**1.1 Рассмотрите основные особенности обучения моделей машинного обучения на размеченных данных.**

Обучение моделей машинного обучения на размеченных данных - это процесс, в котором модель обучается на данных, которые уже были размечены людьми или другими алгоритмами. В этом процессе размеченные данные используются для обучения модели, чтобы она могла находить закономерности в данных и делать предсказания на новых данных.

Основные особенности обучения моделей машинного обучения на размеченных данных включают:

1.Качество разметки данных: Качество разметки данных может существенно влиять на процесс обучения и результаты модели. Если данные содержат ошибки в разметке, то это может привести к неправильным выводам и плохим результатам.

2.Разнообразие данных: Разнообразие данных может помочь модели обучаться на различных примерах и быть более гибкой в различных сценариях. Поэтому, чем больше разнообразных данных у модели, тем лучше ее обучение.

3.Размер данных: Размер данных имеет значение, так как больший объем данных может помочь модели обучаться лучше и уменьшить возможность переобучения. Однако, слишком большой объем данных может затруднить процесс обучения.

4.Выбор модели: Выбор модели также имеет значение. Различные модели могут иметь разные потребности в данных для обучения. Некоторые модели могут требовать больше данных, чем другие, чтобы достичь высокого уровня точности.

5.Разделение данных: Данные нужно разделить на обучающую, валидационную и тестовую выборки. Обучающая выборка используется для обучения модели, валидационная - для настройки параметров модели, а тестовая - для оценки ее точности и обобщающей способности.

6.Анализ результатов: Анализ результатов является важной частью процесса обучения модели. Он позволяет определить, насколько хорошо модель обучилась на данных, и выявить возможные проблемы в модели.

В целом, обучение моделей машинного обучения на размеченных данных требует тщательного подхода и анализа, чтобы достичь высокого уровня точности и обобщающей способности модели.

**1.2. Проанализируйте виды признаков, используемых в машинном обучении.**  
Признаки в машинном обучении - это характеристики объектов, которые используются для построения моделей и принятия решений. В зависимости от типа задачи и доступных данных, могут использоваться различные виды признаков. Ниже я рассмотрю основные виды признаков, используемых в машинном обучении:

1.Категориальные признаки: Это признаки, которые описывают качественные характеристики объектов, такие как цвет, форма, тип и т.д. Они могут быть представлены в виде текстовых меток или числовых кодов. Для обработки категориальных признаков обычно используются методы кодирования, такие как One-Hot Encoding или Label Encoding.

2.Числовые признаки: Это признаки, которые описывают количественные характеристики объектов, такие как размер, вес, стоимость и т.д. Числовые признаки могут быть дискретными (целые числа) или непрерывными (вещественные числа). Для работы с числовыми признаками обычно используются методы масштабирования, например, MinMax Scaling или Z-score Scaling.

3.Текстовые признаки: Это признаки, которые описывают текстовую информацию, такую как название продукта, описание товара, отзывы покупателей и т.д. Для работы с текстовыми признаками используются методы векторизации текста, такие как Bag-of-Words, TF-IDF или Word2Vec.

4.Географические признаки: Это признаки, которые описывают географическое положение объектов, такие как координаты местоположения, почтовый индекс и т.д. Географические признаки могут использоваться для анализа местоположения объектов или для определения расстояний между ними.

5.Временные признаки: Это признаки, которые описывают временные характеристики объектов, такие как дата, время, длительность и т.д. Временные признаки могут использоваться для анализа временных трендов или для прогнозирования будущих значений.

6.Изображения и видео: Это признаки, которые описывают визуальные характеристики объектов, такие как цвет, форма, текстура и т.д. Изображения и видео могут быть представлены в виде пикселей или векторов признаков, извлеченных с помощью методов обработки изображений, таких как сверточные нейронные сети или глубокие нейронные сети.

7.Звуковые данные: Это признаки, которые описывают акустические характеристики объектов, такие как частота, громкость, тембр и т.д. Звуковые данные могут быть представлены в виде амплитуд и частотных спектров или векторов признаков, извлеченных с помощью методов обработки звуковых сигналов, таких как мел-частотные кепстральные коэффициенты или спектрограммы.

8.Социальные данные: Это признаки, которые описывают социальные характеристики объектов, такие как возраст, пол, образование, профессия и т.д. Социальные данные могут использоваться для анализа поведения пользователей или для прогнозирования их предпочтений.

Кроме того, существуют и другие виды признаков, такие как генетические данные, биометрические данные и т.д. В целом, выбор типа признаков зависит от типа задачи, доступных данных и целей анализа.

**1.3. Рассмотрите возможности использования линейных моделей в задачах регрессии.**

Линейные модели являются одними из наиболее простых и широко используемых моделей в машинном обучении. Они могут быть эффективно применены в задачах регрессии, которые требуют предсказания непрерывного значения на основе набора входных признаков.

Основные типы линейных моделей, используемых в задачах регрессии, включают:

1.Простая линейная регрессия: Эта модель представляет собой простую линейную функцию, которая описывает линейную зависимость между одним входным признаком и выходным значением. Функция имеет вид y = w0 + w1x1, где w0 - константа (свободный член), w1 - коэффициент наклона и x1 - входной признак.

2.Множественная линейная регрессия: Эта модель представляет собой линейную функцию, которая описывает линейную зависимость между несколькими входными признаками и выходным значением. Функция имеет вид y = w0 + w1x1 + w2x2 + ... + wnxn, где w0 - константа, w1, w2, ..., wn - коэффициенты наклона и x1, x2, ..., xn - входные признаки.

3.Регуляризованные линейные модели: Эти модели являются модификациями простой или множественной линейной регрессии, в которых добавляется регуляризация для уменьшения переобучения. Два основных типа регуляризации, используемых в линейных моделях, - L1 (Lasso) и L2 (Ridge) регуляризация. L1 регуляризация добавляет штраф к модели за большие значения коэффициентов, что может привести к отбору признаков. L2 регуляризация добавляет квадратичный штраф к модели за большие значения коэффициентов, что позволяет снизить эффект выбросов в данных.

Линейные модели имеют ряд преимуществ в задачах регрессии, включая их простоту, интуитивную интерпретируемость, быстроту обучения и хорошую работу с большими объемами данных. Однако они могут не давать наилучших результатов в задачах, где входные данные не являются линейно зависимыми или когда имеется большое количество входных признаков, что может привести к проблеме мультиколлинеарности (когда признаки сильно коррелируют между собой). В таких случаях могут быть более эффективны другие типы моделей, например, деревья решений, случайный лес, градиентный бустинг и нейронные сети.

**1.4.Рассмотрите возможности использования линейных моделей в задачах классификации.**  
Линейные модели также могут быть эффективно применены в задачах классификации, которые требуют предсказания категориального значения на основе набора входных признаков. В зависимости от числа категорий в выходном значении, линейные модели могут быть применены как в задачах бинарной классификации, так и в многоклассовой классификации.

Основные типы линейных моделей, используемых в задачах классификации, включают:

1.Логистическая регрессия: Эта модель используется в задачах бинарной классификации и представляет собой линейную функцию, которая описывает логистическую зависимость между входными признаками и вероятностью принадлежности к одному из двух классов. Функция имеет вид p = 1 / (1 + exp(-z)), где z = w0 + w1x1 + w2x2 + ... + wnxn - линейная комбинация входных признаков и их коэффициентов, а p - вероятность принадлежности к положительному классу.

2.Множественная логистическая регрессия: Эта модель используется в задачах многоклассовой классификации и представляет собой линейную функцию, которая описывает логистическую зависимость между входными признаками и вероятностями принадлежности к различным классам. Функция имеет вид pi = exp(zi) / (sum(exp(zj))), где zi = w0 + w1x1 + w2x2 + ... + wnxn - линейная комбинация входных признаков и их коэффициентов, а pi - вероятность принадлежности к i-му классу.

3.Регуляризованные линейные модели: Эти модели являются модификациями линейных моделей с добавлением регуляризации для уменьшения переобучения. В задачах классификации, L1 (Lasso) и L2 (Ridge) регуляризация могут быть использованы для отбора признаков и уменьшения эффекта шума в данных.

Линейные модели в задачах классификации обладают такими же преимуществами, как и в задачах регрессии, такие как их простоту, интуитивную интерпретируемость и быстроту обучения. Однако, как ив задачах регрессии, линейные модели могут быть ограничены линейностью и неспособностью улавливать сложные нелинейные зависимости между входными признаками и выходными значениями. В таких случаях могут быть более эффективны другие типы моделей, например, деревья решений, случайный лес, градиентный бустинг и нейронные сети. Кроме того, линейные модели могут быть чувствительны к выбросам в данных и могут потребовать предварительной нормализации входных признаков для более эффективного обучения.

**1.5. Задача регрессии , предсказание значений непрерывной целевой переменной**

Задача регрессии в машинном обучении представляет собой задачу предсказания значений непрерывной целевой переменной на основе заданного набора входных признаков. Эта задача может быть использована для прогнозирования различных числовых показателей, таких как цены на недвижимость, уровень заработной платы, температура и т.д.

Регрессионные модели в машинном обучении могут быть как линейными, так и нелинейными. Линейные модели, такие как линейная регрессия, используют линейную комбинацию входных признаков и их весов для создания прогнозов. В этом случае выходное значение представляет собой линейную функцию от входных признаков и не зависит от порядка этих признаков. Нелинейные модели, такие как деревья решений, градиентный бустинг и нейронные сети, могут улавливать более сложные зависимости между входными признаками и выходными значениями.

Для решения задачи регрессии, модель должна быть обучена на наборе размеченных данных, где для каждого наблюдения известны значения входных признаков и соответствующее значение целевой переменной. Обучение модели включает в себя определение оптимальных значений параметров модели, которые минимизируют ошибку между прогнозами модели и реальными значениями целевой переменной. Ошибка обычно измеряется с помощью функции потерь, такой как средняя квадратичная ошибка (Mean Squared Error, MSE) или средняя абсолютная ошибка (Mean Absolute Error, MAE).

После обучения модели, ее можно использовать для создания прогнозов на новых данных, где значения входных признаков известны, но значение целевой переменной нужно предсказать. Прогнозы модели могут быть оценены с помощью метрик оценки качества модели, таких как коэффициент детерминации (R-squared) или среднеквадратичная ошибка (MSE).

**1.6. Стохастический градиентный спуск**

Стохастический градиентный спуск (Stochastic Gradient Descent, SGD) - это метод оптимизации, используемый для обучения моделей машинного обучения. В отличие от обычного градиентного спуска, который использует всю выборку данных для обновления параметров модели на каждой итерации, SGD использует только одно случайное наблюдение (пример) из выборки на каждой итерации.

Стохастический градиентный спуск имеет ряд преимуществ перед обычным градиентным спуском. Во-первых, он требует гораздо меньше вычислительных ресурсов, поскольку он обрабатывает только один пример на каждой итерации. Это делает SGD идеальным для обучения моделей на больших наборах данных. Во-вторых, SGD может работать с нелинейными функциями потерь, которые могут быть невычисляемыми для всей выборки данных. В-третьих, SGD может использоваться для обучения моделей в режиме онлайн, когда данные поступают последовательно и требуется быстро обновлять параметры модели с каждым новым наблюдением.

Стохастический градиентный спуск имеет несколько особенностей. Во-первых, поскольку он использует только один пример на каждой итерации, то скорость обучения может быть непостоянной и может колебаться в зависимости от случайности выборки примеров. Во-вторых, так как SGD обрабатывает случайные примеры, то в некоторых случаях обновления параметров модели могут быть шумными и нестабильными. В-третьих, настройка параметров SGD может быть сложной задачей и может потребовать дополнительных настроек, таких как скорости обучения и размера мини-пакетов.

Однако, стохастический градиентный спуск является одним из самых популярных методов оптимизации в машинном обучении и часто используется для обучения моделей глубокого обучения.

**1.7. K ближайших соседей**

K ближайших соседей (k-nearest neighbors, k-NN) - это метод обучения с учителем, который используется для классификации и регрессии. Он основан на идее о том, что близкие объекты находятся в одном классе или имеют похожие значения целевой переменной.

Для классификации, k-NN определяет класс объекта путем голосования большинства из k ближайших к нему объектов. То есть, если большинство из k ближайших соседей принадлежат к классу A, то объект также относится к классу A. Для регрессии, k-NN определяет значение целевой переменной путем усреднения значений целевой переменной k ближайших соседей.

Расстояние между объектами может быть определено различными способами, такими как евклидово расстояние, манхэттенское расстояние, косинусное расстояние и другие. Выбор расстояния зависит от конкретной задачи и признакового пространства.

Количество соседей (k) выбирается заранее и может влиять на качество классификации или регрессии. Если k слишком маленькое, то модель может быть шумной и чувствительной к выбросам в данных. Если k слишком большое, то модель может упустить некоторые детали и несоответствия в данных.

Преимущества k-NN заключаются в том, что он не требует обучения модели и может быть использован для классификации и регрессии на небольших наборах данных. Кроме того, он может быть эффективным при обработке несбалансированных классов.

Недостатки k-NN включают высокую вычислительную сложность при большом числе объектов и/или большой размерности пространства признаков. Кроме того, он не может работать с пропущенными значениями и может быть чувствителен к шуму в данных.

**1.8. Простая линейная регрессия**

Простая линейная регрессия - это метод обучения с учителем, который используется для предсказания значения одной непрерывной целевой переменной на основе линейной зависимости от одного признака.

Модель простой линейной регрессии имеет вид:

y = b0 + b1\*x

где y - целевая переменная, x - признак, b0 и b1 - коэффициенты модели, которые настраиваются в процессе обучения. Коэффициент b0 называется свободным членом или сдвигом (intercept), а коэффициент b1 - наклоном (slope) или весом (weight) признака.

Цель обучения простой линейной регрессии заключается в том, чтобы настроить коэффициенты модели таким образом, чтобы минимизировать сумму квадратов отклонений предсказанных значений от фактических значений целевой переменной. Эта мера называется среднеквадратичной ошибкой (mean squared error, MSE).

Метод наименьших квадратов (ordinary least squares, OLS) является одним из наиболее распространенных методов для нахождения коэффициентов модели простой линейной регрессии. Он минимизирует сумму квадратов отклонений между фактическими значениями целевой переменной и предсказанными значениями, находя наилучшие коэффициенты модели, которые дают наименьшее значение MSE.

Преимущества простой линейной регрессии заключаются в том, что она проста в понимании и реализации, а также может быть использована для исследования линейной связи между двумя переменными. Кроме того, её результаты можно легко интерпретировать.

Недостатки простой линейной регрессии заключаются в том, что она может быть ограничена в своей способности моделировать сложные зависимости между признаками и целевой переменной. Также она может быть чувствительна к выбросам в данных и неспособна работать с нелинейными зависимостями.

**1.9. Логистическая регрессия.**

Логистическая регрессия - это метод обучения с учителем, который используется для решения задач классификации, т.е. предсказания категории (класса) для объекта на основе значений его признаков.

Модель логистической регрессии имеет вид:

p = 1 / (1 + e^-(b0 + b1*x1 + b2*x2 + ... + bk\*xk))

где p - вероятность того, что объект относится к классу 1, x1, x2, ..., xk - признаки объекта, b0, b1, b2, ..., bk - коэффициенты модели, которые настраиваются в процессе обучения.

Логистическая регрессия использует сигмоидальную функцию для преобразования линейной комбинации признаков в вероятность принадлежности объекта к классу 1. Эта функция принимает значения от 0 до 1, что позволяет интерпретировать выход модели как вероятность принадлежности к классу 1.

Цель обучения логистической регрессии заключается в том, чтобы настроить коэффициенты модели таким образом, чтобы минимизировать ошибку классификации. Эта ошибка может быть измерена, например, с помощью функции логистической потери (log loss).

Преимущества логистической регрессии заключаются в ее простоте в понимании и реализации, быстром времени обучения, возможности интерпретации результатов. Кроме того, логистическая регрессия может работать с разными типами признаков (непрерывными, категориальными и т.д.).

Недостатки логистической регрессии заключаются в том, что она может быть ограничена в способности моделировать сложные зависимости между признаками и целевой переменной. Кроме того, она может быть чувствительна к наличию выбросов и коррелированных признаков, которые могут снижать ее производительность.

**1.10. Деревья принятия решений.**

Дерево принятия решений (Decision Tree) - это метод машинного обучения, который используется для решения задач классификации и регрессии.

Дерево принятия решений строится в виде дерева, где каждый узел представляет собой тест на значение одного из признаков. Каждый листовой узел дерева соответствует определенному классу (в задачах классификации) или значению целевой переменной (в задачах регрессии).

Процесс построения дерева принятия решений заключается в выборе наилучшего признака для разделения на каждом шаге. Наилучшим признаком считается тот, который обеспечивает максимальное уменьшение неопределенности (энтропии) выборки. Дерево строится рекурсивно, пока не будет достигнут критерий останова (например, максимальная глубина дерева или минимальное количество объектов в листовых узлах).

Преимущества деревьев принятия решений заключаются в их простоте в понимании и интерпретации, возможности работать с различными типами признаков (непрерывными, категориальными и т.д.), а также способности к автоматическому отбору признаков.

Недостатки деревьев принятия решений заключаются в их склонности к переобучению (особенно при большом количестве признаков), возможности получения неоптимальной структуры дерева, а также ограничениях в способности моделировать сложные зависимости между признаками и целевой переменной.

Существует несколько методов для решения проблемы переобучения, например, обрезка дерева, применение ансамблей деревьев (например, случайный лес или градиентный бустинг), использование регуляризации и т.д.

**1.11. Нейронные сети прямого распростанения**

Нейронные сети прямого распространения (feedforward neural networks) - это класс моделей машинного обучения, которые используются для решения задач классификации, регрессии, аппроксимации функций и других задач машинного обучения.

Основная идея нейронных сетей прямого распространения заключается в создании архитектуры сети, состоящей из нескольких слоев нейронов, которые передают информацию друг другу от входа к выходу. Каждый нейрон принимает на вход значения признаков (входной слой), взвешивает их с помощью своих весов и передает полученное значение дальше по сети. Выходной слой нейронной сети представляет собой ответ на задачу машинного обучения.

Преимущества нейронных сетей прямого распространения включают в себя способность моделировать сложные нелинейные зависимости между признаками и целевой переменной, а также возможность использования большого количества признаков.

Недостатки нейронных сетей прямого распространения включают в себя сложность в обучении и настройке гиперпараметров, особенно для больших и сложных архитектур сетей. Также существует проблема переобучения, когда модель выучивает шумы и нерелевантные признаки в обучающей выборке, что приводит к плохой обобщающей способности на новых данных.

Для решения проблемы переобучения нейронных сетей применяются различные методы регуляризации, такие как dropout и L1/L2 регуляризация. Также для настройки гиперпараметров нейронных сетей применяются различные оптимизационные методы, такие как стохастический градиентный спуск и его модификации, а также алгоритмы оптимизации на основе вторых производных, такие как методы оптимизации семейства BFGS

**1.12. Нейронные сети обратного распространения.**

Нейронные сети обратного распространения (backpropagation neural networks) являются наиболее распространенным типом нейронных сетей прямого распространения и используются для решения многих задач машинного обучения, таких как классификация, регрессия, обработка изображений и естественного языка.

Основной принцип работы нейронной сети обратного распространения заключается в передаче информации от выходного слоя сети к входному с помощью алгоритма обратного распространения ошибки. Этот алгоритм позволяет настроить веса и смещения (biases) нейронов в сети, минимизируя ошибку предсказания на обучающей выборке.

Алгоритм обратного распространения ошибки основан на методе градиентного спуска, который позволяет находить оптимальные значения весов и смещений путем минимизации ошибки предсказания на обучающей выборке. Для этого производится обратный проход по сети, где каждый нейрон вычисляет свою ошибку и передает ее в обратном направлении. На каждом шаге вычисляется градиент ошибки по весам и смещениям, и они обновляются с помощью правила обновления весов, определяемого определенной функцией потерь.

Преимущества нейронных сетей обратного распространения включают в себя высокую гибкость и способность обучаться на больших объемах данных с высокой точностью. Также нейронные сети обратного распространения могут использоваться для обработки данных с высокой размерностью и структурированных данных, таких как изображения и текст.

Недостатки нейронных сетей обратного распространения включают в себя сложность в обучении и настройке гиперпараметров, а также возможность переобучения на малых выборках. Для решения этих проблем используются различные методы регуляризации и оптимизации, такие как dropout, L1/L2 регуляризация, а также стохастический градиентный спуск и его модификации.

**2.1.**1Задайте функцию, которая не принимает никаких аргументов и просто выводит на экран строку: It's my first function

В конце программы вызовите эту функцию.

# Определение функции

def my\_first\_function():

print("It's my first function")

# Вызов функции

my\_first\_function()

В этом примере мы создали функцию **my\_first\_function**, которая не принимает никаких аргументов и просто выводит на экран строку "It's my first function". Затем мы вызвали эту функцию с помощью оператора **my\_first\_function()**, что привело к выводу указанной строки на экран.

**2.1.**2Имеется список с оценками студента:

m = [2, 3, 5, 5, 2, 2, 3, 3, 4, 5, 4, 4]

Необходимо с помощью срезов выбрать элементы с 3-го по 7-й (включительно) и вывести их на экран в обратном порядке.

# Определение списка

m = [2, 3, 5, 5, 2, 2, 3, 3, 4, 5, 4, 4]

# Использование срезов для выбора элементов с 3-го по 7-й включительно

selected\_elements = m[2:7]

# Вывод выбранных элементов в обратном порядке

print(selected\_elements[::-1])

В этом примере мы создали список **m**, содержащий оценки студента, и использовали срезы для выбора элементов с 3-го по 7-й включительно. Затем мы использовали обратный срез (**[::-1]**) для вывода выбранных элементов в обратном порядке.

**2.2.**1Вводятся целые числа в одну строчку через пробел. Необходимо преобразовать их в список lst , затем, удалить последнее значение и если оно нечетное, то в список (в конец) добавить True, иначе - False. Отобразить полученный список на экране командой: print(\*lst)

Реализовать программу без использования условного оператора.

# Ввод целых чисел через пробел и преобразование их в список

lst = list(map(int, input().split()))

# Удаление последнего значения из списка

lst.pop()

# Добавление True, если последнее значение нечетное, и False, если оно четное

lst.append((lst[-1] % 2) != 0)

# Вывод полученного списка на экран

print(\*lst)

В этом примере мы сначала вводим строку с целыми числами через пробел, затем преобразовываем ее в список **lst**. Затем мы удаляем последний элемент из списка с помощью метода **pop()**. Далее мы используем арифметическую операцию **%** для проверки на четность последнего элемента списка. Если остаток от деления на 2 не равен нулю, мы добавляем в конец списка значение **True**, в противном случае - **False**. Наконец, мы выводим полученный список на экран с помощью функции **print()** и оператора распаковки **\***.

**2.2.**2На вход программы поступают данные в виде набора строк в формате:

ключ1=значение1  
ключ2=значение2  
...  
ключN=значениеN

Ключами здесь выступают целые числа (см. пример ниже). Необходимо их преобразовать в словарь d (без использования функции dict()) и вывести его на экран командой:

print(\*sorted(d.items()))

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

# Начальные строки, содержащие данные в формате "ключ=значение"

input\_str = '''1=значение1

2=значение2

3=значение3'''

# Разбиение начальных строк на список строк с помощью метода split()

input\_lines = input\_str.split('\n')

# Создание пустого словаря

d = {}

# Цикл по строкам списка input\_lines

for line in input\_lines:

# Разбиение строки на ключ и значение с помощью метода split()

key, value = line.split('=')

# Преобразование ключа в целое число

key = int(key)

# Добавление ключа и значения в словарь

d[key] = value

# Вывод отсортированного словаря на экран

print(\*sorted(d.items()))

В этом примере мы сначала задаем начальные строки, содержащие данные в формате "ключ=значение", и разбиваем их на список строк **input\_lines**. Затем мы создаем пустой словарь **d** и используем цикл **for** для обработки каждой строки в списке **input\_lines**. Внутри цикла мы разбиваем строку на ключ и значение с помощью метода **split()**, преобразуем ключ в целое число, а затем добавляем ключ и значение в словарь **d**. Наконец, мы выводим отсортированный словарь на экран с помощью функции **print()** и метода **items()**.

**2.3.**1Вводятся оценки студента (числа от 2 до 5) в одну строку через пробел. Необходимо определить количество двоек и вывести это значение на экран.

# Ввод оценок студента в одну строку

input\_str = input("Введите оценки студента через пробел: ")

# Разбивка строки на список оценок с помощью метода split()

grades = input\_str.split()

# Подсчет количества двоек в списке оценок

count\_of\_twos = grades.count('2')

# Вывод количества двоек на экран

print(f"Количество двоек: {count\_of\_twos}")

В этом примере мы используем функцию **input()** для получения строки с оценками студента от пользователя. Затем мы используем метод **split()** для разбивки строки на список оценок. Мы затем используем метод **count()** для подсчета количества двоек в списке, и выводим это значение на экран с помощью функции **print()**.

**2.3.**2Вводятся данные в формате ключ=значение в одну строчку через пробел. Необходимо на их основе создать словарь, затем проверить, существуют ли в нем ключи со значениями: 'house', 'True' и '5' (все ключи - строки). Если все они существуют, то вывести на экран ДА, иначе - НЕТ.

# Ввод данных в формате ключ=значение в одну строку

input\_str = input("Введите данные в формате ключ=значение через пробел: ")

# Разбивка строки на пары ключ-значение с помощью метода split()

input\_list = input\_str.split()

# Создание словаря на основе пар ключ-значение

input\_dict = {}

for item in input\_list:

key, value = item.split('=')

input\_dict[key] = value

# Проверка наличия ключей со значениями 'house', 'True' и '5' в словаре

if all(key in input\_dict and input\_dict[key] in ['house', 'True', '5'] for key in ['key1', 'key2', 'key3']):

print("ДА")

else:

print("НЕТ")

В этом примере мы используем функцию **input()** для получения строки в формате "ключ=значение". Затем мы используем метод **split()** для разбивки строки на список пар ключ-значение. Мы затем используем цикл **for** для создания словаря на основе этих пар ключ-значение. Мы затем проверяем наличие ключей со значениями 'house', 'True' и '5' в словаре с помощью выражения **all(key in input\_dict and input\_dict[key] in ['house', 'True', '5'] for key in ['key1', 'key2', 'key3'])**, и выводим соответствующее сообщение на экран с помощью функции **print()**.

**2.4.**1Вводятся данные в формате ключ=значение в одну строчку через пробел. Необходимо на их основе создать словарь d, затем удалить из этого словаря ключи 'False' и '3', если они существуют. Ключами и значениями словаря являются строки. Вывести полученный словарь на экран командой: print(\*sorted(d.items()))

# считываем данные и создаем словарь

data = input().split()

d = {}

for item in data:

key, value = item.split('=')

d[key] = value

# удаляем ключи 'False' и '3', если они существуют

if 'False' in d:

del d['False']

if '3' in d:

del d['3']

# выводим полученный словарь

print(\*sorted(d.items()))

В этой программе мы сначала считываем данные в формате "ключ=значение" в строку **data**, используя метод **split()** для разделения строки на отдельные элементы. Затем мы создаем пустой словарь **d** и в цикле проходимся по каждому элементу списка **data**, разделяя каждый элемент на ключ и значение с помощью метода **split()**. Затем мы добавляем каждый ключ и его значение в словарь **d**.

Затем мы проверяем, есть ли в словаре ключи "False" и "3". Если они есть, то мы используем оператор **del** для удаления соответствующих элементов из словаря.

Наконец, мы выводим на экран содержимое словаря **d** с помощью функции **print()**, используя метод **items()** для получения списка кортежей с ключами и их значениями, а затем сортируем его с помощью функции **sorted()**.

**2.4.**2Вводятся номера телефонов в одну строчку через пробел с разными кодами стран: +7, +6, +2, +4 и т.д. Необходимо составить словарь d, где ключи - это коды +7, +6, +2 и т.п., а значения - список номеров (следующих в том же порядке, что и во входной строке) с соответствующими кодами. Полученный словарь вывести командой: print(\*sorted(d.items()))

phone\_numbers = input("Введите номера телефонов через пробел: ").split()

d = {}

for number in phone\_numbers:

code = number[:2]

if code in d:

d[code].append(number)

else:

d[code] = [number]

print(\*sorted(d.items()))

В этом коде мы считываем строку с номерами телефонов от пользователя, разбиваем ее на список с помощью метода **split()**. Затем мы создаем пустой словарь **d**. Затем мы проходим по каждому номеру в списке **phone\_numbers** и извлекаем код страны, используя срез **number[:2]**. Затем мы проверяем, есть ли код страны уже в словаре **d**. Если да, то мы добавляем текущий номер в список значений по соответствующему ключу. Если код страны еще не добавлен в словарь, мы создаем новый ключ и присваиваем ему список значений, начиная с текущего номера. Наконец, мы выводим полученный словарь, отсортированный по ключу, с помощью функции **print(\*sorted(d.items()))**.

**2.5.**1 Пользователь вводит в цикле целые положительные числа, пока не введет число 0. Для каждого числа вычисляется квадратный корень (с точностью до сотых) и значение выводится на экран (в столбик). С помощью словаря выполните кэширование данных так, чтобы при повторном вводе того же самого числа результат не вычислялся, а бралось ранее вычисленное значение из словаря. При этом на экране должно выводиться: значение из кэша: <число>

import math

cache = {}

while True:

number = int(input("Введите целое положительное число (для выхода введите 0): "))

if number == 0:

break

if number in cache:

print("Значение из кэша:", cache[number])

else:

square\_root = round(math.sqrt(number), 2)

cache[number] = square\_root

print(square\_root)

В этом коде мы используем бесконечный цикл **while True**, чтобы позволить пользователю вводить числа до тех пор, пока не будет введено число 0. Внутри цикла мы считываем введенное число и проверяем, есть ли оно уже в словаре **cache**. Если число уже есть в кэше, мы выводим сообщение "Значение из кэша:" и значение из словаря. Если числа нет в кэше, мы вычисляем его квадратный корень с помощью функции **math.sqrt()** и округляем результат до двух знаков после запятой с помощью функции **round()**. Затем мы добавляем значение в словарь **cache** с ключом **number** и выводим вычисленное значение на экран.

**2.5.**2 Имеется закодированная строка с помощью азбуки Морзе. Коды разделены между собой пробелом. Необходимо ее раскодировать, используя азбуку Морзе из предыдущего занятия. Полученное сообщение (строку) вывести на экран.

morse\_code = {

'A': '.-', 'B': '-...', 'C': '-.-.', 'D': '-..', 'E': '.', 'F': '..-.', 'G': '--.',

'H': '....', 'I': '..', 'J': '.---', 'K': '-.-', 'L': '.-..', 'M': '--', 'N': '-.',

'O': '---', 'P': '.--.', 'Q': '--.-', 'R': '.-.', 'S': '...', 'T': '-', 'U': '..-',

'V': '...-', 'W': '.--', 'X': '-..-', 'Y': '-.--', 'Z': '--..',

'0': '-----', '1': '.----', '2': '..---', '3': '...--', '4': '....-', '5': '.....',

'6': '-....', '7': '--...', '8': '---..', '9': '----.',

'.': '.-.-.-', ',': '--..--', '?': '..--..', "'": '.----.', '!': '-.-.--',

'/': '-..-.', '(': '-.--.', ')': '-.--.-', '&': '.-...',

':': '---...', ';': '-.-.-.', '=': '-...-', '+': '.-.-.',

'-': '-....-', '\_': '..--.-', '"': '.-..-.', '$': '...-..-',

'@': '.--.-.', ' ': '/'

}

def decode\_morse\_code(encoded\_string):

decoded\_string = ''

encoded\_chars = encoded\_string.split(' ')

for char in encoded\_chars:

for key, value in morse\_code.items():

if value == char:

decoded\_string += key

break

return decoded\_string

encoded\_string = input("Введите закодированную строку: ")

decoded\_string = decode\_morse\_code(encoded\_string)

print("Раскодированная строка:", decoded\_string)

В этом коде мы используем словарь **morse\_code**, который содержит соответствия между символами алфавита и цифрами и их кодами Морзе. Затем мы определяем функцию **decode\_morse\_code**, которая принимает закодированную строку в качестве аргумента и возвращает раскодированную строку. Внутри функции мы разделяем закодированную строку на символы с помощью split(' '). Затем мы проходим по каждому символу и ищем соответствующий символ или цифру в словаре morse\_code. Когда мы находим соответствие, мы добавляем символ в раскодированную строку decoded\_string. В конце мы выводим полученную раскодированную строку на экран.

**2.6.1**.В список: a = [5.4, 6.7, 10.4]

добавить в конец вложенный список со значениями, вводимыми в программу (целые числа вводятся в строчку через пробел). Результирующий список lst вывести на экран командой: print(lst)

a = [5.4, 6.7, 10.4]

values = input("Введите целые числа через пробел: ").split()

nested\_list = [int(value) for value in values]

a.append(nested\_list)

print(a)

В этом коде мы сначала создаем список **a** с исходными значениями **[5.4, 6.7, 10.4]**. Затем мы считываем строку ввода от пользователя, содержащую целые числа, разделенные пробелом. С помощью метода **split()** мы разделяем строку на отдельные значения. Затем мы создаем новый вложенный список **nested\_list**, используя генератор списка, чтобы преобразовать каждое значение в целое число с помощью функции **int()**. Наконец, мы добавляем вложенный список в конец списка **a** с помощью метода **append()** и выводим полученный список **a** на экран.

**2.6.**2Имеется кортеж: t = (3.4, -56.7)

Вводится последовательность целых чисел в одну строчку через пробел. Необходимо их добавить в кортеж t. Результат вывести на экран командой: print(t)

t = (3.4, -56.7)

numbers = input("Введите целые числа через пробел: ").split()

numbers = tuple(map(int, numbers))

t += numbers

print(t)

В этом коде мы сначала создаем кортеж **t** с исходными значениями **(3.4, -56.7)**. Затем мы считываем строку ввода от пользователя, содержащую целые числа, разделенные пробелом. С помощью метода **split()** мы разделяем строку на отдельные значения. Затем мы используем функцию **map()** и **int()** для преобразования каждого значения в целое число, а затем создаем новый кортеж **numbers** с помощью **tuple()**. После этого мы объединяем кортежи **t** и **numbers** с помощью оператора **+=**, создавая новый кортеж, содержащий все элементы из обоих кортежей. Наконец, мы выводим полученный кортеж **t** на экран.

**2.7.**1Вводятся целые числа в одну строку через пробел. На их основе формируется кортеж. Необходимо найти и вывести все индексы неуникальных (повторяющихся) значений в этом кортеже. Результат отобразите в виде строки чисел, записанных через пробел.

numbers = input("Введите целые числа через пробел: ").split()

numbers = tuple(map(int, numbers))

indices = [i for i in range(len(numbers)) if numbers.count(numbers[i]) > 1]

indices\_string = ' '.join(map(str, indices))

print(indices\_string)

В этом коде мы сначала считываем строку ввода от пользователя, содержащую целые числа, разделенные пробелом. Затем мы с помощью метода **split()** разделяем строку на отдельные значения. С помощью функции **map()** и **int()** мы преобразуем каждое значение в целое число, а затем создаем кортеж **numbers** с помощью **tuple()**.

Затем мы создаем список **indices**, используя генератор списка. Мы проходим по индексам элементов кортежа **numbers** с помощью функции **range(len(numbers))**, и проверяем, сколько раз элемент встречается в кортеже с помощью метода **count()**. Если элемент встречается более одного раза, мы добавляем его индекс в список **indices**.

Затем мы преобразуем список **indices** в строку, используя метод **join()**, где каждый элемент преобразуется в строку с помощью функции **str()**. И наконец, мы выводим полученную строку **indices\_string** на экран.

**2.7.**2Вводятся три строчки стихотворения (каждая с новой строки). Сохранить его в виде вложенного списка с разбивкой по строкам и словам (слова разделяются пробелом). Результирующий список lst вывести на экран командой: **print(lst)**

lines = [input("Введите строку стихотворения: ") for \_ in range(3)]

lst = [line.split() for line in lines]

print(lst)

В этом коде мы используем цикл **for** и функцию **input()** для считывания трех строк стихотворения от пользователя. Каждая строка сохраняется в списке **lines**.

Затем мы создаем новый список **lst**, который является вложенным списком. Мы используем генератор списка, проходя по каждой строке в списке **lines** и разделяя ее на слова с помощью метода **split()**. Каждое слово становится отдельным элементом вложенного списка.

Наконец, мы выводим полученный список **lst** на экран с помощью команды **print(lst)**.

**2.8.1**Вводится  матрица чисел из трех строк. В каждой строке числа разделяются пробелом. Необходимо вывести на экран последний столбец этой матрицы в виде строки из трех чисел через пробел.

matrix = [input("Введите строку матрицы: ").split() for \_ in range(3)]

last\_column = [row[-1] for row in matrix]

last\_column\_string = ' '.join(last\_column)

print(last\_column\_string)

В этом коде мы используем цикл **for** и функцию **input()** для считывания трех строк матрицы от пользователя. Каждая строка разделяется на отдельные числа с помощью метода **split()**, и каждая строка сохраняется в виде списка.

Затем мы создаем новый список **last\_column**, используя генератор списка. Мы проходим по каждой строке в матрице **matrix** и выбираем последний элемент каждой строки с помощью индекса **[-1]**. Таким образом, мы формируем список, содержащий последний столбец матрицы.

Далее, мы преобразуем список **last\_column** в строку с помощью метода **join()**, где каждый элемент преобразуется в строку. В итоге получаем строку **last\_column\_string**, содержащую числа последнего столбца матрицы, разделенные пробелом.

Наконец, мы выводим полученную строку **last\_column\_string** на экран с помощью команды **print()**.

**2.8.2** Имеется двумерный кортеж, размером 5 x 5 элементов:

t = ((1, 0, 0, 0, 0),

 (0, 1, 0, 0, 0),

 (0, 0, 1, 0, 0),

 (0, 0, 0, 1, 0),

 (0, 0, 0, 0, 1))

Вводится натуральное число N (N < 5). Необходимо на основе кортежа t сформировать новый аналогичный кортеж t2 размером N x N элементов. Результат вывести на экран в виде таблицы чисел.

t = ((1, 0, 0, 0, 0),

(0, 1, 0, 0, 0),

(0, 0, 1, 0, 0),

(0, 0, 0, 1, 0),

(0, 0, 0, 0, 1))

N = int(input("Введите значение N: "))

t2 = tuple(row[:N] for row in t[:N])

for row in t2:

print(\*row)

В этом коде мы сначала определяем исходный двумерный кортеж **t**, содержащий значения размером 5 x 5.

Затем мы считываем натуральное число **N** от пользователя.

Далее мы создаем новый кортеж **t2**, используя генератор кортежа. Мы проходим по первым **N** строкам и первым **N** элементам каждой строки из исходного кортежа **t**, и формируем соответствующую подматрицу размером **N x N**.

Затем мы выводим полученную матрицу **t2** на экран в виде таблицы чисел, используя цикл **for**. Для каждой строки мы используем команду **print(\*row)**, чтобы вывести элементы строки раздельно.

Таким образом, мы получаем на экране таблицу чисел, соответствующую новому кортежу **t2** размером **N x N**.

**2.9.1** Вводятся два вещественных числа в одну строку через пробел. Вывести на экран наибольшее из чисел. Задачу решить с помощью условного оператора.

# Ввод чисел

a, b = map(float, input("Введите два вещественных числа через пробел: ").split())

# Проверка и вывод наибольшего числа

if a > b:

print(a)

else:

print(b)

В этом коде мы используем функцию **input()** для считывания строки с двумя вещественными числами, разделенными пробелом. Затем мы используем метод **split()** для разделения строки на отдельные числа, и функцию **map(float, ...)** для преобразования каждого числа в формат **float** (вещественное число).

Далее мы используем условный оператор **if-else** для сравнения чисел **a** и **b**. Если **a** больше **b**, то мы выводим **a**, иначе мы выводим **b**.

Таким образом, на экране будет выведено наибольшее из двух введенных вещественных чисел.

**2.9.2** Вводится текст в одну строку, слова разделены пробелом. Необходимо подсчитать число уникальных слов (без учета регистра) в этом тексте. Результат (число уникальных слов) вывести на экран.

# Ввод текста

text = input("Введите текст: ")

# Разделение текста на слова

words = text.lower().split()

# Подсчет уникальных слов

unique\_words = set(words)

count = len(unique\_words)

# Вывод результата

print("Число уникальных слов:", count)

В этом коде мы сначала считываем текст с помощью функции input() и сохраняем его в переменной text.

Затем мы приводим текст к нижнему регистру с помощью метода lower(). После этого мы используем метод split() для разделения текста на отдельные слова.

Далее мы создаем множество unique\_words, которое содержит только уникальные слова из списка words.

Затем мы используем функцию len() для подсчета числа элементов в множестве unique\_words и сохраняем результат в переменной count.

Наконец, мы выводим на экран число уникальных слов, используя команду print().

Таким образом, на экране будет выведено число уникальных слов в введенном тексте, не учитывая регистр слов.

**2.10.**1Вводится слово. Необходимо определить, является ли это слово палиндромом (одинаково читается вперед и назад, например, АННА). Регистр букв не учитывать. Если введенное слово палиндром, на экран вывести ДА, иначе - НЕТ.

# Ввод слова

word = input("Введите слово: ")

# Приведение слова к нижнему регистру

word = word.lower()

# Проверка на палиндром

if word == word[::-1]:

print("ДА")

else:

print("НЕТ")

В этом коде мы сначала считываем слово с помощью функции **input()** и сохраняем его в переменной **word**.

Затем мы приводим слово к нижнему регистру с помощью метода **lower()**.

Далее мы используем условный оператор **if-else** для проверки, является ли слово палиндромом. Мы сравниваем исходное слово **word** с его перевернутой версией **word[::-1]**. Если они равны, то слово является палиндромом, и мы выводим "ДА", иначе мы выводим "НЕТ".

Таким образом, на экран будет выведено "ДА", если введенное слово является палиндромом, и "НЕТ", если оно не является палиндромом. Регистр букв не учитывается при определении палиндрома.

**2.10.**2 Вводятся два списка целых чисел каждый с новой строки (в строке наборы чисел через пробел). Необходимо выбрать и отобразить на экране уникальные числа, присутствующие в первом списке, но отсутствующие во втором. Результат выведите на экран в виде строки чисел, записанных по возрастанию через пробел.

# Ввод двух списков целых чисел

list1 = input("Введите первый список чисел: ").split()

list2 = input("Введите второй список чисел: ").split()

# Преобразование элементов списков в целые числа

list1 = list(map(int, list1))

list2 = list(map(int, list2))

# Выбор уникальных чисел из первого списка, отсутствующих во втором списке

unique\_numbers = [num for num in list1 if num not in list2]

# Сортировка уникальных чисел по возрастанию

unique\_numbers.sort()

# Вывод результатов

print(\*unique\_numbers)

В этом коде мы сначала считываем два списка целых чисел с помощью функции **input()**. Каждый список сохраняется в отдельной переменной, **list1** и **list2**.

Затем мы используем метод **split()** для разделения строк на отдельные числа. Результаты сохраняются в виде списков строк.

Далее мы используем функцию **map()** и **int** для преобразования элементов списков из строк в целые числа.

Затем мы используем генератор списка для выбора уникальных чисел из первого списка (**list1**), которые отсутствуют во втором списке (**list2**). Мы проверяем каждое число из **list1** и используем условие **num not in list2**, чтобы проверить, есть ли оно в **list2**.

Далее мы сортируем уникальные числа по возрастанию с помощью метода **sort()**.

Наконец, мы выводим результаты на экран с помощью команды **print(\*unique\_numbers)**, которая разделяет элементы списка пробелами и выводит их по одному.

**2.11.**1Вводятся два целых положительных числа m и n в одну строку через пробел. Если число m делится нацело на число n, то вывести на экран частное от деления (результат деления) в виде целого числа. В противном случае вывести сообщение «m на n нацело не делится» (без кавычек) и вместо m и n подставить соответствующие числа, например: «13 на 2 нацело не делится».

# Ввод двух целых положительных чисел

m, n = map(int, input("Введите два числа m и n через пробел: ").split())

# Проверка условия деления m нацело на n

if m % n == 0:

# Вывод частного от деления

quotient = m // n

print(quotient)

else:

# Вывод сообщения о невозможности деления нацело

print(f"{m} на {n} нацело не делится")

В этом коде мы сначала считываем два целых положительных числа, **m** и **n**, с помощью функции **input()**. Мы используем метод **split()** для разделения строки на отдельные числа, а затем функцию **map(int, ...)** для преобразования строковых значений в целые числа.

Затем мы проверяем условие деления **m** нацело на **n** с помощью оператора **%**. Если остаток от деления равен 0, значит число **m** делится нацело на число **n**.

В случае выполнения условия, мы выводим частное от деления, которое является результатом операции **m // n**. Здесь оператор **//** выполняет целочисленное деление.

Если условие не выполняется, мы выводим сообщение о невозможности деления нацело с использованием **print(f"...")**. Мы используем f-строку для форматирования строки, подставляя значения **m** и **n** в нужные места.

**2.11.**2Вводятся два списка целых чисел каждый с новой строки (в строке наборы чисел через пробел). Необходимо выбрать и отобразить на экране уникальные числа, присутствующие в первом или втором списках, но отсутствующие одновременно в обоих. Результат выведите на экран в виде строки чисел, записанных по возрастанию через пробел.

# Ввод двух списков целых чисел

list1 = list(map(int, input("Введите числа первого списка через пробел: ").split()))

list2 = list(map(int, input("Введите числа второго списка через пробел: ").split()))

# Получение уникальных чисел

unique\_numbers = list(set(list1) ^ set(list2))

# Сортировка чисел по возрастанию

unique\_numbers.sort()

# Вывод уникальных чисел

print(\*unique\_numbers)

В этом коде мы сначала считываем два списка целых чисел, **list1** и **list2**, с помощью функции **input()**. Мы используем метод **split()** для разделения строки на отдельные числа, а затем функцию **map(int, ...)** для преобразования строковых значений в целые числа.

Затем мы используем оператор **^** для получения симметрической разности множеств **set(list1)** и **set(list2)**. Это позволяет нам получить уникальные числа, которые присутствуют либо в первом списке, либо во втором, но не одновременно в обоих.

Мы преобразуем полученное множество обратно в список с помощью **list()** и сохраняем его в переменной **unique\_numbers**.

Затем мы сортируем числа в списке **unique\_numbers** по возрастанию с помощью метода **sort()**.

Наконец, мы выводим уникальные числа на экран, используя **print(\*unique\_numbers)**, чтобы каждое число было отдельным аргументом для функции **print()**, а не выводилось в виде списка.

**2.12.**1Вводится список оценок студента - его ответов у доски по предмету "Информатика" в виде чисел от 2 до 5 в одну строку через пробел. Если студент имеет хотя бы одну двойку, то он не допускается до экзамена. Определить на основе введенного списка, допущен ли студент. Если допущен, то вывести слово ДОПУЩЕН, иначе - НЕ ДОПУЩЕН. При реализации задачи используйте множество для определения наличия двойки.

# Ввод списка оценок студента

grades = list(map(int, input("Введите оценки студента через пробел: ").split()))

# Проверка на наличие двойки

if 2 in set(grades):

print("НЕ ДОПУЩЕН")

else:

print("ДОПУЩЕН")

В этом коде мы сначала считываем список оценок студента с помощью функции **input()**. Мы используем метод **split()** для разделения строки на отдельные оценки, а затем функцию **map(int, ...)** для преобразования строковых значений в целые числа.

Затем мы создаем множество **set(grades)**, чтобы быстро проверить наличие двойки в списке оценок с помощью оператора **in**. Если двойка присутствует во множестве, то выводим "НЕ ДОПУЩЕН". В противном случае, если двойка отсутствует, выводим "ДОПУЩЕН".

**2.12.**2Вводится натуральное число, которое может содержать только простые множители 1, 2, 3, 5 и 7 (любые из них, не обязательно все). Необходимо разложить введенное число на простые множители и проверить, содержит ли оно множители 2, 3 и 5 (обязательно все их, хотя бы один раз). Если это так, то вывести ДА, иначе - НЕТ.

# Ввод числа

number = int(input("Введите натуральное число: "))

# Проверка на наличие простых множителей 2, 3 и 5

has\_2 = False

has\_3 = False

has\_5 = False

# Разложение числа на простые множители

while number % 2 == 0:

number //= 2

has\_2 = True

while number % 3 == 0:

number //= 3

has\_3 = True

while number % 5 == 0:

number //= 5

has\_5 = True

# Проверка наличия всех множителей 2, 3 и 5

if has\_2 and has\_3 and has\_5:

print("ДА")

else:

print("НЕТ")

В этом коде мы сначала считываем натуральное число с помощью функции **input()** и преобразуем его в целое число с помощью **int()**.

Затем мы проверяем, содержит ли число простые множители 2, 3 и 5. Для этого мы итеративно делим число на каждый из множителей, начиная с 2, 3 и 5. Если число делится без остатка на множитель, то устанавливаем соответствующий флаг (**has\_2**, **has\_3**, **has\_5**) в значение **True**.

После разложения числа на простые множители, мы проверяем наличие всех трех множителей. Если каждый из флагов **has\_2**, **has\_3**, **has\_5** равен **True**, то выводим "ДА". В противном случае, если хотя бы один из флагов равен **False**, выводим "НЕТ".