой паллету с товаром. В конструкторе класса `Pallet` инициализируйте значения паллеты и товара из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список паллет (не менее 14):

`['Паллета_1', 'Паллета_2', ..., 'Паллета_14']`

`MasList: list[str]` — это список типов товаров (не менее 4):

`['Напитки', 'Консервы', 'Бытовая химия', 'Бумажная продукция']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `PalletTrain`, который будет представлять собой состав паллет. В конструкторе класса `PalletTrain` инициализируйте список паллет `self.train: list[Pallet]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `PalletTrain`, который будет перемешивать паллеты в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Pallet`, который будет возвращать i-ю паллету и её товар из списка `self.train`.

(e) Создайте экземпляр класса `PalletTrain` и вызовите метод `shuffle` для перемешивания паллет.

(f) Создайте цикл, который будет запрашивать у пользователя номер паллеты и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все паллеты или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора паллет.

(i) Убедитесь, что пользователь вводит корректные номера паллет и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров паллет и товаров.

14 **Формирование состава вагонов-цистерн**

(a) Создайте класс `TankWagon`, который будет представлять собой цистерну с содержимым. В конструкторе класса `TankWagon` инициализируйте значения цистерны и содержимого из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список цистерн (не менее 14):

`['Цистерна_1', 'Цистерна_2', ..., 'Цистерна_14']`

`MasList: list[str]` — это список типов содержимого (не менее 4):

`['Бензин', 'Дизель', 'Газ', 'Вода']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TankTrain`, который будет представлять собой состав цистерн. В конструкторе класса `TankTrain` инициализируйте список цистерн `self.train: list[TankWagon]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TankTrain`, который будет перемешивать цистерны в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> TankWagon`, который будет возвращать i-ю цистерну и её содержимое из списка `self.train`.

(e) Создайте экземпляр класса `TankTrain` и вызовите метод `shuffle` для перемешивания цистерн.

(f) Создайте цикл, который будет запрашивать у пользователя номер цистерны и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все цистерны или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора цистерн.

(i) Убедитесь, что пользователь вводит корректные номера цистерн и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров цистерн и содержимого.

15 **Формирование состава промышленных роботов**

(a) Создайте класс `Robot`, который будет представлять собой робота с модулем. В конструкторе класса `Robot` инициализируйте значения робота и модуля из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список роботов (не менее 14):

```
['Робот_1', 'Робот_2', ..., 'Робот_14']
```

`MasList: list[str]` — это список модулей (не менее 4):

```
['Манипулятор', 'Камера', 'Сенсор', 'Батарея']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `RobotLine`, который будет представлять собой производственную линию роботов. В конструкторе класса `RobotLine` инициализируйте список роботов `self.train: list[Robot]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `RobotLine`, который будет перемешивать роботов в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Robot`, который будет возвращать i-го робота и его модуль из списка `self.train`.

(e) Создайте экземпляр класса `RobotLine` и вызовите метод `shuffle` для перемешивания роботов.

(f) Создайте цикл, который будет запрашивать у пользователя номер робота и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет всех роботов или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора роботов.

(i) Убедитесь, что пользователь вводит корректные номера роботов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров роботов и модулей.

16 **Формирование состава клеток с животными**

(a) Создайте класс `Cage`, который будет представлять собой клетку с животным. В конструкторе класса `Cage` инициализируйте значения клетки и животного из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список клеток (не менее 14):

```
['Клетка_1', 'Клетка_2', ..., 'Клетка_14']
```

`MasList: list[str]` — это список животных (не менее 4):

```
['Собака', 'Кошка', 'Попугай', 'Кролик']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `ZooTrain`, который будет представлять собой состав клеток. В конструкторе класса `ZooTrain` инициализируйте список клеток `self.train: list[Cage]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `ZooTrain`, который будет перемешивать клетки в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Cage`, который будет возвращать i-ю клетку и её животное из списка `self.train`.

(e) Создайте экземпляр класса `ZooTrain` и вызовите метод `shuffle` для перемешивания клеток.

(f) Создайте цикл, который будет запрашивать у пользователя номер клетки и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все клетки или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора клеток.

(i) Убедитесь, что пользователь вводит корректные номера клеток и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров клеток и животных.

17 **Формирование состава прицепов на автодороге**

(a) Создайте класс `Trailer`, который будет представлять собой прицеп с грузом. В конструкторе класса `Trailer` инициализируйте значения прицепа и груза из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список прицепов (не менее 14):

```
['Прицеп_1', 'Прицеп_2', ..., 'Прицеп_14']
```

`MasList: list[str]` — это список типов груза (не менее 4):

```
['Строительные материалы', 'Мебель', 'Техника', 'Сельхозпродукция']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TrailerConvoy`, который будет представлять собой конвой прицепов. В конструкторе класса `TrailerConvoy` инициализируйте список прицепов `self.train: list[Trailer]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TrailerConvoy`, который будет перемешивать прицепы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Trailer`, который будет возвращать i-й прицеп и его груз из списка `self.train`.

(e) Создайте экземпляр класса `TrailerConvoy` и вызовите метод `shuffle` для перемешивания прицепов.

(f) Создайте цикл, который будет запрашивать у пользователя номер прицепа и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все прицепы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора прицепов.

(i) Убедитесь, что пользователь вводит корректные номера прицепов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров прицепов и грузов.

18 **Формирование состава морских контейнеровозов**

(a) Создайте класс `Vessel`, который будет представлять собой контейнеровоз с типом контейнера. В конструкторе класса `Vessel` инициализируйте значения судна и типа контейнера из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список судов (не менее 14):

`['Корабль_1', 'Корабль_2', ..., 'Корабль_14']`

`MasList: list[str]` — это список типов контейнеров (не менее 4):

`['20ft', '40ft', 'Рефрижератор', 'Открытый']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `VesselFleet`, который будет представлять собой флот контейнеровозов. В конструкторе класса `VesselFleet` инициализируйте список судов `self.train: list[Vessel]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `VesselFleet`, который будет перемешивать суда в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Vessel`, который будет возвращать i-е судно и тип его контейнера из списка `self.train`.

(e) Создайте экземпляр класса `VesselFleet` и вызовите метод `shuffle` для перемешивания судов.

(f) Создайте цикл, который будет запрашивать у пользователя номер судна и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все суда или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора судов.

(i) Убедитесь, что пользователь вводит корректные номера судов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров судов и типов контейнеров.

19 **Формирование состава банковских сейфов**

(a) Создайте класс `Safe`, который будет представлять собой сейф с содержимым. В конструкторе класса `Safe` инициализируйте значения сейфа и содержимого из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список сейфов (не менее 14):

`['Сейф_1', 'Сейф_2', ..., 'Сейф_14']`

`MasList: list[str]` — это список типов содержимого (не менее 4):

`['Документы', 'Драгоценности', 'Деньги', 'Антиквариат']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `VaultTrain`, который будет представлять собой состав сейфов. В конструкторе класса `VaultTrain` инициализируйте список сейфов `self.train: list[Safe]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `VaultTrain`, который будет перемешивать сейфы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Safe`, который будет возвращать i-й сейф и его содержимое из списка `self.train`.

(e) Создайте экземпляр класса `VaultTrain` и вызовите метод `shuffle` для перемешивания сейфов.

(f) Создайте цикл, который будет запрашивать у пользователя номер сейфа и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все сейфы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора сейфов.

(i) Убедитесь, что пользователь вводит корректные номера сейфов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров сейфов и содержимого.

20 **Формирование состава капсул экспресс-доставки**

(a) Создайте класс `Capsule`, который будет представлять собой капсулу с грузом. В конструкторе класса `Capsule` инициализируйте значения капсулы и груза из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список капсул (не менее 14):

`['Капсула_1', 'Капсула_2', ..., 'Капсула_14']`

`MasList: list[str]` — это список типов груза (не менее 4):

`['Медикаменты', 'Еда', 'Посылки', 'Образцы']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `CapsuleTrain`, который будет представлять собой состав капсул. В конструкторе класса `CapsuleTrain` инициализируйте список капсул `self.train: list[Capsule]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `CapsuleTrain`, который будет перемешивать капсулы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Capsule`, который будет возвращать i-ю капсулу и её груз из списка `self.train`.

(e) Создайте экземпляр класса `CapsuleTrain` и вызовите метод `shuffle` для перемешивания капсул.

(f) Создайте цикл, который будет запрашивать у пользователя номер капсулы и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все капсулы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора капсул.

(i) Убедитесь, что пользователь вводит корректные номера капсул и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров капсул и грузов.

21 **Формирование состава тележек в аэропорту**

(a) Создайте класс `Trolley`, который будет представлять собой тележку с типом пассажира. В конструкторе класса `Trolley` инициализируйте значения тележки и типа пассажира из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список тележек (не менее 14):

`['Тележка_1', 'Тележка_2', ..., 'Тележка_14']`

`MasList: list[str]` — это список типов пассажиров (не менее 4):

`['Бизнес', 'Эконом', 'Первый класс', 'Транзит']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `AirportTrolleyTrain`, который будет представлять собой состав тележек. В конструкторе класса `AirportTrolleyTrain` инициализируйте список тележек `self.train: list[Trolley]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `AirportTrolleyTrain`, который будет перемешивать тележки в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Trolley`, который будет возвращать i-ю тележку и тип её пассажира из списка `self.train`.

(e) Создайте экземпляр класса `AirportTrolleyTrain` и вызовите метод `shuffle` для перемешивания тележек.

(f) Создайте цикл, который будет запрашивать у пользователя номер тележки и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все тележки или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора тележек.

(i) Убедитесь, что пользователь вводит корректные номера тележек и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров тележек и типов пассажиров.

22 Формирование состава мобильных платформ с оборудованием

(a) Создайте класс `Platform`, который будет представлять собой платформу с оборудованием. В конструкторе класса `Platform` инициализируйте значения платформы и оборудования из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список платформ (не менее 14):

`['Платформа_1', 'Платформа_2', ..., 'Платформа_14']`

`MasList: list[str]` — это список типов оборудования (не менее 4):

`['Генератор', 'Компрессор', 'Насос', 'Сварочный аппарат']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `PlatformTrain`, который будет представлять собой состав платформ. В конструкторе класса `PlatformTrain` инициализируйте список платформ `self.train: list[Platform]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `PlatformTrain`, который будет перемешивать платформы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Platform`, который будет возвращать i-ю платформу и её оборудование из списка `self.train`.

(e) Создайте экземпляр класса `PlatformTrain` и вызовите метод `shuffle` для перемешивания платформ.

(f) Создайте цикл, который будет запрашивать у пользователя номер платформы и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все платформы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора платформ.

(i) Убедитесь, что пользователь вводит корректные номера платформ и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров платформ и оборудования.

23 **Формирование состава ящиков с инструментами**

(a) Создайте класс `Toolbox`, который будет представлять собой ящик с набором инструментов. В конструкторе класса `Toolbox` инициализируйте значения ящика и набора из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список ящиков (не менее 14):

`['Ящик_1', 'Ящик_2', ..., 'Ящик_14']`

`MasList: list[str]` — это список типов наборов (не менее 4):

`['Слесарный', 'Электромонтажный', 'Столярный', 'Автомобильный']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `ToolTrain`, который будет представлять собой состав ящиков. В конструкторе класса `ToolTrain` инициализируйте список ящиков `self.train: list[Toolbox]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `ToolTrain`, который будет перемешивать ящики в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Toolbox`, который будет возвращать i-й ящик и его набор инструментов из списка `self.train`.

(e) Создайте экземпляр класса `ToolTrain` и вызовите метод `shuffle` для перемешивания ящиков.

(f) Создайте цикл, который будет запрашивать у пользователя номер ящика и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все ящики или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора ящиков.

(i) Убедитесь, что пользователь вводит корректные номера ящиков и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров ящиков и наборов инструментов.

24 **Формирование состава подводных аппаратов**

(a) Создайте класс `Submersible`, который будет представлять собой аппарат с миссией. В конструкторе класса `Submersible` инициализируйте значения аппарата и миссии из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список аппаратов (не менее 14):

`[''Аппарат_1'', ''Аппарат_2'', ..., ''Аппарат_14'']`

`MasList: list[str]` — это список миссий (не менее 4):

`[''Исследование'', ''Спасение'', ''Инспекция'', ''Добыча'']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `SubmersibleSquadron`, который будет представлять собой эскадрилью аппаратов. В конструкторе класса `SubmersibleSquadron` инициализируйте список аппаратов `self.train: list[Submersible]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `SubmersibleSquadron`, который будет перемешивать аппараты в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Submersible`, который будет возвращать i-й аппарат и его миссию из списка `self.train`.

(e) Создайте экземпляр класса `SubmersibleSquadron` и вызовите метод `shuffle` для перемешивания аппаратов.

(f) Создайте цикл, который будет запрашивать у пользователя номер аппарата и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все аппараты или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора аппаратов.

(i) Убедитесь, что пользователь вводит корректные номера аппаратов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров аппаратов и миссий.

25 Формирование состава контейнеров с растениями

(a) Создайте класс `Planter`, который будет представлять собой контейнер с растением. В конструкторе класса `Planter` инициализируйте значения контейнера и растения из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список контейнеров (не менее 14):

`[''Контейнер_1'', ''Контейнер_2'', ..., ''Контейнер_14'']`

`MasList: list[str]` — это список растений (не менее 4):

`[''Орхидеи'', ''Кактусы'', ''Пальмы'', ''Бонсаи'']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `GreenTrain`, который будет представлять собой состав контейнеров. В конструкторе класса `GreenTrain` инициализируйте список контейнеров `self.train: list[Planter]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `GreenTrain`, который будет перемешивать контейнеры в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Planter`, который будет возвращать i-й контейнер и его растение из списка `self.train`.

(e) Создайте экземпляр класса `GreenTrain` и вызовите метод `shuffle` для перемешивания контейнеров.

(f) Создайте цикл, который будет запрашивать у пользователя номер контейнера и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все контейнеры или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора контейнеров.

(i) Убедитесь, что пользователь вводит корректные номера контейнеров и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров контейнеров и растений.

26 **Формирование состава машин скорой помощи**

(a) Создайте класс `Ambulance`, который будет представлять собой машину с типом бригады. В конструкторе класса `Ambulance` инициализируйте значения машины и типа бригады из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список машин (не менее 14):

```
['Машина_1', 'Машина_2', ..., 'Машина_14']
```

`MasList: list[str]` — это список типов бригад (не менее 4):

```
['Травматологи', 'Кардиологи', 'Психиатры', 'Реаниматологи']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `AmbulanceConvoy`, который будет представлять собой конвой машин. В конструкторе класса `AmbulanceConvoy` инициализируйте список машин `self.train: list[Ambulance]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `AmbulanceConvoy`, который будет перемешивать машины в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Ambulance`, который будет возвращать i-ю машину и её бригаду из списка `self.train`.

(e) Создайте экземпляр класса `AmbulanceConvoy` и вызовите метод `shuffle` для перемешивания машин.

(f) Создайте цикл, который будет запрашивать у пользователя номер машины и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все машины или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора машин.

(i) Убедитесь, что пользователь вводит корректные номера машин и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров машин и типов бригад.

27 **Формирование состава пожарных машин**

(a) Создайте класс `FireTruck`, который будет представлять собой пожарную машину со специализацией. В конструкторе класса `FireTruck` инициализируйте значения машины и специализации из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список машин (не менее 14):

`['Машина_1', 'Машина_2', ..., 'Машина_14']`

`MasList: list[str]` — это список специализаций (не менее 4):

`['Тушение', 'Спасение', 'Химзащита', 'Высотные работы']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `FireTrain`, который будет представлять собой состав пожарных машин. В конструкторе класса `FireTrain` инициализируйте список машин `self.train: list[FireTruck]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `FireTrain`, который будет перемешивать машины в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> FireTruck`, который будет возвращать i-ю машину и её специализацию из списка `self.train`.

(e) Создайте экземпляр класса `FireTrain` и вызовите метод `shuffle` для перемешивания машин.

(f) Создайте цикл, который будет запрашивать у пользователя номер машины и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все машины или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора машин.

(i) Убедитесь, что пользователь вводит корректные номера машин и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров машин и специализаций.

28 **Формирование состава эвакуаторов**

(a) Создайте класс `TowTruck`, который будет представлять собой эвакуатор с типом транспортного средства. В конструкторе класса `TowTruck` инициализируйте значения эвакуатора и типа ТС из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список эвакуаторов (не менее 14):

`['Эвакуатор_1', 'Эвакуатор_2', ..., 'Эвакуатор_14']`

`MasList: list[str]` — это список типов ТС (не менее 4):

`['Легковой', 'Грузовик', 'Мотоцикл', 'Автобус']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TowTrain`, который будет представлять собой состав эвакуаторов. В конструкторе класса `TowTrain` инициализируйте список эвакуаторов `self.train: list[TowTruck]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TowTrain`, который будет перемешивать эвакуаторы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> TowTruck`, который будет возвращать i-й эвакуатор и тип его ТС из списка `self.train`.

(e) Создайте экземпляр класса `TowTrain` и вызовите метод `shuffle` для перемешивания эвакуаторов.

(f) Создайте цикл, который будет запрашивать у пользователя номер эвакуатора и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все эвакуаторы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора эвакуаторов.

(i) Убедитесь, что пользователь вводит корректные номера эвакуаторов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров эвакуаторов и типов ТС.

29 **Формирование состава контейнеров с лекарствами**

(a) Создайте класс `MedBox`, который будет представлять собой контейнер с типом лекарств. В конструкторе класса `MedBox` инициализируйте значения контейнера и типа лекарств из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список контейнеров (не менее 14):

`['Контейнер_1', 'Контейнер_2', ..., 'Контейнер_14']`

`MasList: list[str]` — это список типов лекарств (не менее 4):

`['Антибиотики', 'Вакцины', 'Обезболивающие', 'Витамины']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `MedTrain`, который будет представлять собой состав контейнеров. В конструкторе класса `MedTrain` инициализируйте список контейнеров `self.train: list[MedBox]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `MedTrain`, который будет перемешивать контейнеры в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> MedBox`, который будет возвращать i-й контейнер и его лекарства из списка `self.train`.

(e) Создайте экземпляр класса `MedTrain` и вызовите метод `shuffle` для перемешивания контейнеров.

(f) Создайте цикл, который будет запрашивать у пользователя номер контейнера и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все контейнеры или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора контейнеров.

(i) Убедитесь, что пользователь вводит корректные номера контейнеров и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров контейнеров и типов лекарств.

30 **Формирование состава транспорта с опасными грузами**

(a) Создайте класс `HazmatTruck`, который будет представлять собой грузовик с классом опасности. В конструкторе класса `HazmatTruck` инициализируйте значения грузовика и класса опасности из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список грузовиков (не менее 14):

`['Грузовик_1', 'Грузовик_2', ..., 'Грузовик_14']`

`MasList: list[str]` — это список классов опасности (не менее 4):

`['Взрывчатка', 'Газы', 'Легковоспламеняющиеся', 'Токсичные']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `HazmatConvoy`, который будет представлять собой конвой грузовиков. В конструкторе класса `HazmatConvoy` инициализируйте список грузовиков `self.train: list[HazmatTruck]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `HazmatConvoy`, который будет перемешивать грузовики в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> HazmatTruck`, который будет возвращать i-й грузовик и его класс опасности из списка `self.train`.

(e) Создайте экземпляр класса `HazmatConvoy` и вызовите метод `shuffle` для перемешивания грузовиков.

(f) Создайте цикл, который будет запрашивать у пользователя номер грузовика и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все грузовики или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора грузовиков.

(i) Убедитесь, что пользователь вводит корректные номера грузовиков и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров грузовиков и классов опасности.

31 **Формирование состава курьерских пакетов**

(a) Создайте класс `Package`, который будет представлять собой пакет с типом доставки. В конструкторе класса `Package` инициализируйте значения пакета и типа доставки из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список пакетов (не менее 14):

`['Пакет_1', 'Пакет_2', ..., 'Пакет_14']`

`MasList: list[str]` — это список типов доставки (не менее 4):

`['Экспресс', 'Стандарт', 'Международный', 'Хрупкий']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `PackageTrain`, который будет представлять собой состав пакетов. В конструкторе класса `PackageTrain` инициализируйте список пакетов `self.train: list[Package]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `PackageTrain`, который будет перемешивать пакеты в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Package`, который будет возвращать i-й пакет и его тип доставки из списка `self.train`.

(e) Создайте экземпляр класса `PackageTrain` и вызовите метод `shuffle` для перемешивания пакетов.

(f) Создайте цикл, который будет запрашивать у пользователя номер пакета и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все пакеты или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора пакетов.

(i) Убедитесь, что пользователь вводит корректные номера пакетов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров пакетов и типов доставки.

32 **Формирование состава мобильных медицинских лабораторий**

(a) Создайте класс `LabVan`, который будет представлять собой лабораторию с типом анализа. В конструкторе класса `LabVan` инициализируйте значения лаборатории и типа анализа из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список лабораторий (не менее 14):

```
['Лаборатория_1', 'Лаборатория_2', ..., 'Лаборатория_14']
```

`MasList: list[str]` — это список типов анализов (не менее 4):

```
['PCR', 'Анализы крови', 'Токсикология', 'Микробиология']
```

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `LabConvoy`, который будет представлять собой конвой лабораторий. В конструкторе класса `LabConvoy` инициализируйте список лабораторий `self.train: list[LabVan]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `LabConvoy`, который будет перемешивать лаборатории в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> LabVan`, который будет возвращать i-ю лабораторию и её тип анализа из списка `self.train`.

(e) Создайте экземпляр класса `LabConvoy` и вызовите метод `shuffle` для перемешивания лабораторий.

(f) Создайте цикл, который будет запрашивать у пользователя номер лаборатории и выводить информацию о ней.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все лаборатории или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора лабораторий.

(i) Убедитесь, что пользователь вводит корректные номера лабораторий и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров лабораторий и типов анализов.

33 **Формирование состава контейнеров с артефактами**

(a) Создайте класс `ArtifactCase`, который будет представлять собой кейс с происхождением артефакта. В конструкторе класса `ArtifactCase` инициализируйте значения кейса и происхождения из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список кейсов (не менее 14):

`['Кейс_1', 'Кейс_2', ..., 'Кейс_14']`

`MasList: list[str]` — это список происхождений (не менее 4):

`['Египет', 'Греция', 'Мезоамерика', 'Древний Китай']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `ArtifactTrain`, который будет представлять собой состав кейсов. В конструкторе класса `ArtifactTrain` инициализируйте список кейсов `self.train: list[ArtifactCase]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `ArtifactTrain`, который будет перемешивать кейсы в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> ArtifactCase`, который будет возвращать i-й кейс и происхождение его артефакта из списка `self.train`.

(e) Создайте экземпляр класса `ArtifactTrain` и вызовите метод `shuffle` для перемешивания кейсов.

(f) Создайте цикл, который будет запрашивать у пользователя номер кейса и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все кейсы или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора кейсов.

(i) Убедитесь, что пользователь вводит корректные номера кейсов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров кейсов и происхождений.

34 Формирование состава беспилотных грузовиков

(a) Создайте класс `AutonomousTruck`, который будет представлять собой грузовик с типом маршрута. В конструкторе класса `AutonomousTruck` инициализируйте значения грузовика и маршрута из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список грузовиков (не менее 14):

`['Грузовик_1', 'Грузовик_2', ..., 'Грузовик_14']`

`MasList: list[str]` — это список типов маршрутов (не менее 4):

`['Город', 'Шоссе', 'Горы', 'Пустыня']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TruckConvoy`, который будет представлять собой конвой грузовиков. В конструкторе класса `TruckConvoy` инициализируйте список грузовиков `self.train: list[AutonomousTruck]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TruckConvoy`, который будет перемешивать грузовики в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> AutonomousTruck`, который будет возвращать i-й грузовик и его маршрут из списка `self.train`.

(e) Создайте экземпляр класса `TruckConvoy` и вызовите метод `shuffle` для перемешивания грузовиков.

(f) Создайте цикл, который будет запрашивать у пользователя номер грузовика и выводить информацию о нём.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все грузовики или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора грузовиков.

(i) Убедитесь, что пользователь вводит корректные номера грузовиков и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров грузовиков и маршрутов.

35 **Формирование состава грузовых вагонов** (оригинальный вариант)

(a) Создайте класс `Vagon`, который будет представлять собой вагон с грузом. В конструкторе класса `Vagon` инициализируйте значения вагона и груза из списков `NumList` и `MasList`, которые объявлены как общие атрибуты класса. `NumList: list[str]` — это список крытых вагонов (не менее 14):

`['Вагон_1', 'Вагон_2', ..., 'Вагон_14']`

`MasList: list[str]` — это список грузов для крытых вагонов (не менее 4):

`['Станки', 'Автозапчасти', 'Бумага', 'Керамическая плитка']`

Конструктор должен иметь сигнатуру: `__init__(self) -> None`.

(b) Создайте класс `TrainOfVagons`, который будет представлять собой грузовой поезд, состоящий из моделей вагонов. В конструкторе класса `TrainOfVagons` инициализируйте список вагонов `self.train: list[Vagon]` длиной 56.

(c) Добавьте метод `shuffle(self) -> None` в класс `TrainOfVagons`, который будет перемешивать вагоны в списке `self.train`.

(d) Добавьте метод `get(self, i: int) -> Vagon`, который будет возвращать i-й вагон и груз из списка `self.train`.

(e) Создайте экземпляр класса `TrainOfVagons` и вызовите метод `shuffle` для перемешивания вагонов.

(f) Создайте цикл, который будет запрашивать у пользователя номер вагона из поезда и выводить информацию о выбранном вагоне.

(g) Повторите шаги 5–6 до тех пор, пока пользователь не выберет все вагоны или не завершит выбор.

(h) В конце программы выводите сообщение о завершении выбора вагонов.

(i) Убедитесь, что пользователь вводит корректные номера вагонов и что программа обрабатывает ошибки, связанные с вводом пользователя.

(j) Проверьте работу программы, используя различные комбинации номеров вагонов и грузов.

2.5.2 Задача 2

1 **Расчёт площади стен в зависимости от наличия встроенных картин и зеркал**

(a) Создайте два класса: `PictureMirror` и `Room`. Класс `PictureMirror` представляет отдельный встроенный элемент (картину или зеркало). Его конструктор принимает два аргумента: `width` — ширина элемента в метрах (положительное дробное число), `height` — высота элемента в метрах (положительное дробное число). Объект этого класса хранит только эти два значения. Класс `Room` описывает прямоугольную комнату. Его конструктор принимает три аргумента: `width` — ширина комнаты в метрах, `length` — длина комнаты в метрах, `height` — высота стен в метрах. Все значения должны быть положительными. Объект `Room` хранит геометрические размеры комнаты и список объектов `PictureMirror`, изначально пустой.

(b) В классе `Room` реализуйте следующие методы:

- `add_item(self, item: PictureMirror) -> None` — добавляет переданный объект `PictureMirror` в внутренний список встроенных элементов комнаты. Метод не проверяет, помещается ли элемент на стене; предполагается, что все элементы корректно размещены.
- `get_area_to_cover(self) -> float` — вычисляет и возвращает площадь стен, подлежащую отделке. Общая площадь стен комнаты рассчитывается по формуле $2 \cdot$ `height` $\cdot$ (`width` + `length`). Из этой площади вычитается суммарная площадь всех встроенных элементов (каждый элемент вносит вклад `width` $\cdot$ `height`). Результат не может быть отрицательным: если суммарная площадь элементов превышает площадь стен, метод возвращает 0.0.
- `get_panels_count(self, panel_width: float, panel_height: float) -> int` — рассчитывает минимальное количество декоративных панелей, необходимых для отделки вычисленной ранее площади. Площадь одной панели равна `panel_width`$\cdot$`panel_height`. Количество панелей определяется как результат деления площади под отделку на площадь одной панели, округлённый вверх до ближайшего целого (поскольку панели продаются только целиком).

(c) Создайте три различных экземпляра класса `Room` с разными размерами и разным набором встроенных элементов (например, комната без элементов, комната с одной большой картиной, комната с несколькими зеркалами). Для каждого экземпляра вызовите методы `add_item` (при необходимости), `get_area_to_cover` и `get_panels_count`, чтобы продемонстрировать корректность реализации.

(d) Запросите у пользователя данные для одной комнаты: ширину, длину и высоту комнаты (все — дробные числа), а также ширину и высоту одной декоративной панели (дробные числа).

(e) Выведите на экран два значения: площадь стен под отделку (в квадратных метрах, с дробной частью) и минимальное количество необходимых панелей (целое число, округлённое вверх).

2 **Расчёт площади стен в зависимости от наличия встроенных настенных устройств**

(a) Создайте два класса: `WallDevice` и `Room`. Класс `WallDevice` представляет отдельное настенное устройство (например, панель управления). Его конструктор принимает два аргумента: `width` — ширина устройства в метрах (положительное

дробное число), `height` — высота устройства в метрах (положительное дробное число). Объект хранит только эти два значения. Класс `Room` описывает прямоугольную комнату. Его конструктор принимает три аргумента: `width` — ширина комнаты в метрах, `length` — длина комнаты в метрах, `height` — высота стен в метрах. Все значения должны быть положительными. Объект `Room` хранит размеры комнаты и список объектов `WallDevice`, изначально пустой.

(b) В классе `Room` реализуйте следующие методы:

- `add_device(self, dev: WallDevice) -> None` — добавляет переданный объект `WallDevice` в список встроенных устройств комнаты.
- `get_area_to_cover(self) -> float` — вычисляет площадь стен под отделку: из общей площади стен $2 \cdot$ `height` $\cdot$ (`width` + `length`) вычитается суммарная площадь всех устройств. Результат не может быть меньше нуля.
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — рассчитывает количество керамических плиток, необходимых для облицовки. Площадь одной плитки равна `tile_width` $\cdot$ `tile_height`. Количество плиток — это результат деления площади под отделку на площадь плитки, округлённый вверх до целого числа.

(c) Создайте три различных экземпляра класса `Room` с разными параметрами и разным числом устройств. Для каждого вызовите методы для получения площади и количества плиток.

(d) Запросите у пользователя размеры комнаты (ширина, длина, высота) и размеры одной плитки (ширина и высота), все — дробные числа.

(e) Выведите площадь стен под облицовку ($м^2$) и минимальное количество плиток (целое число, округлённое вверх).

3 Расчёт площади стен в зависимости от наличия встроенных светильников и бра

(a) Создайте два класса: `Lamp` и `Room`. Класс `Lamp` представляет один настенный светильник или бра. Его конструктор принимает: `width` — ширина светильника в метрах, `height` — высота светильника в метрах. Оба значения — положительные дробные числа. Класс `Room` описывает комнату. Его конструктор принимает: `width`, `length`, `height` — размеры комнаты в метрах (все положительные). Объект `Room` хранит эти размеры и список объектов `Lamp`.

(b) В классе `Room` реализуйте методы:

- `add_lamp(self, lamp: Lamp) -> None` — добавляет светильник в список встроенных элементов.
- `get_area_to_cover(self) -> float` — возвращает площадь стен без учёта площадей всех светильников. Общая площадь стен: $2 \cdot$ `height` $\cdot$ (`width` + `length`). Из неё вычитается сумма площадей всех светильников. Результат $\geqslant 0$.
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — вычисляет количество рулонов обоев. Площадь одного рулона: `roll_width` $\cdot$ `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` (с разным числом светильников) и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры одного рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов обоев (целое число, округлённое вверх).

4 **Расчёт площади стен в зависимости от наличия встроенных полок и стеллажей**

(a) Создайте два класса: `Shelf` и `Room`. Класс `Shelf` описывает одну полку или стеллаж. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (оба — положительные дробные числа). Класс `Room` описывает комнату с параметрами: `width`, `length`, `height` — размеры комнаты в метрах. Объект `Room` хранит список объектов `Shelf`.

(b) В классе `Room` реализуйте методы:

- `add_shelf(self, shelf: Shelf) -> None` — добавляет полку в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус суммарная площадь всех полок (результат $\geqslant 0$).
- `get_paint_liters(self, coverage: float) -> float` — вычисляет необходимый объём краски в литрах. Аргумент `coverage` задаёт, сколько квадратных метров можно покрыть одним литром краски (м^2/л). Объём краски = площадь под покраску / `coverage`. Результат может быть дробным, так как краску можно купить нецелыми банками.

(c) Создайте три различных комнаты с разным числом полок и проверьте работу методов.

(d) Запросите у пользователя размеры комнаты и значение `coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и необходимое количество литров краски (с дробной частью).

5 **Расчёт площади стен в зависимости от наличия встроенных розеток и выключателей**

(a) Создайте два класса: `SocketSwitch` и `Room`. Класс `SocketSwitch` представляет одну розетку или выключатель. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` (все — положительные дробные числа) и хранит список объектов `SocketSwitch`.

(b) В классе `Room` реализуйте методы:

- `add_electrical(self, el: SocketSwitch) -> None` — добавляет электроарматуру в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под штукатурку: общая площадь стен минус сумма площадей всех розеток и выключателей (результат $\geqslant 0$).
- `get_plaster_bags(self, bag_coverage: float) -> int` — количество мешков штукатурки. Аргумент `bag_coverage` — сколько квадратных метров покрывает один мешок (м^2/мешок). Количество мешков = площадь под штукатурку / `bag_coverage`, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом электроустройств и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и `bag_coverage` (дробное число).

(e) Выведите площадь под штукатурку (м²) и количество мешков (целое число, округлённое вверх).

6 **Расчёт площади стен в зависимости от наличия встроенных вентиляционных решёток**

(a) Создайте два класса: `VentGrille` и `Room`. Класс `VentGrille` описывает одну вентиляционную решётку. Его конструктор принимает: `width` — ширина решётки в метрах, `height` — высота решётки в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `VentGrille`.

(b) В классе `Room` реализуйте методы:

- `add_vent(self, vent: VentGrille) -> None` — добавляет решётку в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под обшивку: общая площадь стен минус сумма площадей всех решёток (результат $\geqslant 0$).
- `get_panel_sheets(self, sheet_width: float, sheet_height: float) -> int` — количество листов панелей. Площадь одного листа = `sheet_width`·`sheet_height`. Количество листов — частное от деления площади под обшивку на площадь листа, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом решёток и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры одного листа панели (все — дробные числа).

(e) Выведите площадь под обшивку (м²) и количество листов (целое число, округлённое вверх).

7 **Расчёт площади стен в зависимости от наличия встроенных настенных кондиционеров**

(a) Создайте два класса: `WallAC` и `Room`. Класс `WallAC` представляет один настенный кондиционер. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallAC`.

(b) В классе `Room` реализуйте методы:

- `add_ac(self, ac: WallAC) -> None` — добавляет кондиционер в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку обоями: общая площадь стен минус сумма площадей всех кондиционеров (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` · `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом кондиционеров и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

8 **Расчёт площади стен в зависимости от наличия встроенных настенных экранов**

(a) Создайте два класса: `WallScreen` и `Room`. Класс `WallScreen` описывает один настенный экран. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallScreen`.

(b) В классе `Room` реализуйте методы:

- `add_screen(self, scr: WallScreen) -> None` — добавляет экран в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех экранов (результат $\geqslant 0$).
- `get_paint_cans(self, can_coverage: float) -> int` — количество банок краски. Аргумент `can_coverage` — сколько квадратных метров покрывает одна банка (м^2/банка). Количество банок = площадь под покраску / `can_coverage`, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом экранов и проверьте работу методов.

(d) Запросите у пользователя размеры комнаты и `can_coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество банок (целое число, округлённое вверх).

9 **Расчёт площади стен в зависимости от наличия встроенных настенных сейфов**

(a) Создайте два класса: `WallSafe` и `Room`. Класс `WallSafe` представляет один настенный сейф. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallSafe`.

(b) В классе `Room` реализуйте методы:

- `add_safe(self, safe: WallSafe) -> None` — добавляет сейф в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под облицовку: общая площадь стен минус сумма площадей всех сейфов (результат $\geqslant 0$).
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — количество плиток. Площадь одной плитки = `tile_width` $\cdot$ `tile_height`. Количество плиток — частное от деления площади под облицовку на площадь плитки, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом сейфов и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры плитки (все — дробные числа).

(e) Выведите площадь под облицовку (м^2) и количество плиток (целое число, округлённое вверх).

10 **Расчёт площади стен в зависимости от наличия встроенных настенных досок**

(a) Создайте два класса: `WallBoard` и `Room`. Класс `WallBoard` описывает одну настенную доску. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallBoard`.

(b) В классе `Room` реализуйте методы:

- `add_board(self, board: WallBoard) -> None` — добавляет доску в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под драпировку: общая площадь стен минус сумма площадей всех досок (результат $\geqslant 0$).
- `get_fabric_meters(self, fabric_width: float) -> float` — необходимая длина ткани в метрах. Аргумент `fabric_width` — ширина ткани в метрах. Длина ткани = площадь под драпировку / `fabric_width`. Результат может быть дробным, так как ткань продаётся погонными метрами.

(c) Создайте три различных комнаты с разным числом досок и проверьте методы.

(d) Запросите у пользователя размеры комнаты и ширину ткани (дробные числа).

(e) Выведите площадь под драпировку (м^2) и количество метров ткани (с дробной частью).

11 **Расчёт площади стен в зависимости от наличия встроенных настенных календарей**

(a) Создайте два класса: `WallCalendar` и `Room`. Класс `WallCalendar` представляет один настенный календарь. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallCalendar`.

(b) В классе `Room` реализуйте методы:

- `add_calendar(self, cal: WallCalendar) -> None` — добавляет календарь в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех календарей (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` $\cdot$ `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом календарей и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

12 **Расчёт площади стен в зависимости от наличия встроенных настенных карт**

(a) Создайте два класса: `WallMap` и `Room`. Класс `WallMap` описывает одну настенную карту. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallMap`.

(b) В классе `Room` реализуйте методы:

- `add_map(self, map: WallMap) -> None` — добавляет карту в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех карт (результат $\geqslant 0$).
- `get_paint_liters(self, coverage: float) -> float` — объём краски в литрах. Аргумент `coverage` — расход краски (м²/л). Объём = площадь под покраску / `coverage`. Результат может быть дробным.

(c) Создайте три различных комнаты с разным числом карт и проверьте методы.

(d) Запросите у пользователя размеры комнаты и `coverage` (дробное число).

(e) Выведите площадь под покраску (м²) и количество литров краски (с дробной частью).

13 **Расчёт площади стен в зависимости от наличия встроенных настенных террариумов**

(a) Создайте два класса: `WallTerrarium` и `Room`. Класс `WallTerrarium` представляет один настенный террариум. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallTerrarium`.

(b) В классе `Room` реализуйте методы:

- `add_terrarium(self, terr: WallTerrarium) -> None` — добавляет террариум в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под отделку: общая площадь стен минус сумма площадей всех террариумов (результат $\geqslant 0$).
- `get_panels_count(self, panel_width: float, panel_height: float) -> int` — количество декоративных панелей. Площадь одной панели = `panel_width`$\cdot$ `panel_height`. Количество панелей — частное от деления площади под отделку на площадь панели, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом террариумов и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры панели (все — дробные числа).

(e) Выведите площадь под отделку (м²) и количество панелей (целое число, округлённое вверх).

14 **Расчёт площади стен в зависимости от наличия встроенных настенных аквариумов**

(a) Создайте два класса: `WallAquarium` и `Room`. Класс `WallAquarium` описывает один настенный аквариум. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallAquarium`.

(b) В классе `Room` реализуйте методы:

- `add_aquarium(self, aq: WallAquarium) -> None` — добавляет аквариум в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под облицовку: общая площадь стен минус сумма площадей всех аквариумов (результат $\geqslant 0$).
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — количество плиток. Площадь одной плитки = `tile_width` $\cdot$ `tile_height`. Количество плиток — частное от деления площади под облицовку на площадь плитки, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом аквариумов и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры плитки (все — дробные числа).

(e) Выведите площадь под облицовку (м^2) и количество плиток (целое число, округлённое вверх).

15 **Расчёт площади стен в зависимости от наличия встроенных настенных динамиков**

(a) Создайте два класса: `WallSpeaker` и `Room`. Класс `WallSpeaker` представляет один настенный динамик. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallSpeaker`.

(b) В классе `Room` реализуйте методы:

- `add_speaker(self, sp: WallSpeaker) -> None` — добавляет динамик в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех динамиков (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` $\cdot$ `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом динамиков и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

16 **Расчёт площади стен в зависимости от наличия встроенных настенных датчиков**

(a) Создайте два класса: `WallSensor` и `Room`. Класс `WallSensor` описывает один настенный датчик. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallSensor`.

(b) В классе `Room` реализуйте методы:

- `add_sensor(self, sens: WallSensor) -> None` — добавляет датчик в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех датчиков (результат $\geqslant 0$).
- `get_paint_cans(self, can_coverage: float) -> int` — количество банок краски. Аргумент `can_coverage` — покрытие одной банки (м^2/банка). Количество банок = площадь под покраску / `can_coverage`, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом датчиков и проверьте методы.

(d) Запросите у пользователя размеры комнаты и `can_coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество банок (целое число, округлённое вверх).

17 Расчёт площади стен в зависимости от наличия встроенных настенных панно

(a) Создайте два класса: `WallPanel` и `Room`. Класс `WallPanel` представляет одно настенное панно. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallPanel`.

(b) В классе `Room` реализуйте методы:

- `add_panel(self, p: WallPanel) -> None` — добавляет панно в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под драпировку: общая площадь стен минус сумма площадей всех панно (результат $\geqslant 0$).
- `get_fabric_meters(self, fabric_width: float) -> float` — длина ткани в метрах. Аргумент `fabric_width` — ширина ткани. Длина = площадь под драпировку / `fabric_width`. Результат может быть дробным.

(c) Создайте три разных экземпляра `Room` с разным числом панно и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и ширину ткани (дробные числа).

(e) Выведите площадь под драпировку (м^2) и количество метров ткани (с дробной частью).

18 Расчёт площади стен в зависимости от наличия встроенных настенных рамок

(a) Создайте два класса: `WallFrame` и `Room`. Класс `WallFrame` описывает одну настенную рамку. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallFrame`.

(b) В классе `Room` реализуйте методы:

- `add_frame(self, f: WallFrame) -> None` — добавляет рамку в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех рамок (результат $\geqslant 0$).

- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` · `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом рамок и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

19 **Расчёт площади стен в зависимости от наличия встроенных настенных витрин**

(a) Создайте два класса: `WallShowcase` и `Room`. Класс `WallShowcase` представляет одну настенную витрину. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallShowcase`.

(b) В классе `Room` реализуйте методы:

- `add_showcase(self, sc: WallShowcase) -> None` — добавляет витрину в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под облицовку: общая площадь стен минус сумма площадей всех витрин (результат $\geqslant 0$).
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — количество плиток. Площадь одной плитки = `tile_width` · `tile_height`. Количество плиток — частное от деления площади под облицовку на площадь плитки, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом витрин и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры плитки (все — дробные числа).

(e) Выведите площадь под облицовку (м^2) и количество плиток (целое число, округлённое вверх).

20 **Расчёт площади стен в зависимости от наличия встроенных настенных кронштейнов**

(a) Создайте два класса: `WallBracket` и `Room`. Класс `WallBracket` описывает один настенный кронштейн. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallBracket`.

(b) В классе `Room` реализуйте методы:

- `add_bracket(self, br: WallBracket) -> None` — добавляет кронштейн в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех кронштейнов (результат $\geqslant 0$).

- `get_paint_liters(self, coverage: float) -> float` — объём краски в литрах. Аргумент `coverage` — расход (м²/л). Объём = площадь под покраску / `coverage`. Результат может быть дробным.

(c) Создайте три различных комнаты с разным числом кронштейнов и проверьте методы.

(d) Запросите у пользователя размеры комнаты и `coverage` (дробное число).

(e) Выведите площадь под покраску (м²) и количество литров краски (с дробной частью).

21 **Расчёт площади стен в зависимости от наличия встроенных настенных жалюзи**

(a) Создайте два класса: `WallBlind` и `Room`. Класс `WallBlind` представляет одни настенные жалюзи. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallBlind`.

(b) В классе `Room` реализуйте методы:

- `add_blind(self, bl: WallBlind) -> None` — добавляет жалюзи в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под драпировку: общая площадь стен минус сумма площадей всех жалюзи (результат $\geqslant 0$).
- `get_fabric_meters(self, fabric_width: float) -> float` — длина ткани в метрах. Аргумент `fabric_width` — ширина ткани. Длина = площадь под драпировку / `fabric_width`. Результат может быть дробным.

(c) Создайте три разных экземпляра `Room` с разным числом жалюзи и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и ширину ткани (дробные числа).

(e) Выведите площадь под драпировку (м²) и количество метров ткани (с дробной частью).

22 **Расчёт площади стен в зависимости от наличия встроенных настенных флагов**

(a) Создайте два класса: `WallFlag` и `Room`. Класс `WallFlag` описывает один настенный флаг. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallFlag`.

(b) В классе `Room` реализуйте методы:

- `add_flag(self, fl: WallFlag) -> None` — добавляет флаг в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех флагов (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` · `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом флагов и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

23 Расчёт площади стен в зависимости от наличия встроенных грифельных досок

(a) Создайте два класса: `Chalkboard` и `Room`. Класс `Chalkboard` представляет одну грифельную доску. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Chalkboard`.

(b) В классе `Room` реализуйте методы:

- `add_board(self, cb: Chalkboard) -> None` — добавляет доску в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех досок (результат $\geqslant 0$).
- `get_paint_cans(self, can_coverage: float) -> int` — количество банок краски. Аргумент `can_coverage` — покрытие одной банки (м^2/банка). Количество банок = площадь под покраску / `can_coverage`, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом досок и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и `can_coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество банок (целое число, округлённое вверх).

24 Расчёт площади стен в зависимости от наличия встроенных маркерных досок

(a) Создайте два класса: `Whiteboard` и `Room`. Класс `Whiteboard` описывает одну маркерную доску. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Whiteboard`.

(b) В классе `Room` реализуйте методы:

- `add_board(self, wb: Whiteboard) -> None` — добавляет доску в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под отделку: общая площадь стен минус сумма площадей всех досок (результат $\geqslant 0$).
- `get_panels_count(self, panel_width: float, panel_height: float) -> int` — количество декоративных панелей. Площадь одной панели = `panel_width`$\cdot$ `panel_height`. Количество панелей — частное от деления площади под отделку на площадь панели, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом досок и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры панели (все — дробные числа).

(e) Выведите площадь под отделку (м^2) и количество панелей (целое число, округлённое вверх).

25 Расчёт площади стен в зависимости от наличия встроенных зеркал

(a) Создайте два класса: `Mirror` и `Room`. Класс `Mirror` представляет одно зеркало. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Mirror`.

(b) В классе `Room` реализуйте методы:

- `add_mirror(self, m: Mirror) -> None` — добавляет зеркало в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под облицовку: общая площадь стен минус сумма площадей всех зеркал (результат $\geqslant 0$).
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — количество плиток. Площадь одной плитки = `tile_width` $\cdot$ `tile_height`. Количество плиток — частное от деления площади под облицовку на площадь плитки, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом зеркал и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры плитки (все — дробные числа).

(e) Выведите площадь под облицовку (м^2) и количество плиток (целое число, округлённое вверх).

26 **Расчёт площади стен в зависимости от наличия встроенных часов**

(a) Создайте два класса: `WallClock` и `Room`. Класс `WallClock` описывает одни настенные часы. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallClock`.

(b) В классе `Room` реализуйте методы:

- `add_clock(self, cl: WallClock) -> None` — добавляет часы в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех часов (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` $\cdot$ `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом часов и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

27 **Расчёт площади стен в зависимости от наличия встроенных термометров**

(a) Создайте два класса: `Thermometer` и `Room`. Класс `Thermometer` представляет один настенный термометр. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Thermometer`.

(b) В классе `Room` реализуйте методы:

- `add_thermometer(self, t: Thermometer) -> None` — добавляет термометр в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех термометров (результат $\geqslant 0$).
- `get_paint_liters(self, coverage: float) -> float` — объём краски в литрах. Аргумент `coverage` — расход (м^2/л). Объём = площадь под покраску / `coverage`. Результат может быть дробным.

(c) Создайте три разных экземпляра `Room` с разным числом термометров и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и `coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество литров краски (с дробной частью).

28 **Расчёт площади стен в зависимости от наличия встроенных барометров**

(a) Создайте два класса: `Barometer` и `Room`. Класс `Barometer` описывает один настенный барометр. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Barometer`.

(b) В классе `Room` реализуйте методы:

- `add_barometer(self, b: Barometer) -> None` — добавляет барометр в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под драпировку: общая площадь стен минус сумма площадей всех барометров (результат $\geqslant 0$).
- `get_fabric_meters(self, fabric_width: float) -> float` — длина ткани в метрах. Аргумент `fabric_width` — ширина ткани. Длина = площадь под драпировку / `fabric_width`. Результат может быть дробным.

(c) Создайте три различных комнаты с разным числом барометров и проверьте методы.

(d) Запросите у пользователя размеры комнаты и ширину ткани (дробные числа).

(e) Выведите площадь под драпировку (м^2) и количество метров ткани (с дробной частью).

29 **Расчёт площади стен в зависимости от наличия встроенных гидрометров**

(a) Создайте два класса: `Hygrometer` и `Room`. Класс `Hygrometer` представляет один настенный гидрометр. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `Hygrometer`.

(b) В классе `Room` реализуйте методы:

- `add_hygrometer(self, h: Hygrometer) -> None` — добавляет гидрометр в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех гидрометров (результат $\geqslant 0$).

- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` · `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом гидрометров и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

30 **Расчёт площади стен в зависимости от наличия встроенных настенных растений**

(a) Создайте два класса: `WallPlant` и `Room`. Класс `WallPlant` описывает одно настенное растение (в кашпо или модуле). Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallPlant`.

(b) В классе `Room` реализуйте методы:

- `add_plant(self, p: WallPlant) -> None` — добавляет растение в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех растений (результат $\geqslant 0$).
- `get_paint_cans(self, can_coverage: float) -> int` — количество банок краски. Аргумент `can_coverage` — покрытие одной банки (м^2/банка). Количество банок = площадь под покраску / `can_coverage`, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом растений и проверьте методы.

(d) Запросите у пользователя размеры комнаты и `can_coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество банок (целое число, округлённое вверх).

31 **Расчёт площади стен в зависимости от наличия встроенных настенных фонарей**

(a) Создайте два класса: `WallLantern` и `Room`. Класс `WallLantern` представляет один настенный фонарь. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallLantern`.

(b) В классе `Room` реализуйте методы:

- `add_lantern(self, l: WallLantern) -> None` — добавляет фонарь в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под отделку: общая площадь стен минус сумма площадей всех фонарей (результат $\geqslant 0$).
- `get_panels_count(self, panel_width: float, panel_height: float) -> int` — количество декоративных панелей. Площадь одной панели = `panel_width` · `panel_height`. Количество панелей — частное от деления площади под отделку на площадь панели, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом фонарей и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры панели (все — дробные числа).

(e) Выведите площадь под отделку (м^2) и количество панелей (целое число, округлённое вверх).

32 Расчёт площади стен в зависимости от наличия встроенных настенных вентиляторов

(a) Создайте два класса: `WallFan` и `Room`. Класс `WallFan` описывает один настенный вентилятор. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallFan`.

(b) В классе `Room` реализуйте методы:

- `add_fan(self, f: WallFan) -> None` — добавляет вентилятор в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под облицовку: общая площадь стен минус сумма площадей всех вентиляторов (результат $\geqslant 0$).
- `get_tiles_count(self, tile_width: float, tile_height: float) -> int` — количество плиток. Площадь одной плитки = `tile_width` $\cdot$ `tile_height`. Количество плиток — частное от деления площади под облицовку на площадь плитки, округлённое вверх до целого.

(c) Создайте три различных комнаты с разным числом вентиляторов и проверьте методы.

(d) Запросите у пользователя размеры комнаты и размеры плитки (все — дробные числа).

(e) Выведите площадь под облицовку (м^2) и количество плиток (целое число, округлённое вверх).

33 Расчёт площади стен в зависимости от наличия встроенных увлажнителей

(a) Создайте два класса: `WallHumidifier` и `Room`. Класс `WallHumidifier` представляет один настенный увлажнитель. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallHumidifier`.

(b) В классе `Room` реализуйте методы:

- `add_humidifier(self, h: WallHumidifier) -> None` — добавляет увлажнитель в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех увлажнителей (результат $\geqslant 0$).
- `get_wallpaper_rolls(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь рулона = `roll_width` $\cdot$ `roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом увлажнителей и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов (целое число, округлённое вверх).

34 Расчёт площади стен в зависимости от наличия встроенных обогревателей

(a) Создайте два класса: `WallHeater` и `Room`. Класс `WallHeater` описывает один настенный обогреватель. Его конструктор принимает: `width` — ширина в метрах, `height` — высота в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WallHeater`.

(b) В классе `Room` реализуйте методы:

- `add_heater(self, h: WallHeater) -> None` — добавляет обогреватель в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под покраску: общая площадь стен минус сумма площадей всех обогревателей (результат $\geqslant 0$).
- `get_paint_liters(self, coverage: float) -> float` — объём краски в литрах. Аргумент `coverage` — расход (м^2/л). Объём = площадь под покраску / `coverage`. Результат может быть дробным.

(c) Создайте три различных комнаты с разным числом обогревателей и проверьте методы.

(d) Запросите у пользователя размеры комнаты и `coverage` (дробное число).

(e) Выведите площадь под покраску (м^2) и количество литров краски (с дробной частью).

35 Расчёт площади стен в зависимости от наличия окон и дверей

(a) Создайте два класса: `WinDoor` и `Room`. Класс `WinDoor` представляет один проём (окно или дверь). Его конструктор принимает: `width` — ширина проёма в метрах, `height` — высота проёма в метрах (положительные дробные числа). Класс `Room` описывает комнату с размерами `width`, `length`, `height` и хранит список объектов `WinDoor`.

(b) В классе `Room` реализуйте методы:

- `add_windoor(self, wd: WinDoor) -> None` — добавляет проём в комнату.
- `get_area_to_cover(self) -> float` — площадь стен под оклейку: общая площадь стен минус сумма площадей всех проёмов (результат $\geqslant 0$).
- `get_rolls_count(self, roll_width: float, roll_length: float) -> int` — количество рулонов обоев. Площадь одного рулона = `roll_width`·`roll_length`. Количество рулонов — частное от деления площади под оклейку на площадь рулона, округлённое вверх до целого.

(c) Создайте три разных экземпляра `Room` с разным числом проёмов и протестируйте методы.

(d) Запросите у пользователя размеры комнаты и размеры рулона обоев (все — дробные числа).

(e) Выведите площадь под оклейку (м^2) и количество рулонов обоев (целое число, округлённое вверх).

2.5.3 Задача 3

1. **Моделирование поединка между двумя дуэлянтами**

   (a) Импортируйте функцию `randint` из модуля `random`.

   (b) Создайте класс `Duelist` («Дуэлянт»). В конструкторе класса должны задаваться имя дуэлянта и его начальное здоровье (по умолчанию — 100 единиц). Также реализуйте следующие методы:

   - `set_name` — позволяет изменить имя дуэлянта;
   - `attack` — моделирует атаку на другого дуэлянта: генерирует случайный урон в диапазоне от 10 до 30 и уменьшает здоровье противника на эту величину.

   (c) Создайте класс `Duel` («Дуэль»). Его конструктор принимает двух дуэлянтов и сохраняет их как внутренние атрибуты. Также в конструкторе инициализируется пустая строка для хранения результата поединка.

   (d) Реализуйте метод `fight`, который моделирует сам поединок:

   - Поединок продолжается, пока у обоих дуэлянтов здоровье больше нуля;
   - На каждом шаге случайным образом (с равной вероятностью) выбирается, кто из дуэлянтов наносит удар;
   - После каждой атаки, если здоровье любого из участников стало меньше или равно нулю, оно устанавливается в ноль.

   (e) После завершения поединка определите его исход:

   - Если у первого дуэлянта осталось здоровье, а у второго — нет, побеждает первый;
   - Если у второго осталось здоровье, а у первого — нет, побеждает второй;
   - Если здоровье обоих участников равно нулю, объявляется ничья.

   Результат сохраняется в виде понятной строки (например, «Алексей побеждает!» или «Ничья!»).

   (f) Добавьте метод `who_wins`, который выводит на экран сохранённый результат поединка.

2. **Моделирование боя между двумя боксёрами**

   (a) Импортируйте функцию `randint` из модуля `random`.

   (b) Создайте класс `Boxer` («Боксёр»). В конструкторе задаются имя и начальное здоровье (по умолчанию — 100). Реализуйте методы:

   - `set_name` — изменение имени боксёра;
   - `punch` — нанесение удара противнику с уроном от 10 до 30.

   (c) Создайте класс `BoxingMatch` («Боксёрский поединок»). Его конструктор принимает двух боксёров и инициализирует атрибут для хранения результата.

   (d) Реализуйте метод `match`, моделирующий бой по тем же правилам, что и в первом задании: случайный выбор атакующего, цикл до тех пор, пока у одного из участников не закончится здоровье, коррекция здоровья до нуля при необходимости.

   (e) После завершения боя определите победителя или ничью и сохраните результат в виде строки.

   (f) Добавьте метод `announce_winner`, выводящий результат на экран.

3. **Моделирование поединка между двумя шахматистами**

   (a) Импортируйте функцию `randint`.

   (b) Создайте класс `ChessPlayer` («Шахматист»). В конструкторе задаются имя и начальное здоровье (по умолчанию — 100). Реализуйте методы:

   - `set_name` — изменение имени;
   - `play_move` — «ход» в рамках метафорического интеллектуального поединка, наносящий урон от 10 до 30.

   (c) Создайте класс `ChessGame` («Шахматная партия»), принимающий двух шахматистов и хранящий результат.

   (d) Реализуйте метод `simulate`, моделирующий поединок: случайный выбор ходящего, цикл до обнуления здоровья одного или обоих участников.

   (e) Определите победителя или ничью по оставшемуся здоровью.

   (f) Добавьте метод `show_result`, выводящий итог партии.

4. **Моделирование схватки между двумя борцами**

   (a) Импортируйте функцию `randint`.

   (b) Создайте класс `Wrestler` («Борец») с именем и здоровьем (по умолчанию — 100). Методы:

   - `set_name` — изменение имени;
   - `grapple` — захват, наносящий урон от 10 до 30.

   (c) Создайте класс `WrestlingMatch` («Борцовский поединок»), принимающий двух борцов.

   (d) Реализуйте метод `compete`, моделирующий схватку по стандартной схеме: случайный выбор атакующего, цикл до поражения одного или обоих.

   (e) Определите исход схватки и сохраните его в виде строки.

   (f) Добавьте метод `declare_champion`, выводящий победителя.

5. **Моделирование битвы между двумя магами**

   (a) Импортируйте функцию `randint`.

   (b) Создайте класс `Mage` («Маг») с именем и здоровьем (по умолчанию — 100). Методы:

   - `set_name` — изменение имени;
   - `cast_spell` — заклинание, наносящее урон от 10 до 30.

   (c) Создайте класс `MagicDuel` («Магическая дуэль»), принимающий двух магов.

   (d) Реализуйте метод `duel`, моделирующий битву по стандартной логике.

   (e) Определите победителя или ничью.

   (f) Добавьте метод `reveal_winner`, выводящий результат.

6. **Моделирование поединка между двумя киберспортсменами**

   (a) Импортируйте функцию `randint`.

(b) Создайте класс `Gamer` («Киберспортсмен») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `make_move` — игровой ход в метафоре «кибер-боя», наносящий урон от 10 до 30.

(c) Создайте класс `EsportsMatch` («Кибертурнир»), принимающий двух игроков.

(d) Реализуйте метод `play`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `show_champion`, выводящий чемпиона.

7. **Моделирование соревнования между двумя пловцами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Swimmer` («Пловец») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `swim_lap` — заплыв в игровой интерпретации как «атака», наносящая урон от 10 до 30.

(c) Создайте класс `SwimRace` («Заплыв»), принимающий двух пловцов.

(d) Реализуйте метод `race`, моделирующий соревнование.

(e) Определите победителя или ничью.

(f) Добавьте метод `announce_medal`, выводящий результат.

8. **Моделирование боя между двумя роботами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `RobotFighter` («Боевой робот») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `strike` — удар, наносящий урон от 10 до 30.

(c) Создайте класс `RobotBattle` («Робобой»), принимающий двух роботов.

(d) Реализуйте метод `fight`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `display_result`, выводящий результат.

9. **Моделирование дуэли между двумя ковбоями**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Cowboy` («Ковбой») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `draw` — выстрел, наносящий урон от 10 до 30.

(c) Создайте класс `Showdown` («Разборка»), принимающий двух ковбоев.

(d) Реализуйте метод `shootout`, моделирующий перестрелку.

(e) Определите победителя или ничью.

(f) Добавьте метод `proclaim_winner`, выводящий результат.

10. **Моделирование битвы между двумя ниндзя**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Ninja` («Ниндзя») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `throw_shuriken` — бросок сюрикена, наносящий урон от 10 до 30.

(c) Создайте класс `NinjaClash` («Столкновение ниндзя»), принимающий двух бойцов.

(d) Реализуйте метод `clash`, моделирующий битву.

(e) Определите победителя или ничью.

(f) Добавьте метод `declare_victor`, выводящий результат.

11. **Моделирование поединка между двумя пиратами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Pirate` («Пират») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `sword_fight` — удар мечом, наносящий урон от 10 до 30.

(c) Создайте класс `PirateDuel` («Пиратская дуэль»), принимающий двух пиратов.

(d) Реализуйте метод `battle`, моделирующий схватку.

(e) Определите победителя или ничью.

(f) Добавьте метод `shout_winner`, выводящий результат.

12. **Моделирование схватки между двумя гладиаторами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Gladiator` («Гладиатор») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `attack_with_sword` — удар мечом, наносящий урон от 10 до 30.

(c) Создайте класс `ArenaFight` («Арена»), принимающий двух гладиаторов.

(d) Реализуйте метод `fight_to_death`, моделирующий бой до конца.

(e) Определите победителя или ничью.

(f) Добавьте метод `crowd_cheers`, выводящий результат.

13. **Моделирование поединка между двумя самураями**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Samurai` («Самурай») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `katana_strike` — удар катаной, наносящий урон от 10 до 30.

(c) Создайте класс `SamuraiDuel` («Самурайская дуэль»), принимающий двух самураев.

(d) Реализуйте метод `duel`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `bow_to_winner`, выводящий результат.

14. **Моделирование поединка между двумя драконами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Dragon` («Дракон») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `breathe_fire` — огненное дыхание, наносящее урон от 10 до 30.

(c) Создайте класс `DragonBattle` («Битва драконов»), принимающий двух драконов.

(d) Реализуйте метод `clash`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `roar_victory`, выводящий результат.

15. **Моделирование битвы между двумя титанами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Titan` («Титан») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `stomp` — топот, наносящий урон от 10 до 30.

(c) Создайте класс `TitanClash` («Столкновение титанов»), принимающий двух титанов.

(d) Реализуйте метод `battle`, моделирующий битву.

(e) Определите победителя или ничью.

(f) Добавьте метод `earth_shakes`, выводящий результат.

16. **Моделирование поединка между двумя рыцарями**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Knight` («Рыцарь») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `lance_charge` — рывок с копьём, наносящий урон от 10 до 30.

(c) Создайте класс `Joust` («Турнир»), принимающий двух рыцарей.

(d) Реализуйте метод `tournament`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `king_declares`, выводящий результат.

17. **Моделирование поединка между двумя ведьмами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Witch` («Ведьма») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `brew_curse` — наложение проклятия, наносящего урон от 10 до 30.

(c) Создайте класс `WitchDuel` («Ведьмин поединок»), принимающий двух ведьм.

(d) Реализуйте метод `hex_battle`, моделирующий битву.

(e) Определите победителя или ничью.

(f) Добавьте метод `cackle_in_triumph`, выводящий результат.

18. **Моделирование боя между двумя зомби**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Zombie` («Зомби») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `bite` — укус, наносящий урон от 10 до 30.

(c) Создайте класс `ZombieFight` («Зомби-битва»), принимающий двух зомби.

(d) Реализуйте метод `apocalypse`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `groan_winner`, выводящий результат.

19. **Моделирование схватки между двумя вампирами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Vampire` («Вампир») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `drain` — высасывание жизненных сил, наносящее урон от 10 до 30.

(c) Создайте класс `VampireDuel` («Вампирская дуэль»), принимающий двух вампиров.

(d) Реализуйте метод `night_fight`, моделирующий ночную битву.

(e) Определите победителя или ничью.

(f) Добавьте метод `howl_at_moon`, выводящий результат.

20. **Моделирование битвы между двумя оборотнями**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Werewolf` («Оборотень») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `claw` — удар когтями, наносящий урон от 10 до 30.

(c) Создайте класс `MoonBattle` («Лунная битва»), принимающий двух оборотней.

(d) Реализуйте метод `howl_and_fight`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `moon_witnesses`, выводящий результат.

21. **Моделирование поединка между двумя призраками**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Ghost` («Призрак») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `terrify` — устрашение, наносящее урон от 10 до 30.

(c) Создайте класс `HauntedDuel` («Призрачная дуэль»), принимающий двух призраков.

(d) Реализуйте метод `scare_off`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `echo_victory`, выводящий результат.

22. **Моделирование боя между двумя гоблинами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Goblin` («Гоблин») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `stab` — удар кинжалом, наносящий урон от 10 до 30.

(c) Создайте класс `GoblinSkirmish` («Гоблинская стычка»), принимающий двух гоблинов.

(d) Реализуйте метод `loot_fight`, моделирующий драку.

(e) Определите победителя или ничью.

(f) Добавьте метод `squeal_winner`, выводящий результат.

23. **Моделирование схватки между двумя орками**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Orc` («Орк») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `bash` — мощный удар, наносящий урон от 10 до 30.

(c) Создайте класс `OrcBattle` («Орковская битва»), принимающий двух орков.

(d) Реализуйте метод `war_cry`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `grunt_victory`, выводящий результат.

24. **Моделирование битвы между двумя эльфами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Elf` («Эльф») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `arrow_shot` — выстрел из лука, наносящий урон от 10 до 30.

(c) Создайте класс `ElvenDuel` («Эльфийская дуэль»), принимающий двух эльфов.

(d) Реализуйте метод `forest_clash`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `whisper_winner`, выводящий результат.

25. **Моделирование поединка между двумя гномами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Dwarf` («Гном») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `swing_axe` — удар топором, наносящий урон от 10 до 30.

(c) Создайте класс `DwarfFight` («Гномья драка»), принимающий двух гномов.

(d) Реализуйте метод `mine_battle`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `roar_ale`, выводящий результат.

26. **Моделирование боя между двумя кентаврами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Centaur` («Кентавр») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `gallop_attack` — атака в галопе, наносящая урон от 10 до 30.

(c) Создайте класс `CentaurClash` («Столкновение кентавров»), принимающий двух кентавров.

(d) Реализуйте метод `plain_duel`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `neigh_victory`, выводящий результат.

27. **Моделирование схватки между двумя минотаврами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Minotaur` («Минотавр») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `gore` — удар рогами, наносящий урон от 10 до 30.

(c) Создайте класс `LabyrinthFight` («Лабиринтная битва»), принимающий двух минотавров.

(d) Реализуйте метод `maze_battle`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `bellow_winner`, выводящий результат.

28. **Моделирование битвы между двумя фениксами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Phoenix` («Феникс») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `rebirth_strike` — удар, связанный с возрождением, наносящий урон от 10 до 30.

(c) Создайте класс `PhoenixClash` («Столкновение фениксов»), принимающий двух фениксов.

(d) Реализуйте метод `ash_duel`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `soar_victorious`, выводящий результат.

29. **Моделирование поединка между двумя единорогами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Unicorn` («Единорог») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `horn_charge` — удар рогом, наносящий урон от 10 до 30.

(c) Создайте класс `UnicornDuel` («Дуэль единорогов»), принимающий двух единорогов.

(d) Реализуйте метод `meadow_clash`, моделирующий поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `gallop_in_glory`, выводящий результат.

30. **Моделирование боя между двумя троллями**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Troll` («Тролль») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `club_smash` — удар дубиной, наносящий урон от 10 до 30.

(c) Создайте класс `TrollFight` («Троллья драка»), принимающий двух троллей.

(d) Реализуйте метод `bridge_battle`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `grunt_and_laugh`, выводящий результат.

31. **Моделирование схватки между двумя грифонами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Griffin` («Грифон») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `dive_attack` — пикирующая атака, наносящая урон от 10 до 30.

(c) Создайте класс `GriffinClash` («Столкновение грифонов»), принимающий двух грифонов.

(d) Реализуйте метод `aerial_duel`, моделирующий воздушный поединок.

(e) Определите победителя или ничью.

(f) Добавьте метод `screech_victory`, выводящий результат.

32. **Моделирование битвы между двумя драконоборцами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Dragonslayer` («Драконоборец») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `slay` — удар, направленный на убийство, наносящий урон от 10 до 30.

(c) Создайте класс `SlayerDuel` («Дуэль драконоборцев»), принимающий двух героев.

(d) Реализуйте метод `heroic_fight`, моделирующий битву.

(e) Определите победителя или ничью.

(f) Добавьте метод `bard_sings`, выводящий результат.

33. **Моделирование поединка между двумя наёмниками**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Mercenary` («Наёмник») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `strike_for_hire` — удар за плату, наносящий урон от 10 до 30.

(c) Создайте класс `MercenaryClash` («Стычка наёмников»), принимающий двух бойцов.

(d) Реализуйте метод `contract_battle`, моделирующий сражение.

(e) Определите победителя или ничью.

(f) Добавьте метод `count_coins`, выводящий результат.

34. **Моделирование боя между двумя ассасинами**

(a) Импортируйте функцию `randint`.

(b) Создайте класс `Assassin` («Ассасин») с именем и здоровьем (по умолчанию — 100). Методы:

- `set_name` — изменение имени;
- `backstab` — удар в спину, наносящий урон от 10 до 30.

(c) Создайте класс `ShadowDuel` («Теневая дуэль»), принимающий двух ассасинов.

(d) Реализуйте метод `night_kill`, моделирующий ночной бой.

(e) Определите победителя или ничью.

(f) Добавьте метод `vanish_in_dark`, выводящий результат.

35. **Моделирование сражения между двумя солдатами** (оригинальный вариант)

(a) Импортируйте функцию `randint` из модуля `random`.

(b) Создайте класс `Soldier` («Солдат»). В конструкторе задаются имя и начальное здоровье (по умолчанию — 100). Реализуйте методы:

- `set_name` — изменение имени;
- `attack` — атака противника с уроном от 10 до 30.

(c) Создайте класс `Battle` («Сражение»), принимающий двух солдат и хранящий результат.

(d) Реализуйте метод `battle`, моделирующий бой:

- Поединок продолжается, пока у обоих солдат здоровье больше нуля;
- Атакующий выбирается случайно;
- После каждой атаки здоровье, упавшее до нуля или ниже, устанавливается в ноль.

(e) После завершения боя определите исход: победа одного из солдат или ничья — и сохраните результат в виде строки.

(f) Добавьте метод `who_win`, выводящий результат на экран.

2.6 Семинар «Ограничения доступа и Unit-тестирование» (2 часа)

В ходе работы решите 2 задачи.

Первое задание предполагает просто описание способов доступа к свойствам и методам различными способами.

Второе задание – реализацию простого класса и unit-тестов для него.

2.6.1 Принципы unit-тестирования в Python

Unit-тестирование позволяет проверять отдельные части кода — функции, методы или классы. Основные принципы:

- Каждый тест проверяет **одну конкретную функциональность**.
- Тесты должны покрывать **все важные сценарии использования**, включая краевые и граничные значения.
- Тесты должны быть **повторяемыми и независимыми** друг от друга.
- Используются утверждения: `assertEqual`, `assertTrue`, `assertFalse`, `assertRaises`.

2.6.2 Как анализировать код для тестирования всех случаев

При разработке unit-тестов важно систематически анализировать код и выявлять все ветви и варианты поведения:

1. **Анализ условных операторов (if/else)**: Для каждого условия нужно проверить как «истинный» путь, так и «ложный». Пример:

   ```
   def divide(a, b):
       if b == 0:
           raise ValueError("Division by zero")
       return a / b
   ```

   Тесты должны проверять:

   - деление на ненулевое число (if=False)
   - деление на ноль (if=True)

2. **Анализ циклов (for/while)**: Циклы проверяются на:

   - пустой вход (0 итераций)
   - одну итерацию
   - несколько итераций
   - граничные случаи (максимально допустимое число элементов)

3. **Граничные значения (boundary values)**: Любой метод, работающий с числами или индексами, должен проверяться на:

   - минимальные допустимые значения
   - максимальные допустимые значения

- ноль и отрицательные значения (если применимо)

4. **Исключения и ошибки**: Нужно проверять, что код корректно реагирует на некорректные входные данные, выбрасывая ожидаемые исключения.

5. **Комбинации входных данных**: Для методов с несколькими параметрами важно проверять сочетания «нормальных» и «краевых» значений.

2.6.3 Пример простого класса с unit-тестами

Рассмотрим класс `Calculator`, который выполняет сложение и деление чисел:

```
# calculator.py
class Calculator:
    def add(self, a, b):
        return a + b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Division by zero")
        return a / b
```

```
# test_calculator.py
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):

    def setUp(self):
        self.calc = Calculator()

    # Проверка всех важных случаев для сложения
    def test_add_positive_numbers(self):
        self.assertEqual(self.calc.add(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(self.calc.add(-2, -3), -5)

    def test_add_zero(self):
        self.assertEqual(self.calc.add(0, 5), 5)
        self.assertEqual(self.calc.add(5, 0), 5)

    # Проверка всех важных случаев для деления
    def test_divide_normal(self):
        self.assertEqual(self.calc.divide(10, 2), 5)

    def test_divide_fraction(self):
        self.assertEqual(self.calc.divide(1, 2), 0.5)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
```

```
            self.calc.divide(5, 0)

if __name__ == '__main__':
    unittest.main()
```

Объяснение:

- `setUp()` создает объект перед каждым тестом.
- Каждый метод, имя которого начинается с `test_`, проверяет отдельный сценарий.
- Мы покрыли:
  - положительные и отрицательные числа
  - ноль
  - дробные значения
  - исключения (деление на ноль)
- Для более сложного кода нужно аналогично анализировать все условия и ветвления.

Для сдачи работы будьте готовы пояснить или модифицировать любую часть кода, а также ответить на вопросы:

1. Как можно обеспечить инкапсуляцию в Python (перечислите все варианты)

Если вы нашли в задачнике ошибки, опечатки и другие недостатки, то вы можете сделать pull-request.

2.6.4 Задача 1

1 Разработать класс `Bus`, который будет описывать модель автобуса. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения автобуса
- `__distance`: расстояние, которое автобус проехал
- `__max_speed`: максимальная разрешённая скорость движения автобуса
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в автобусе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в автобусе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж автобуса

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mybus1`, `mybus2`, `mybus3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mybus3._Bus__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра автобуса, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

2 Разработать класс `Train`, который будет описывать модель поезда. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения поезда
- `__distance`: расстояние, которое поезд проехал
- `__max_speed`: максимальная разрешённая скорость движения поезда
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в поезде
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в поезде
- `__fuel_tank`: объём топливного бака (для дизельных поездов; для электрических — не применимо, но оставлено для единообразия)
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж поезда

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mytrain1`, `mytrain2`, `mytrain3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mytrain3._Train__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра поезда, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

3 Разработать класс `Airplane`, который будет описывать модель самолёта. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения самолёта
- `__distance`: расстояние, которое самолёт пролетел
- `__max_speed`: максимальная разрешённая скорость движения самолёта
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в самолёте
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в самолёте
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж самолёта

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.

- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myplane1`, `myplane2`, `myplane3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне

(включая `myplane3._Airplane__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра самолёта, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

4 Разработать класс `Ship`, который будет описывать модель корабля. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения корабля
- `__distance`: расстояние, которое корабль прошёл
- `__max_speed`: максимальная разрешённая скорость движения корабля
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на корабле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на корабле
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж корабля

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myship1`, `myship2`, `myship3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myship3._Ship__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра корабля, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

5 Разработать класс `Truck`, который будет описывать модель грузовика. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения грузовика

- `__distance`: расстояние, которое грузовик проехал
- `__max_speed`: максимальная разрешённая скорость движения грузовика
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в грузовике
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в грузовике
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж грузовика

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mytruck1`, `mytruck2`, `mytruck3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mytruck3._Truck__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра грузовика, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

6 Разработать класс `Motorcycle`, который будет описывать модель мотоцикла. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения мотоцикла
- `__distance`: расстояние, которое мотоцикл проехал
- `__max_speed`: максимальная разрешённая скорость движения мотоцикла
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на мотоцикле (обычно 1–2)
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на мотоцикле
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах

- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест (обычно сумки/кофры)
- `__luggage`: багаж мотоцикла

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mymoto1`, `mymoto2`, `mymoto3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
```

```
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mymoto3._Motorcycle__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра мотоцикла, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

7 Разработать класс `Bicycle`, который будет описывать модель велосипеда. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения велосипеда
- `__distance`: расстояние, которое велосипед проехал
- `__max_speed`: максимальная разрешённая скорость движения велосипеда
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на велосипеде (обычно 1, иногда 2)
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на велосипеде
- `__fuel_tank`: объём топливного бака (не применимо к обычному велосипеду; оставлено для единообразия)
- `__fuel`: количество топлива в литрах (обычно 0 для обычного велосипеда)
- `__engine_oil_capacity`: объём картера масла двигателя (не применимо; оставлено для единообразия)
- `__engine_oil`: количество моторного масла в литрах (обычно 0)
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж велосипеда

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными (обычно 0).

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mybike1`, `mybike2`, `mybike3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mybike3._Bicycle__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра велосипеда, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

8 Разработать класс `Helicopter`, который будет описывать модель вертолёта. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения вертолёта
- `__distance`: расстояние, которое вертолёт пролетел
- `__max_speed`: максимальная разрешённая скорость движения вертолёта
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в вертолёте
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в вертолёте
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж вертолёта

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myheli1`, `myheli2`, `myheli3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myheli3._Helicopter__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра вертолёта, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

9 Разработать класс `Submarine`, который будет описывать модель подводной лодки. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения подводной лодки
- `__distance`: расстояние, которое подводная лодка прошла
- `__max_speed`: максимальная разрешённая скорость движения подводной лодки
- `__passengers`: список пассажиров (обычно экипаж и, возможно, пассажиры)
- `__capacity`: максимальная вместимость пассажиров в подводной лодке
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в подводной лодке
- `__fuel_tank`: объём топливного бака (для дизель-электрических; для атомных — не применимо, но оставлено для единообразия)
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж подводной лодки

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.

- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mysub1`, `mysub2`, `mysub3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих

`@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mysub3._Submarine__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра подводной лодки, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

10 Разработать класс `Spaceship`, который будет описывать модель космического корабля. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения космического корабля
- `__distance`: расстояние, которое космический корабль пролетел
- `__max_speed`: максимальная разрешённая скорость движения космического корабля
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в космическом корабле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в космическом корабле
- `__fuel_tank`: объём топливного бака (для ракетного топлива)
- `__fuel`: количество топлива в литрах (или в условных единицах)
- `__engine_oil_capacity`: объём картера масла двигателя (не применимо к большинству космических двигателей; оставлено для единообразия)
- `__engine_oil`: количество моторного масла в литрах (обычно 0)
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж космического корабля

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.

- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myspace1`, `myspace2`, `myspace3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myspace3._Spaceship__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра космического корабля, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

11 Разработать класс `Drone`, который будет описывать модель дрона. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения дрона
- `__distance`: расстояние, которое дрон пролетел
- `__max_speed`: максимальная разрешённая скорость движения дрона
- `__passengers`: список пассажиров (обычно пустой; оставлено для единообразия)
- `__capacity`: максимальная вместимость пассажиров на дроне (обычно 0)
- `__empty_seats`: число свободных мест (обычно 0)
- `__seats_occupied`: число занятых мест на дроне (обычно 0)
- `__fuel_tank`: объём топливного бака (для топливных дронов; для электрических — не применимо)
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (обычно 0)
- `__engine_oil`: количество моторного масла в литрах (обычно 0)
- `__luggage_spaces`: количество багажных мест (для грузовых дронов)
- `__luggage`: багаж дрона

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mydrone1`, `mydrone2`, `mydrone3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mydrone3._Drone__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра дрона, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

12 Разработать класс `Scooter`, который будет описывать модель скутера. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения скутера
- `__distance`: расстояние, которое скутер проехал

- `__max_speed`: максимальная разрешённая скорость движения скутера
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на скутере (обычно 1–2)
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на скутере
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж скутера

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myscoot1`, `myscoot2`, `myscoot3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myscoot3._Scooter__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра скутера, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

13 Разработать класс `Taxi`, который будет описывать модель такси. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения такси
- `__distance`: расстояние, которое такси проехало
- `__max_speed`: максимальная разрешённая скорость движения такси
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в такси
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в такси
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах

- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж такси

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mytaxi1`, `mytaxi2`, `mytaxi3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
```

```
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mytaxi3._Taxi__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра такси, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

14 Разработать класс `Ambulance`, который будет описывать модель скорой помощи. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения скорой помощи
- `__distance`: расстояние, которое скорая помощь проехала
- `__max_speed`: максимальная разрешённая скорость движения скорой помощи
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в скорой помощи
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в скорой помощи
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж скорой помощи

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myamb1`, `myamb2`, `myamb3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myamb3._Ambulance__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра скорой помощи, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

15 Разработать класс `FireTruck`, который будет описывать модель пожарной машины. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения пожарной машины
- `__distance`: расстояние, которое пожарная машина проехала
- `__max_speed`: максимальная разрешённая скорость движения пожарной машины
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в пожарной машине
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в пожарной машине
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж пожарной машины

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myfire1`, `myfire2`, `myfire3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myfire3._FireTruck__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра пожарной машины, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

16 Разработать класс `PoliceCar`, который будет описывать модель полицейского автомобиля. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения полицейского автомобиля
- `__distance`: расстояние, которое полицейский автомобиль проехал
- `__max_speed`: максимальная разрешённая скорость движения полицейского автомобиля
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в полицейском автомобиле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в полицейском автомобиле
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж полицейского автомобиля

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.

- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mypolice1`, `mypolice2`, `mypolice3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих

`@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mypolice3._PoliceCar__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра полицейского автомобиля, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

17 Разработать класс `Crane`, который будет описывать модель подъёмного крана. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения крана
- `__distance`: расстояние, которое кран проехал
- `__max_speed`: максимальная разрешённая скорость движения крана
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в кране
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в кране
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж крана

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mycrane1`, `mycrane2`, `mycrane3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mycrane3._Crane__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра подъёмного крана, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

18 Разработать класс `Excavator`, который будет описывать модель экскаватора. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения экскаватора
- `__distance`: расстояние, которое экскаватор проехал
- `__max_speed`: максимальная разрешённая скорость движения экскаватора
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в экскаваторе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в экскаваторе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж экскаватора

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myex1`, `myex2`, `myex3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myex3._Excavator__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра экскаватора, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

19 Разработать класс `Tractor`, который будет описывать модель трактора. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения трактора
- `__distance`: расстояние, которое трактор проехал
- `__max_speed`: максимальная разрешённая скорость движения трактора

- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в тракторе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в тракторе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж трактора

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mytractor1`, `mytractor2`, `mytractor3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mytractor3._Tractor__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра трактора, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

20 Разработать класс `Snowmobile`, который будет описывать модель снегохода. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения снегохода
- `__distance`: расстояние, которое снегоход проехал
- `__max_speed`: максимальная разрешённая скорость движения снегохода
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на снегоходе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на снегоходе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах

- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж снегохода

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mysnow1`, `mysnow2`, `mysnow3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
```

```
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mysnow3._Snowmobile__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра снегохода, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

21 Разработать класс `ATV`, который будет описывать модель вездехода (quad bike). В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения вездехода
- `__distance`: расстояние, которое вездеход проехал
- `__max_speed`: максимальная разрешённая скорость движения вездехода
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на вездеходе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на вездеходе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж вездехода

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myatv1`, `myatv2`, `myatv3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myatv3._ATV__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра вездехода, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

22 Разработать класс `Hovercraft`, который будет описывать модель судна на воздушной подушке. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения судна на воздушной подушке
- `__distance`: расстояние, которое судно на воздушной подушке прошло
- `__max_speed`: максимальная разрешённая скорость движения судна на воздушной подушке
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на судне на воздушной подушке
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на судне на воздушной подушке
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж судна на воздушной подушке

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myhover1`, `myhover2`, `myhover3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myhover3._Hovercraft__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра судна на воздушной подушке, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

23 Разработать класс `Rocket`, который будет описывать модель ракеты. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения ракеты
- `__distance`: расстояние, которое ракета пролетела
- `__max_speed`: максимальная разрешённая скорость движения ракеты
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в ракете
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в ракете
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж ракеты

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.

- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myrocket1`, `myrocket2`, `myrocket3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне

(включая `myrocket3._Rocket__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра ракеты, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

24 Разработать класс `Glider`, который будет описывать модель планера. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения планера
- `__distance`: расстояние, которое планер пролетел
- `__max_speed`: максимальная разрешённая скорость движения планера
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в планере
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в планере
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж планера

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myglider1`, `myglider2`, `myglider3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myglider3._Glider__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра планера, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

25 Разработать класс `Zeppelin`, который будет описывать модель дирижабля. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения дирижабля

- `__distance`: расстояние, которое дирижабль пролетел
- `__max_speed`: максимальная разрешённая скорость движения дирижабля
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в дирижабле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в дирижабле
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж дирижабля

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myzep1`, `myzep2`, `myzep3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myzep3._Zeppelin__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра дирижабля, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

26 Разработать класс `Ferry`, который будет описывать модель парома. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения парома
- `__distance`: расстояние, которое паром прошёл
- `__max_speed`: максимальная разрешённая скорость движения парома
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на пароме
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на пароме
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах

- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж парома

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myferry1`, `myferry2`, `myferry3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
```

```
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myferry3._Ferry__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра парома, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

27 Разработать класс `Yacht`, который будет описывать модель яхты. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения яхты
- `__distance`: расстояние, которое яхта прошла
- `__max_speed`: максимальная разрешённая скорость движения яхты
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на яхте
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на яхте
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж яхты

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myyacht1`, `myyacht2`, `myyacht3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myyacht3._Yacht__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра яхты, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

28 Разработать класс `Speedboat`, который будет описывать модель быстроходной лодки. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения быстроходной лодки
- `__distance`: расстояние, которое быстроходная лодка прошла
- `__max_speed`: максимальная разрешённая скорость движения быстроходной лодки
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на быстроходной лодке
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на быстроходной лодке
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж быстроходной лодки

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myspeed1`, `myspeed2`, `myspeed3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myspeed3._Speedboat__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра быстроходной лодки, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

29 Разработать класс `CargoPlane`, который будет описывать модель грузового самолёта. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения грузового самолёта
- `__distance`: расстояние, которое грузовой самолёт пролетел
- `__max_speed`: максимальная разрешённая скорость движения грузового самолёта
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в грузовом самолёте
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в грузовом самолёте
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж грузового самолёта

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.

- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mycargo1`, `mycargo2`, `mycargo3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне

(включая `mycargo3._CargoPlane__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра грузового самолёта, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

30 Разработать класс `PassengerPlane`, который будет описывать модель пассажирского самолёта. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения пассажирского самолёта
- `__distance`: расстояние, которое пассажирский самолёт пролетел
- `__max_speed`: максимальная разрешённая скорость движения пассажирского самолёта
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в пассажирском самолёте
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в пассажирском самолёте
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж пассажирского самолёта

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mypass1`, `mypass2`, `mypass3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mypass3._PassengerPlane__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра пассажирского самолёта, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

31 Разработать класс `MetroCar`, который будет описывать модель вагона метро. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения вагона метро
- `__distance`: расстояние, которое вагон метро проехал
- `__max_speed`: максимальная разрешённая скорость движения вагона метро
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в вагоне метро
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в вагоне метро
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж вагона метро

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mymetro1`, `mymetro2`, `mymetro3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mymetro3._MetroCar__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра вагона метро, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

32 Разработать класс `Trolleybus`, который будет описывать модель троллейбуса. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения троллейбуса
- `__distance`: расстояние, которое троллейбус проехал
- `__max_speed`: максимальная разрешённая скорость движения троллейбуса

- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в троллейбусе
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в троллейбусе
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж троллейбуса

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `mytrol1`, `mytrol2`, `mytrol3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов `@property` и `@<имя>.setter`**: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля `accessify`**: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `mytrol3._Trolleybus__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра троллейбуса, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

33 Разработать класс `ElectricCar`, который будет описывать модель электромобиля. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения электромобиля
- `__distance`: расстояние, которое электромобиль проехал
- `__max_speed`: максимальная разрешённая скорость движения электромобиля
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров в электромобиле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест в электромобиле
- `__fuel_tank`: объём топливного бака

- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж электромобиля

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myev1`, `myev2`, `myev3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myev3._ElectricCar__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра электромобиля, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

34 Разработать класс `Hydrofoil`, который будет описывать модель гидрофойла. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения гидрофойла
- `__distance`: расстояние, которое гидрофойл прошёл
- `__max_speed`: максимальная разрешённая скорость движения гидрофойла
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на гидрофойле
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на гидрофойле
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест

- `__luggage`: багаж гидрофойла

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myhydro1`, `myhydro2`, `myhydro3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
```

```
raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myhydro3._Hydrofoil__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра гидрофойла, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

35 Разработать класс `Segway`, который будет описывать модель сигвея. В классе должны быть следующие поля с доступом уровня **private** (только внутри класса):

- `__speed`: скорость движения сигвея
- `__distance`: расстояние, которое сигвей проехал
- `__max_speed`: максимальная разрешённая скорость движения сигвея
- `__passengers`: список пассажиров
- `__capacity`: максимальная вместимость пассажиров на сигвее
- `__empty_seats`: число свободных мест
- `__seats_occupied`: число занятых мест на сигвее
- `__fuel_tank`: объём топливного бака
- `__fuel`: количество топлива в литрах
- `__engine_oil_capacity`: объём картера масла двигателя (литры)
- `__engine_oil`: количество моторного масла в литрах
- `__luggage_spaces`: количество багажных мест
- `__luggage`: багаж сигвея

Уровень доступа к полям должен быть следующим:

- `__max_speed`, `__capacity`, `__fuel_tank`, `__engine_oil_capacity`, `__luggage_spaces`: **только чтение** (через геттеры)
- `__speed`, `__distance`, `__passengers`, `__empty_seats`, `__seats_occupied`, `__fuel`, `__engine_oil`, `__luggage`: **чтение и запись** (через геттеры и сеттеры)

Требования к сеттерам:

- Для полей `__empty_seats` и `__seats_occupied` в сеттерах необходимо проверять, что передаваемое значение не превышает `__capacity` и неотрицательно.
- Для поля `__passengers` в сеттере необходимо проверять, что количество пассажиров (длина списка) не превышает `__capacity`.
- Для поля `__speed` в сеттере необходимо проверять, что заданная скорость не превышает `__max_speed` и неотрицательна.
- Для поля `__luggage` в сеттере необходимо проверять, что количество единиц багажа не превышает `__luggage_spaces`.
- Для полей `__fuel` и `__engine_oil` значения не должны превышать соответствующие ёмкости (`__fuel_tank` и `__engine_oil_capacity`) и должны быть неотрицательными.

Реализовать метод вывода всех установленных через сеттеры значений закрытых полей экземпляра класса. На основе этого класса реализовать три подхода к управлению доступом:

(a) **С использованием объекта** `property`: Для каждого поля определить отдельные методы-геттеры и сеттеры (например, `get_speed`, `set_speed`), а затем создать свойство:

```
speed = property(get_speed, set_speed)
```

Этот код должен располагаться после определения соответствующих методов. Первый аргумент — геттер, второй — сеттер. Продемонстрировать работу на трёх экземплярах класса: создать `myseg1`, `myseg2`, `myseg3`, установить значения через свойства и вывести их.

(b) **С использованием декораторов** `@property` **и** `@<имя>.setter`: Создать новую версию класса, в которой геттеры оформляются с декоратором `@property`, а сеттеры — с декоратором вида `@speed.setter`. Имена методов должны совпадать и не содержать префиксов `get_`/`set_`. Пример:

```
@property
def speed(self):
    return self.__speed
@speed.setter
def speed(self, value):
    if 0 <= value <= self.__max_speed:
        self.__speed = value
    else:
        raise ValueError("Недопустимая скорость")
```

Продемонстрировать работу на трёх экземплярах и сделать выводы об оптимизации кода по сравнению с первым подходом.

(c) **С использованием модуля** `accessify`: Установить модуль командой `pip install accessify` и импортировать:

```
from accessify import private, protected
```

Сделать поля `max_speed`, `capacity`, `fuel_tank`, `engine_oil_capacity`, `luggage_spaces` по-настоящему приватными с помощью функции `private` (например, как атрибуты класса до `__init__`). Удалить их из инициализатора. Проверки в сеттерах реализовать через вспомогательные методы, помеченные декоратором `@private`. Учитывать, что методы с `@private` нельзя вызывать из методов, использующих `@property`, поэтому для этой версии использовать только классические геттеры и сеттеры (`get_...`, `set_...`). Продемонстрировать, что попытка доступа извне (включая `myseg3._Segway__max_speed`) **не даёт результата**, а вызов приватного метода или чтение приватного поля вызывает ошибку доступа.

Для всех трёх подходов создать по три экземпляра сигвея, установить значения полей с учётом всех ограничений и вывести текущие значения всех полей каждого экземпляра.

2.6.5 Задача 2

Инструкция: Напишите функцию и соответствующие unit-тесты, покрывающие все важные случаи. Для заданий с ветвлениями (`if`/`elif`/`else`) обязательно проверяйте все ветви.

Пояснения:

- **Triangle types:**
  - `equilateral` — все стороны равны
  - `isosceles` — две стороны равны
  - `scalene` — все стороны разные
  - `invalid` — невозможно построить треугольник
- **BMI (Body Mass Index):** индекс массы тела. Категории: `Underweight`, `Normal`, `Overweight`, `Obese`
- **Palindrome:** строка или число, читающееся одинаково слева направо и справа налево
- **Perfect number:** число, равное сумме своих делителей, исключая само число
- **Triangle angles:**
  - `acute` — все углы $< 90°$
  - `right` — один угол $= 90°$
  - `obtuse` — один угол $> 90°$
  - `invalid` — треугольник не существует
- **Traffic fine:** штраф за превышение скорости. Функция должна учитывать разные зоны (`residential`, `city`, `highway`) и уровни превышения скорости.

1. `classify_triangle(a, b, c)` — возвращает тип треугольника.
2. `classify_number(n)` — возвращает ‘"positive even„, ‘"positive odd„, ‘"negative even„, ‘"negative odd„, ‘"zero„.
3. `middle_value(a, b, c)` — возвращает среднее (не арифметическое) число среди трёх, через сравнения.

4. `median_of_three(a, b, c)` — медиана трёх чисел через if/elif/else.
5. `is_leap_year(year)` — проверяет високосный год (делится на 4, но не на 100, или на 400).
6. `bmi_category(weight, height)` — возвращает категорию BMI.
7. `categorize_temperature(temp)` — диапазоны: ‘"freezing„ $\leqslant 0$°C, ‘"cold„ 1–10°C, ‘"cool„ 11–20°C, ‘"warm„ 21–30°C, ‘"hot„ >30°C.
8. `triangle_area_type(a, b, c)` — возвращает ‘"acute„„, ‘"right„„, ‘"obtuse„ или ‘"invalid„.
9. `quadrant(x, y)` — возвращает номер четверти (1–4) или ‘"origin„/‘"axis„.
10. `days_in_month(month, leap)` — возвращает число дней в месяце; `leap` = True для високосного года.
11. `traffic_fine(speed, zone)` — вычисляет штраф за превышение скорости.
    - **speed** — скорость автомобиля (км/ч)
    - **zone** — тип зоны: ‘"residential„„, ‘"city„„, ‘"highway„
    - **правила:**
      - ‘"residential„: превышение >20 км/ч → 200, >10 км/ч → 100, иначе 0
      - ‘"city„: превышение >30 → 150, >15 → 75, иначе 0
      - ‘"highway„: превышение >40 → 100, >20 → 50, иначе 0
      - некорректная зона → ‘"invalid zone„
    - **требования:** использовать ветвления if/elif/else, проверить все сценарии превышения и отсутствие превышения.
12. `compare_three_numbers(a, b, c)` — возвращает ‘"all equal„„, ‘"all different„„, или ‘"two equal„.
13. `max_digit(n)` — наибольшая цифра числа.
14. `triangle_angle_category(a, b, c)` — вычисляет углы через теорему косинусов и возвращает ‘"acute„„, ‘"right„„, ‘"obtuse„„, ‘"invalid„.
15. `next_day(day, month, leap)` — возвращает следующий день месяца, учитывая количество дней.
16. `is_inside_rectangle(x, y, x1, y1, x2, y2)` — проверка попадания точки в прямоугольник.
17. `discount(price)` — возврат цены со скидкой по условию (>1000 → 10
18. `closest_to_zero(lst)` — элемент списка, ближайший к нулю.
19. `season(month)` — ‘"Winter„„, ‘"Spring„„, ‘"Summer„„, ‘"Autumn„.
20. `simple_calculator(a, b, op)` — +, -, *, /; деление на 0 → ‘"error„.
21. `sum_positive(lst)` — сумма положительных чисел.
22. `sum_even(lst)` — сумма чётных чисел.

23. `sum_odd(lst)` — сумма нечётных чисел.
24. `reverse_signs(lst)` — меняет знак всех элементов.
25. `nearest_multiple(n, m)` — ближайшее к n кратное m.
26. `sort_three(a, b, c)` — тройка чисел в порядке возрастания через if/elif/else.
27. `validate_password(password)` — True, если длина $\geqslant$8, есть цифра и заглавная буква.
28. `is_perfect(n)` — True, если число совершенное.
29. `sum_digits(n)` — сумма цифр числа.
30. `count_vowels(s)` — количество гласных.
31. `is_palindrome(s)` — True, если строка читается одинаково слева направо и справа налево.
32. `remove_duplicates(lst)` — возвращает список без повторов.
33. `factorial_iterative(n)` — факториал через цикл, без рекурсии.
34. `fizz_buzz(n)` — числа от 1 до n с заменой кратных 3 → ‘"Fizz„, кратных 5 → ‘"Buzz„, кратных 15 → ‘"FizzBuzz„.