

Федеральное государственное образовательное бюджетное учреждение
высшего образования
«Финансовый университет при Правительстве Российской Федерации»
(Пермский филиал)

ОДОБРЕНО

на заседании НМС Протокол №1
от «24» октября 2024 г.
Председатель, к.п.н.



Е.А. Шистерова

РАССМОТРЕНО

на заседании ПЦК ИСП Протокол
№ 11 от «27» июня 2024 г.
Председатель ПЦК



И.В. Галкин

Методические рекомендации
по выполнению практических работ по учебной дисциплине
«Информатика»

38.02.02 Страхование

Методические рекомендации предназначены для студентов специальностей 38.02.01 Экономика и бухгалтерский учёт, 38.02.02 Страхование дело (по отраслям), 38.02.06 Финансы, 38.02.07 Банковское дело, 40.02.04 Юриспруденция (юрист в сфере правоохранительной деятельности), 09.02.07 Информационные системы и программирование при выполнении практических работ с применением дистанционных образовательных технологий по дисциплине «Информатика». Предлагаемые методические рекомендации помогут студентам освоить работы из модуля 1. «Аналитика и визуализация данных на Python».

Методические рекомендации адресованы преподавателям образовательных учреждений, реализующих образовательные программы среднего профессионального образования в соответствии с Федеральными государственными образовательными стандартами.

Разработчик:

Галкин И.В., преподаватель Пермского филиала ФГОБУ ВО «Финансовый университет при Правительстве Российской Федерации»

Содержание

Введение	4
Распределение тем учебного материала дисциплины «Информатика»	9
Аннотация материала.....	10
Контроль и оценивание.....	10
Содержание дистанционного материала	11
Практика 1. Списки. Методы split и join. Файловый ввод/вывод	11
Практика 2. Списки. Методы split и join. Файловый ввод/вывод	15
Практика 3. Списки. Методы split и join. Файловый ввод/вывод	21
Практика 4. Рекурсия. Множества. Словари	28
Практика 5. Рекурсия. Множества. Словари	32
Практика 6. Рекурсия. Множества. Словари	35
Практика 7. Рекурсия. Множества. Словари	40
Список использованных источников	45

Введение

Требования работодателей к современному специалисту, отраженные в профессиональных стандартах, а также требования ФГОС СОО, ФГОС СПО ориентированы, прежде всего, на умения самостоятельной работы и творческий подход к профессиональной деятельности. В современных условиях способности студента грамотно организовать самостоятельную практическую работу становятся чрезвычайно актуальными. Методические рекомендации по выполнению практических работ студентов по дисциплине «Информатика» предназначены для преподавателей образовательных учреждений, реализующих образовательные программы среднего профессионального образования в соответствии с Федеральными государственными образовательными стандартами специальностей 38.02.01 Экономика и бухгалтерский учёт, 38.02.02 Страхование дело (по отраслям), 38.02.06 Финансы, 38.02.07 Банковское дело, 40.02.04 Юриспруденция (юрист в сфере правоохранительной деятельности), 09.02.07 Информационные системы и программирование.

В рекомендациях представлены материалы, способствующие формированию у обучающихся знаний, умений и навыков применения среды программирования Python в целях анализа и визуализации данных. Выполнение данных практических работ в соответствии с ФГОС СПО и ФГОС СОО позволит сформировать у обучающихся следующие результаты:

Код и наименование формируемых компетенций	Планируемые результаты освоения дисциплины	
	Общие	Дисциплинарные
ОК 01. Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам	В части трудового воспитания: - готовность к труду, осознание ценности мастерства, трудолюбие; - готовность к активной деятельности технологической и социальной направленности, способность инициировать, планировать и самостоятельно выполнять такую деятельность; - интерес к различным сферам профессиональной деятельности, Овладение универсальными учебными познавательными действиями: а) базовые логические действия: - самостоятельно формулировать и актуализировать проблему, рассматривать ее всесторонне; - устанавливать существенный признак или основания для сравнения, классификации и обобщения; - определять цели деятельности, задавать параметры и критерии их достижения; - выявлять закономерности и	- понимать угрозу информационной безопасности, использовать методы и средства противодействия этим угрозам, соблюдение мер безопасности, предотвращающих незаконное распространение персональных данных; соблюдение требований техники безопасности и гигиены при работе с компьютерами и другими компонентами цифрового окружения; понимание правовых основ использования компьютерных программ, баз данных и работы в сети Интернет; - уметь организовывать личное информационное пространство с использованием различных средств цифровых технологий; понимание возможностей цифровых сервисов государственных услуг, цифровых образовательных сервисов; понимание возможностей и ограничений технологий искусственного интеллекта в различных областях; наличие представлений об использовании информационных технологий в различных профессиональных сферах - уметь реализовать этапы решения

	противоречия в рассматриваемых явлениях; - вносить коррективы в деятельность, оценивать соответствие результатов целям, оценивать риски последствий деятельности; - развивать креативное мышление при решении жизненных проблем б) базовые исследовательские действия: - владеть навыками учебно-исследовательской и проектной деятельности, навыками разрешения проблем; - выявлять причинно-следственные связи и актуализировать задачу, выдвигать гипотезу ее решения, находить аргументы для доказательства своих утверждений, задавать параметры и критерии решения; - анализировать полученные в ходе решения задачи результаты, критически оценивать их достоверность, прогнозировать изменение в новых условиях; - уметь переносить знания в познавательную и практическую области жизнедеятельности; - уметь интегрировать знания из разных предметных областей; - выдвигать новые идеи, предлагать оригинальные подходы и решения; - способность их использования в познавательной и социальной практике	задач на компьютере; умение реализовывать на выбранном для изучения языке программирования высокого уровня (Паскаль, Python, Java, C++, C#) типовые алгоритмы обработки чисел, числовых последовательностей и массивов: представление числа в виде набора простых сомножителей; нахождение максимальной (минимальной) цифры натурального числа, записанного в системе счисления с основанием, не превышающим 10; вычисление обобщенных характеристик элементов массива или числовой последовательности (суммы, произведения среднего арифметического, минимального и максимального элементов, количества элементов, удовлетворяющих заданному условию); сортировку элементов массива;
ОК 02. Использовать современные средства поиска, анализа и интерпретации информации и информационные технологии для выполнения задач профессиональной	В области ценности научного познания: - сформированность мировоззрения, соответствующего современному уровню развития науки и общественной практики, основанного на диалоге культур, способствующего осознанию своего места в поликультурном	- владеть представлениями о роли информации и связанных с ней процессов в природе, технике и обществе; понятиями «информация», «информационный процесс», «система», «компоненты системы» «системный эффект», «информационная система», «система управления»; владеть методами поиска информации в сети Интернет; уметь критически оценивать

деятельности	мире; - совершенствование языковой и читательской культуры как средства взаимодействия между людьми и познания мира; - осознание ценности научной деятельности, готовность осуществлять проектную и исследовательскую деятельность индивидуально и в группе; Овладение универсальными учебными познавательными действиями: в) работа с информацией: - владеть навыками получения информации из источников разных типов, самостоятельно осуществлять поиск, анализ, систематизацию и интерпретацию информации различных видов и форм представления; - создавать тексты в различных форматах с учетом назначения информации и целевой аудитории, выбирая оптимальную форму представления и визуализации; - оценивать достоверность, легитимность информации, ее соответствие правовым и морально-этическим нормам; - использовать средства информационных и коммуникационных технологий в решении когнитивных, коммуникативных и организационных задач с соблюдением требований эргономики, техники безопасности, гигиены, ресурсосбережения, правовых и этических норм, норм информационной безопасности; - владеть навыками распознавания и защиты информации, информационной безопасности личности	информацию, полученную из сети Интернет; характеризовать большие данные, приводить примеры источников их получения и направления использования; - понимать основные принципы устройства и функционирования современных стационарных и мобильных компьютеров; тенденций развития компьютерных технологий; владеть навыками работы с операционными системами и основными видами программного обеспечения для решения учебных задач по выбранной специализации; - иметь представления о компьютерных сетях и их роли в современном мире; об общих принципах разработки и функционирования интернет-приложений; - понимать основные принципы дискретизации различных видов информации; уметь определять информационный объем текстовых, графических и звуковых данных при заданных параметрах дискретизации; - уметь строить неравномерные коды, допускающие однозначное декодирование сообщений (префиксные коды); использовать простейшие коды, которые позволяют обнаруживать и исправлять ошибки при передаче данных; - владеть теоретическим аппаратом, позволяющим осуществлять представление заданного натурального числа в различных системах счисления; выполнять преобразования логических выражений, используя законы алгебры логики; определять кратчайший путь во взвешенном графе и количество путей между вершинами ориентированного ациклического графа; - уметь читать и понимать программы, реализующие несложные алгоритмы обработки числовых и текстовых данных (в том числе массивов и символьных строк) на выбранном для изучения
---------------------	---	---

		универсальном языке программирования высокого уровня (Паскаль, Python, Java, C++, C#); анализировать алгоритмы с использованием таблиц трассировки; определять без использования компьютера результаты выполнения несложных программ, включающих циклы, ветвления и подпрограммы, при заданных исходных данных; модифицировать готовые программы для решения новых задач, использовать их в своих программах в качестве подпрограмм (процедур, функций); - уметь создавать структурированные текстовые документы и демонстрационные материалы с использованием возможностей современных программных средств и облачных сервисов; умение использовать табличные (реляционные) базы данных, в частности, составлять запросы в базах данных (в том числе вычисляемые запросы), выполнять сортировку и поиск записей в базе данных; наполнять разработанную базу данных; умение использовать электронные таблицы для анализа, представления и обработки данных (включая вычисление суммы, среднего арифметического, наибольшего и наименьшего значений, решение уравнений); - иметь представления о базовых принципах организации и функционирования компьютерных сетей; - уметь использовать при решении задач свойства позиционной записи чисел, алгоритмы построения записи числа в позиционной системе счисления с заданным основанием и построения числа по строке, содержащей запись этого числа в позиционной системе счисления с заданным основанием; уметь выполнять арифметические операции в позиционных системах счисления; умение строить логическое выражение в дизъюнктивной и
--	--	--

		конъюнктивной нормальных формах по заданной таблице истинности; исследовать область истинности высказывания, содержащего переменные; решать несложные логические уравнения; - понимать базовые алгоритмы обработки числовой и текстовой информации (запись чисел в позиционной системе счисления, делимость целых чисел; нахождение всех простых чисел в заданном диапазоне; обработка многоразрядных целых чисел; анализ символьных строк и других), алгоритмов поиска и сортировки; умение определять сложность изучаемых в курсе базовых алгоритмов (суммирование элементов массива, сортировка массива, переборные алгоритмы, двоичный поиск) и приводить примеры нескольких алгоритмов разной сложности для решения одной задачи; - владеть универсальным языком программирования высокого уровня (Паскаль, Python, Java, C++, C#), представлениями о базовых типах данных и структурах данных; умение использовать основные управляющие конструкции; уметь осуществлять анализ предложенной программы: определять результаты работы программы при заданных исходных данных; определять, при каких исходных данных возможно получение указанных результатов; выявлять данные, которые могут привести к ошибке в работе программы; формулировать предложения по улучшению программного кода; - уметь разрабатывать и реализовывать в виде программ базовые алгоритмы; использовать в программах данные различных типов с учетом ограничений на диапазон их возможных значений, применять при решении задач структуры данных (списки, словари, стеки, очереди, деревья); применять стандартные и собственные подпрограммы для
--	--	--

		обработки числовых данных и символьных строк; использовать при разработке программ библиотеки подпрограмм; знать функциональные возможности инструментальных средств среды разработки; умение использовать средства отладки программ в среде программирования; умение документировать программы; - уметь создавать веб-страницы; умение использовать электронные таблицы для анализа, представления и обработки данных (включая выбор оптимального решения, подбор линии тренда, решение задач прогнозирования); владеть основными сведениями о базах данных, их структуре, средствах создания и работы с ними; использовать табличные (реляционные) базы данных и справочные системы
--	--	--

Распределение тем учебного материала дисциплины «Информатика»

Календарно-тематический план содержания, осваиваемого во взаимодействии с преподавателем и дистанционно

Наименование темы	Вид учебных занятий	Часы
Модуль 1. Аналитика и визуализация данных на Python		34
в том числе практические занятия		
Практическая работа: написание программ по заданиям (Счастливая последовательность, Программа – разделители, Программа - вычисления по формуле, Программа - квадратное уравнение.	Практическое занятие	2
Практическая работа: написание программы по вычислениям с помощью библиотеки Math	Практическое занятие	2
Практическая работа: написание программы по вычислению площади и длины окружности	Практическое занятие	2
Всего:		6

Аннотация материала

Практика 1–3. Списки. Методы split и join. Файловый ввод/вывод

В данных практиках рассмотрены методы работы со списками и файлами в Python. Основное внимание уделено методам split и join, которые позволяют разбивать строку на части и объединять их обратно. Также описаны принципы файлового ввода/вывода, включая чтение данных из файла, запись данных в файл и работу с файлами различных форматов.

Практика 4–7. Рекурсия. Множества. Словари

Данные практики охватывают темы рекурсии, множеств и словарей на Python. Рекурсия рассматривается как метод программирования, который позволяет решать задачи путем многократного вызова самого себя до достижения конечного условия. Описываются различные примеры использования рекурсивных функций для решения задач, таких как вычисление факториалов, разбиение чисел на простые множители и сортировка списков.

Также уделяется внимание множествам и словарям – структурам данных, которые часто используются в программировании. Рассматриваются способы создания и манипуляции множествами и словарями, включая операции добавления, удаления и поиска элементов. Приводятся примеры кода, демонстрирующие применение этих структур данных для решения реальных задач.

Контроль и оценивание

Контроль и оценивание осуществляется в рамках действующей в филиале модульно-рейтинговой системы оценивания освоения содержания обучения, входит в план рейтинга по учебной дисциплине. План рейтинга по каждой теме включает в себя выполнение различного рода работ в дистанционной форме. Таким образом, за практические работы в дистанционной форме каждый студент может получить max = 5 баллов

Содержание дистанционного материала

Практика 1. Списки. Методы `split` и `join`. Файловый ввод/вывод

Списки

В предыдущих уроках мы работали с последовательностями чисел, символов, строк, но не сохраняли всю последовательность в памяти компьютера, а обрабатывали ее поэлементно, считывая раз за разом новый элемент. Однако во многих задачах требуется **сохранять всю последовательность**. Например, классическая задача сортировки (упорядочения) некоторой последовательности требует сохранения всех данных в памяти компьютера. Увы, не сохранив, их невозможно отсортировать. И тут на помощь приходит структура данных, которая в большинстве языков программирования называется массивом. В Python она называется **списком**.

Создание списка

Чтобы создать список, нужно перечислить его элементы через запятую в квадратных скобках:

```
numbers = [2, 4, 6, 8, 10]
```

```
languages = ['Python', 'C#', 'C++', 'Java']
```

Список `numbers` состоит из 5 элементов, и каждый из них — целое число.

- `numbers[0] == 2;`
- `numbers[1] == 4;`
- `numbers[2] == 6;`
- `numbers[3] == 8;`
- `numbers[4] == 10.`

Список `languages` состоит из 4 элементов, каждый из которых — строка.

- `languages[0] == 'Python';`
- `languages[1] == 'C#';`
- `languages[2] == 'C++';`
- `languages[3] == 'Java'.`

Список может содержать значения **разных типов данных**:

```
info = ['Timur', 1992, 61.5]
```

Список `info` содержит строковое значение, целое число и число с плавающей точкой.

- `info[0] == 'Timur';`
- `info[1] == 1992;`
- `info[2] == 61.5.`

Пустой список

Создать пустой список можно двумя способами:

1. Использовать пустые квадратные скобки `[]`;
2. Использовать встроенную функцию, которая называется `list`.

Следующие две строки кода создают пустой список:

```
mylist = [] # пустой список  
mylist = list() # пустой список
```

Вывод списка

Для вывода всего списка можно применить функцию `print()`:

```
numbers = [2, 4, 6, 8, 10]  
languages = ['Python', 'C#', 'C++', 'Java']  
print(numbers)  
print(languages)
```

Функция `print()` выводит на экран элементы списка, в квадратных скобках, разделенные запятыми:

```
[2, 4, 6, 8, 10]  
['Python', 'C#', 'C++', 'Java']
```

Встроенная функция list

Python имеет встроенную функцию `list()`, которая помимо создания пустого списка может преобразовывать некоторые типы объектов в списки.

Например, мы знаем, что функция `range()` создает последовательность целых чисел в заданном диапазоне. Для преобразования этой последовательности в список мы пишем следующий код:

```
numbers = list(range(5))
```

Во время исполнения этого кода происходит следующее:

1. Вызывается функция `range()`, в которую в качестве аргумента передается число 5;
2. Эта функция возвращает последовательность чисел `0, 1, 2, 3, 4`;
3. Последовательность чисел `0, 1, 2, 3, 4` передается в качестве аргумента в функцию `list()`;
4. Функция `list()` возвращает список `[0, 1, 2, 3, 4]`;
5. Список `[0, 1, 2, 3, 4]` присваивается переменной `numbers`.

Вот еще один пример:

```
even_numbers = list(range(0, 10, 2)) # список содержит четные  
числа 0, 2, 4, 6, 8  
odd_numbers = list(range(1, 10, 2)) # список содержит нечетные  
числа 1, 3, 5, 7, 9
```

Точно также с помощью функции `list()` мы можем создать список из символов строки. Для преобразования строки в список мы пишем следующий код:

```
s = 'abcde'

chars = list(s) # список содержит символы 'a', 'b', 'c', 'd',
'e'
```

Во время исполнения этого кода происходит следующее:

1. Вызывается функция `list()`, в которую в качестве аргумента передается строка `'abcde'`;
2. Функция `list()` возвращает список `['a', 'b', 'c', 'd', 'e']`;
3. Список `['a', 'b', 'c', 'd', 'e']` присваивается переменной `chars`.

2. Основы работы со списками

Работа со списками очень сильно напоминает работу со строками, поскольку и списки, и строки содержат отдельные элементы: элементы списка могут иметь произвольный тип, а элементами строк всегда являются символы. Многое из того, что мы делали со строками, доступно и при работе со списками.

2.1. Функция `len()`

Длиной списка называется количество его элементов. Для того, чтобы посчитать длину списка мы используем встроенную функцию `len()` (от слова *length* – длина).

Следующий программный код:

```
numbers = [2, 4, 6, 8, 10]

languages = ['Python', 'C#', 'C++', 'Java']

print(len(numbers)) # выводим длину списка numbers

print(len(languages)) # выводим длину списка languages

print(len(['apple', 'banana', 'cherry'])) # выводим длину
списка, состоящего из 3 элементов
```

выведет:

```
5
4
3
```

2.2. Оператор принадлежности `in`

Оператор `in` позволяет проверить, содержит ли список некоторый элемент.

Рассмотрим следующий код:

```
numbers = [2, 4, 6, 8, 10]

if 2 in numbers:

    print('Список numbers содержит число 2')
```

```
else:
```

```
    print('Список numbers не содержит число 2')
```

Такой код проверяет, содержит ли список `numbers` число 2 и выводит соответствующий текст:

```
Список numbers содержит число 2
```

Мы можем использовать оператор `in` вместе с логическим оператором `not`. Например

```
numbers = [2, 4, 6, 8, 10]
```

```
if 0 not in numbers:
```

```
    print('Список numbers не содержит нулей')
```

Примеры

Фильтрация четных чисел:

```
all_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
even_numbers = [num for num in all_numbers if num % 2 == 0]
```

```
print(f'Исходный список: {all_numbers}')
```

```
print(f'Список четных чисел: {even_numbers}')
```

Обработка текста в списке строк:

```
words = ["apple", "banana", "orange", "grape"]
```

```
capitalized_words = [word.capitalize() for word in words]
```

```
print(f'Исходный список слов: {words}')
```

```
print(f'Список слов с заглавными буквами: {capitalized_words}')
```

У вас есть список чисел, и вам нужно найти среднее арифметическое всех чисел в этом списке.

```
# Исходный список чисел
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# Фильтруем только четные числа и суммируем их
```

```
sum_of_even_numbers = sum(num for num in numbers if num % 2 == 0)
```

```
print(f'Сумма четных чисел: {sum_of_even_numbers}')
```

Разделите этот список на два подсписка: один содержит только четные числа, а другой – только нечетные. Затем отсортируйте оба подсписка в порядке возрастания.

```
# Исходный список чисел
```

```
numbers = [7, 2, 5, 8, 3, 9, 4, 6]
```

```
# Разделяем на четные и нечетные числа
```

```
even_numbers = [num for num in numbers if num % 2 == 0]
```

```
odd_numbers = [num for num in numbers if num % 2 != 0]
```

```
# Сортируем оба списка
```

```
even_numbers.sort()
```

```
odd_numbers.sort()
```

```
# Выводим результат
```

```
print("Четные числа:", even_numbers)
```

```
print("Нечетные числа:", odd_numbers)
```

Практика 2. Списки. Методы split и join. Файловый ввод/вывод

Основы работы со списками

Работа со списками очень сильно напоминает работу со строками, поскольку и списки, и строки содержат отдельные элементы: элементы списка могут иметь произвольный тип, а элементами строк всегда являются символы. Многое из того, что мы делали со строками, доступно и при работе со списками.

Индексация

При работе со строками мы использовали **индексацию**, то есть обращение к конкретному символу строки по его индексу. Аналогично, можно индексировать и списки.

Для индексации списков в Python используются квадратные скобки `[]`, в которых указывается индекс (номер) нужного элемента в списке:

Пусть `numbers = [2, 4, 6, 8, 10]`.

Таблица ниже, показывает, как работает индексация:

Выражение	Результат	Пояснение
<code>numbers[0]</code>	2	первый элемент списка
<code>numbers[1]</code>	4	второй элемент списка
<code>numbers[2]</code>	6	третий элемент списка
<code>numbers[3]</code>	8	четвертый элемент списка
<code>numbers[4]</code>	10	пятый элемент списка

Срезы

Рассмотрим список `numbers = [2, 4, 6, 8, 10]`.

С помощью среза мы можем получить несколько элементов списка, создав диапазон индексов разделенных двоеточием `numbers[x:y]`.

Следующий программный код:

```
print(numbers[1:3])
```

```
print(numbers[2:5])
```

ВЫВОДИТ:

```
[4, 6]
```

```
[6, 8, 10]
```

При построении среза `numbers[x:y]` первое число – это то место, где начинается срез (**включительно**), а второе – это место, где заканчивается срез (**невключительно**). Разрезая списки, мы создаем новые списки, по сути, подсписки исходного.

При использовании срезов со списками мы также можем опускать второй параметр в срезе `numbers[x:]` (но поставить двоеточие), тогда срез берется до конца списка. Аналогично, если опустить первый параметр `numbers[:y]`, то можно взять срез от начала списка.

Использование срезов для изменения элементов в заданном диапазоне

Для изменения целого диапазона элементов списка можно использовать срезы. Например, если мы хотим перевести на русский язык названия фруктов `'banana', 'cherry', 'kiwi'`, то это можно сделать с помощью среза.

Следующий программный код:

```
fruits = ['apple', 'apricot', 'banana', 'cherry', 'kiwi',  
'lemon', 'mango']
```

```
fruits[2:5] = ['банан', 'вишня', 'киви']
```

```
print(fruits)
```

ВЫВОДИТ:

```
['apple', 'apricot', 'банан', 'вишня', 'киви', 'lemon', 'mango']
```

Операция конкатенации + и умножения на число *

Мы можем применять операторы `+` и `*` для списков подобно тому, как мы это делали со строками.

Следующий программный код:

```
print([1, 2, 3, 4] + [5, 6, 7, 8])
```

```
print([7, 8] * 3)
```

```
print([0] * 10)
```

ВЫВОДИТ:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
[7, 8, 7, 8, 7, 8]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Встроенные функции `sum()`, `min()`, `max()`

Встроенная функция `sum()` принимает в качестве параметра список чисел и вычисляет сумму его элементов.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print('Сумма всех элементов списка =', sum(numbers))
```

ВЫВОДИТ:

```
Сумма всех элементов списка = 55
```

Встроенные функции `min()` и `max()` принимают в качестве параметра список и находят минимальный и максимальный элементы соответственно.

Следующий программный код:

```
numbers = [3, 4, 10, 3333, 12, -7, -5, 4]

print('Минимальный элемент =', min(numbers))

print('Максимальный элемент =', max(numbers))
```

ВЫВОДИТ:

```
Минимальный элемент = -7
```

```
Максимальный элемент = 3333
```

Методы добавления и удаления элементов

Добавление элементов

Мы научились создавать статические списки, то есть списки, элементы которых известны на этапе создания. Следующий шаг – научиться добавлять элементы в уже существующие списки.

Метод `append()`

Для добавления нового элемента **в конец списка** используется метод `append()`.

Следующий программный код:

```
numbers = [1, 1, 2, 3, 5, 8, 13] # создаем список

numbers.append(21) # добавляем число 21 в конец списка

numbers.append(34) # добавляем число 34 в конец списка

print(numbers)
```

Выведет:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Обратите внимание, для того чтобы использовать метод `append()`, нужно, чтобы список был создан, при этом он может быть пустым.

Следующий программный код:

```
numbers = [] # создаем пустой список

numbers.append(1)

numbers.append(2)

numbers.append(3)

print(numbers)
```

выведет:

```
[1, 2, 3]
```

```
numbers = [] # создаем пустой список

numbers[0] = 1

numbers[1] = 2

numbers[2] = 3

print(numbers)
```

приводит к ошибке:

```
IndexError: list assignment index out of range
```

Метод `extend()`

Можно также расширить список другим списком, путем вызова метода `extend()`.

Следующий программный код:

```
numbers = [0, 2, 4, 6, 8, 10]

odds = [1, 3, 5, 7]

numbers.extend(odds)

print(numbers)
```

выведет:

```
[0, 2, 4, 6, 8, 10, 1, 3, 5, 7]
```

Метод `extend()` как бы расширяет один список, добавляя к нему элементы другого списка.

Отличие между методами `append()` и `extend()` проявляется при добавлении строки к списку.

Следующий программный код:

```
words1 = ['iq option', 'stepik', 'beegEEK']  
words2 = ['iq option', 'stepik', 'beegEEK']  
words1.append('python')  
words2.extend('python')  
print(words1)  
print(words2)
```

выведет:

```
['iq option', 'stepik', 'beegEEK', 'python']  
['iq option', 'stepik', 'beegEEK', 'p', 'y', 't', 'h', 'o', 'n']
```

Метод `append()` добавляет строку `'python'` целиком к списку, а метод `extend()` разбивает строку `'python'` на символы `'p', 'y', 't', 'h', 'o', 'n'` и их добавляет в качестве элементов списка.

Удаление элементов

С помощью оператора `del` можно удалять элементы списка по определенному индексу. Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
del numbers[5] # удаляем элемент имеющий индекс 5  
print(numbers)
```

выведет:

```
[1, 2, 3, 4, 5, 7, 8, 9]
```

Элемент под указанным индексом удаляется, а список перестраивается.

Оператор `del` работает и со срезами: мы можем удалить целый диапазон элементов списка.

Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
del numbers[2:7] # удаляем элементы с 2 по 6 включительно  
print(numbers)
```

выведет:

```
[1, 2, 8, 9]
```

Мы можем удалить все элементы на четных позициях (0, 2, 4, ...) исходного списка. Следующий программный код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
del numbers[::2]
```

```
print(numbers)
```

выведет:

```
[2, 4, 6, 8]
```

Примеры

Список чисел и их сумма:

```
numbers = [2, 4, 6, 8, 10]
```

```
sum_of_numbers = sum(numbers)
```

```
print(f"Список чисел: {numbers}")
```

```
print(f"Сумма чисел: {sum_of_numbers}")
```

у вас есть список пользователей, и вы хотите добавлять новых пользователей, удалять тех, кто отписался, и выводить список активных пользователей.

```
# Исходный список пользователей
```

```
users = ["Alice", "Bob", "Charlie", "David"]
```

```
# Добавление нового пользователя
```

```
new_user = "Eve"
```

```
users.append(new_user)
```

```
# Удаление пользователя, который отписался
```

```
unsubscribed_user = "Charlie"
```

```
if unsubscribed_user in users:
```

```
    users.remove(unsubscribed_user)
```

```
# Вывод активных пользователей
```

```
print("Активные пользователи:", users)
```

у вас есть список студентов в классе, и вы хотите написать алгоритм для добавления новых студентов и удаления тех, кто закончил обучение.

```
# Исходный список студентов
```

```
students = ["Анна", "Петр", "Мария", "Иван", "Елена"]
```

```
# Цикл для добавления новых студентов
```

```
for i in range(3):
```

```
    new_student = input("Введите имя нового студента: ")
```

```
    students.append(new_student)
```

```
# Вывод текущего состояния класса
```

```

print("Список студентов после добавления:", students)

# Цикл для удаления студентов, закончивших обучение
for i in range(2):

    graduated_student = input("Введите имя студента, который закончил
обучение: ")

    if graduated_student in students:

        students.remove(graduated_student)

    else:

        print("Студент не найден в списке.")

# Вывод окончательного состояния класса
print("Список студентов после удаления:", students)

```

Практика 3. Списки. Методы `split` и `join`. Файловый ввод/вывод

Метод `split()`

Метод `split()` разбивает строку на слова, используя в качестве разделителя последовательность пробельных символов.

Следующий программный код:

```

s = 'Python is the most powerful language'

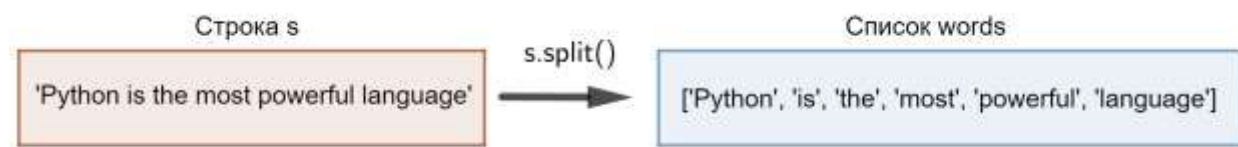
words = s.split()

print(words)

```

выведет:

```
['Python', 'is', 'the', 'most', 'powerful', 'language']
```



Таким образом, вызов метода `split()` разбивает строку на слова и возвращает список, содержащий все слова.

Рассмотрим следующий программный код:

```
numbers = input().split()
```

Если при запуске этой программы ввести строку `1 2 3 4 5`, то список `numbers` будет следующим `['1', '2', '3', '4', '5']`. Обратите внимание, что список будет состоять

из строк, а не из чисел. Если требуется получить именно список чисел, то затем нужно элементы списка по одному преобразовать в числа:

```
numbers = input().split()

for i in range(len(numbers)):

    numbers[i] = int(numbers[i])
```

Необязательный параметр

У метода `split()` есть необязательный параметр, который определяет, какой набор символов будет использоваться в качестве разделителя между элементами списка. Например, вызов метода `split('.')` вернет список, полученный разделением исходной строки по символу `'.'`.

Следующий программный код:

```
ip = '192.168.1.24'

numbers = ip.split('.') # указываем явно разделитель

print(numbers)

выведет список:

['192', '168', '1', '24']
```



Метод join()

Метод `join()` собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод.

Следующий программный код:

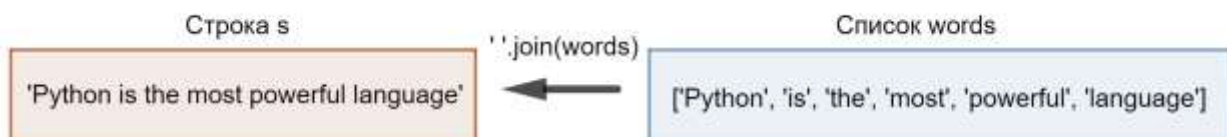
```
words = ['Python', 'is', 'the', 'most', 'powerful', 'language']

s = ' '.join(words)

print(s)
```

выведет:

```
Python is the most powerful language
```



Обратите внимание, все слова разделены одним пробелом, поскольку метод `join()` вызывался на строке состоящей из одного символа пробела `' '`.

Рассмотрим еще пару примеров:

```
words = ['Мы', 'учим', 'язык', 'Python']

print('*'.join(words))

print('-'.join(words))

print('?'.join(words))

print('!'.join(words))

print('*****'.join(words))

print('abc'.join(words))

print('123'.join(words))
```

Результатом выполнения такого кода будет:

```
Мы*учим*язык*Python

Мы-учим-язык-Python

Мы?учим?язык?Python

Мы!учим!язык!Python

Мы*****учим*****язык*****Python

МыabcучимabcязыкabcPython

Мы123учим123язык123Python
```

Примечания

Примечание 1. Существует большая разница между результатами вызова методов `s.split()` и `s.split(' ')`. Разница в поведении проявляется, когда строка содержит несколько пробелов между словами.

Следующий программный код:

```
s = 'Python  is  the  most  powerful  language'

words1 = s.split()

words2 = s.split(' ')

print(words1)

print(words2)
```

Выведет списки:

```
['Python', 'is', 'the', 'most', 'powerful', 'language']  
['Python', '', '', '', 'is', '', '', 'the', '', 'most', '', 'powerful', '', 'language']
```

Примечание 2. Методы `split()` и `join()` являются строковыми методами. Следующий код приводит к ошибке:

```
print([1, 2].split())  
print([1, 2].join([3, 4, 5]))
```

Примечание 3. Строковый метод `join()` работает только со списком строк. Следующий код приводит к ошибке:

```
numbers = [1, 2, 3, 4] # список чисел  
  
s = '*'.join(numbers)  
  
print(s)
```

Метод `insert()`

Метод `insert()` позволяет вставлять значение в список в заданной позиции. В него передается два аргумента:

1. `index`: индекс, задающий место вставки значения;
2. `value`: значение, которое требуется вставить.

Когда значение вставляется в список, список расширяется в размере, чтобы разместить новое значение. Значение, которое ранее находилось в заданной индексной позиции, и все элементы после него сдвигаются на одну позицию к концу списка.

Следующий программный код:

```
names = ['Gvido', 'Roman', 'Timur']  
  
print(names)  
  
names.insert(0, 'Anders')  
  
print(names)  
  
names.insert(3, 'Josef')  
  
print(names)
```

Выведет:

```
['Gvido', 'Roman', 'Timur']  
['Anders', 'Gvido', 'Roman', 'Timur']  
['Anders', 'Gvido', 'Roman', 'Josef', 'Timur']
```


Если указан недопустимый индекс, то во время выполнения программы не происходит ошибки. Если задан индекс за пределами конца списка, то значение будет добавлено в конец списка. Если применен отрицательный индекс, который указывает на недопустимую позицию, то значение будет вставлено в начало списка.

Метод `index()`

Метод `index()` возвращает индекс первого элемента, значение которого равняется переданному в метод значению. Таким образом, в метод передается один параметр:

1. `value`: значение, индекс которого требуется найти.

Если элемент в списке не найден, то во время выполнения происходит ошибка.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']

position = names.index('Timur')

print(position)
```

выведет:

2

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']

position = names.index('Anders')

print(position)
```

приводит к ошибке:

```
ValueError: 'Anders' is not in list
```

Чтобы избежать таких ошибок, можно использовать метод `index()` вместе с оператором принадлежности `in`:

```
names = ['Gvido', 'Roman' , 'Timur']

if 'Anders' in names:

    position = names.index('Anders')

    print(position)

else:

    print('Такого значения нет в списке')
```

Метод `remove()`

Метод `remove()` удаляет первый элемент, значение которого равняется переданному в метод значению. В метод передается один параметр:

2. `value`: значение, которое требуется удалить.

Метод уменьшает размер списка на один элемент. Все элементы после удаленного элемента смещаются на одну позицию к началу списка. Если элемент в списке не найден, то во время выполнения происходит ошибка.

Следующий программный код:

```
food = ['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']  
  
print(food)  
  
food.remove('Рис')  
  
print(food)
```

выведет:

```
['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']  
  
['Курица', 'Рыба', 'Брокколи', 'Рис']
```

Метод `pop()`

Метод `pop()` удаляет элемент по указанному индексу и возвращает его. В метод `pop()` передается один **необязательный** аргумент:

`index`: индекс элемента, который требуется удалить.

Если индекс не указан, то метод удаляет и возвращает последний элемент списка. Если список пуст или указан индекс за пределами диапазона, то во время выполнения происходит ошибка.

Следующий программный код:

```
names = ['Gvido', 'Roman', 'Timur']  
  
item = names.pop(1)  
  
print(item)  
  
print(names)
```

выведет:

```
Roman  
['Gvido', 'Timur']
```

Метод `count()`

Метод `count()` возвращает количество элементов в списке, значения которых равны переданному в метод значению.

Таким образом, в метод передается один параметр:

`value`: значение, количество элементов, равных которому, нужно посчитать.

Если значение в списке не найдено, то метод возвращает 0.

Следующий программный код:

```
names = ['Timur', 'Gvido', 'Roman', 'Timur', 'Anders', 'Timur']

cnt1 = names.count('Timur')

cnt2 = names.count('Gvido')

cnt3 = names.count('Josef')


print(cnt1)

print(cnt2)

print(cnt3)
```

ВЫВЕДЕТ:

```
3
1
0
```

Метод `reverse()`

Метод `reverse()` инвертирует порядок следования значений в списке, то есть меняет его на противоположный.

Следующий программный код:

```
names = ['Gvido', 'Roman' , 'Timur']

names.reverse()

print(names)
```

ВЫВЕДЕТ:

```
['Timur', 'Roman', 'Gvido']
```

Метод `sort()`

В Python списки имеют встроенный метод `sort()`, который сортирует элементы списка по возрастанию или убыванию.

Следующий программный код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]

a.sort()

print('Отсортированный список:', a)
```

ВЫВЕДЕТ:

```
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

По умолчанию метод `sort()` сортирует список по возрастанию. Если требуется отсортировать список по убыванию, необходимо явно указать параметр `reverse = True`.

Следующий программный код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]

a.sort(reverse = True) # сортируем по убыванию

print('Отсортированный список:', a)
```

выведет:

```
Отсортированный список: [1000, 99, 45, 34,
```

Практика 4. Рекурсия. Множества. Словари

Создание множеств

Множество в Python – это неупорядоченная коллекция уникальных элементов. Множества в Python поддерживают базовые операции множеств, такие как объединение, пересечение, разность, а также проверку вхождения элемента. Создание множества в Python осуществляется с использованием фигурных скобок `{}` или функции `set()`.

Пример создания множества и основных операций:

```
# Создание множества с использованием фигурных скобок
fruits = {"apple", "banana", "orange"}
```

```
# Добавление элемента в множество
fruits.add("grape")
```

```
# Попытка добавить существующий элемент (не приведет к ошибке, но
ничего не изменит)
fruits.add("apple")
```

```
# Удаление элемента из множества
fruits.remove("banana")
```

```
# Проверка вхождения элемента в множество
is_apple_in_set = "apple" in fruits
```

```
# Вывод множества
print(fruits)
print("Is 'apple' in the set?", is_apple_in_set)
```

Пошаговое объяснение:

1. Создается множество `fruits` с помощью фигурных скобок `{}`. В данном примере, множество содержит элементы "apple", "banana" и "orange".

2. С использованием метода `add()`, добавляется новый элемент "grape" в множество.
3. Пытаемся добавить существующий элемент "apple". Это не вызывает ошибку, но множество остается неизменным, так как множество содержит только уникальные элементы.
4. С использованием метода `remove()`, удаляется элемент "banana" из множества.
5. С использованием оператора `in`, проверяется вхождение элемента "apple" в множество, и результат сохраняется в переменную `is_apple_in_set`.
6. Выводится содержимое множества и результат проверки вхождения "apple" в множество.
7. Множества в Python предоставляют удобный способ работы с уникальными элементами и поддерживают разнообразные операции для работы с ними.

Заполнение множества (For, While)

Заполнение множества при помощи циклов, представляет собой процесс добавления элементов в множество с использованием циклов, таких как цикл `for` или `while`. Это может быть полезным, например, при обработке большого объема данных или при создании множества на основе каких-то условий.

Пример заполнения множества при помощи цикла:

```
# Исходный список цветов
colors_list = ["red", "green", "blue", "yellow", "green", "orange"]

# Создание пустого множества
unique_colors = set()

# Заполнение множества при помощи цикла for
for color in colors_list:
    unique_colors.add(color)

# Вывод заполненного множества
print("Unique colors:", unique_colors)
```

Пошаговое объяснение:

1. Задается исходный список `colors_list`, который содержит повторяющиеся значения.
2. Создается пустое множество `unique_colors` с использованием функции `set()`. Это множество будет использоваться для хранения уникальных цветов.
3. Запускается цикл `for`, который проходит по каждому элементу `color` в списке `colors_list`.
4. Для каждого элемента цвета `color` вызывается метод `add()`, который добавляет этот цвет в множество `unique_colors`. Поскольку множество не может содержать повторяющиеся элементы, это гарантирует уникальность элементов в множестве.
5. В конце цикла выводится заполненное множество `unique_colors`.

После выполнения цикла в множестве `unique_colors` будут только уникальные цвета из исходного списка.

Методы и функции множеств

Методы и функции множеств в Python:

Множества в Python предоставляют набор методов и функций для выполнения различных операций с элементами множества. Некоторые из основных методов включают добавление элементов, удаление элементов, проверку вхождения, объединение, пересечение и другие.

Пример:

```

# Создание двух множеств
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Добавление элемента в множество
set1.add(6)

# Удаление элемента из множества
set2.remove(7)

# Проверка вхождения элемента в множество
is_present = 4 in set1

# Объединение множеств
union_set = set1.union(set2)

# Пересечение множеств
intersection_set = set1.intersection(set2)

# Разность множеств
difference_set = set1.difference(set2)

# Вывод результатов
print("Set 1:", set1)
print("Set 2:", set2)
print("Is 4 present in set 1?", is_present)
print("Union of sets:", union_set)
print("Intersection of sets:", intersection_set)
print("Difference of sets:", difference_set)

```

Пошаговое объяснение:

1. Создаются два множества set1 и set2.
2. Метод add() используется для добавления элемента 6 в множество set1.
3. Метод remove() удаляет элемент 7 из множества set2.
4. С использованием оператора in проверяется вхождение элемента 4 в множество set1, и результат сохраняется в переменной is_present.
5. Метод union() создает новое множество, содержащее все уникальные элементы из обоих множеств.
6. Метод intersection() создает новое множество, содержащее элементы, которые присутствуют в обоих множествах.
7. Метод difference() создает новое множество, содержащее элементы из set1, которых нет в set2.
8. Результаты выводятся на экран.

discard() метод.

discard(element) — удаляет элемент из множества, если он присутствует. Если элемент отсутствует, метод не вызывает ошибку.

`pop()` – удаляет и возвращает произвольный элемент из множества. Если множество пусто, вызывает ошибку `KeyError`.

`clear()` – удаляет все элементы из множества, делая его пустым.

Пример:

```
# Создание множества
```

```
my_set = {1, 2, 3, 4, 5}
```

```
# Использование метода discard() для удаления элемента 3
```

```
my_set.discard(3)
```

```
# Вывод текущего состояния множества
```

```
print("After using discard(3):", my_set)
```

```
# Использование метода pop() для удаления и возврата произвольного  
элемента
```

```
popped_element = my_set.pop()
```

```
# Вывод текущего состояния множества и удаленного элемента
```

```
print("After using pop():", my_set)
```

```
print("Popped element:", popped_element)
```

```
# Использование метода clear() для удаления всех элементов из множества
```

```
my_set.clear()
```

```
# Вывод состояния множества после использования clear()
```

```
print("After using clear():", my_set)
```

Пошаговое объяснение:

1. Создается множество `my_set` с элементами от 1 до 5.
2. Метод `discard(3)` используется для удаления элемента 3 из множества. Выводится текущее состояние множества.
3. Метод `pop()` используется для удаления и возврата произвольного элемента из множества. Удаленный элемент сохраняется в переменной `popped_element`. Выводится текущее состояние множества и удаленный элемент.
4. Метод `clear()` используется для удаления всех элементов из множества. Выводится пустое множество.

Метод пересечения (`intersection()`)

`intersection(*other_sets)`. Возвращает новое множество, содержащее элементы, присутствующие во всех указанных множествах.

Метод разности (`difference()`):

`difference(*other_sets)`. Возвращает новое множество, содержащее элементы, присутствующие в текущем множестве, но отсутствующие в любом из указанных множеств.

Метод симметричной разности (`symmetric_difference()`)

`symmetric_difference(other_set)`. Возвращает новое множество, содержащее элементы, присутствующие только в текущем множестве или только в указанном множестве, но не в обоих.

Пример:

```
# Создание двух множеств
```

```
set1 = {1, 2, 3, 4, 5}
```

```
set2 = {3, 4, 5, 6, 7}
```

```
# Использование метода intersection() для нахождения пересечения множеств  
intersection_result = set1.intersection(set2)
```

```
# Вывод результатов
```

```
print("Set 1:", set1)
```

```
print("Set 2:", set2)
```

```
print("Intersection of sets:", intersection_result)
```

```
# Использование метода difference() для нахождения разности множеств  
difference_result = set1.difference(set2)
```

```
# Вывод результатов
```

```
print("Difference of sets (set1 - set2):", difference_result)
```

```
# Использование метода symmetric_difference() для нахождения  
симметричной разности множеств
```

```
symmetric_difference_result = set1.symmetric_difference(set2)
```

```
# Вывод результатов
```

```
print("Symmetric difference of sets:", symmetric_difference_result)
```

Пошаговое объяснение:

1. Создаются два множества set1 и set2.
2. Метод intersection() используется для нахождения пересечения множеств set1 и set2. Результат сохраняется в переменной intersection_result.
3. Выводится исходное состояние множеств и результат пересечения.
4. Метод difference() используется для нахождения разности множеств set1 - set2. Результат сохраняется в переменной difference_result.
5. Выводится результат разности множеств.
6. Метод symmetric_difference() используется для нахождения симметричной разности множеств set1 и set2. Результат сохраняется в переменной symmetric_difference_result.
7. Выводится результат симметричной разности множеств.

Методы и функции множеств предоставляют широкие возможности для работы с данными и выполнения различных операций, таких как объединение, пересечение и разность множеств.

Практика 5. Рекурсия. Множества. Словари

Суммирование целых

Суммирование целых чисел при помощи рекурсии – это процесс вычисления суммы всех целых чисел от заданного числа до нуля с использованием рекурсивных вызовов. Каждый шаг

рекурсии уменьшает заданное число, приближаясь к базовому случаю, в котором суммирование завершается.

Пример суммирования целых чисел с использованием рекурсии в Python:

```
def sum_integers(n):
    # Базовый случай: если n равно 0, возвращаем 0
    if n == 0:
        return 0
    # Шаг рекурсии: возвращаем сумму n и результата рекурсивного вызова
    # для n-1
    else:
        return n + sum_integers(n - 1)

# Пример вызова функции
result = sum_integers(5)
print(result)
```

Пошаговое объяснение:

1. Вызывается функция `sum_integers` с аргументом 5.
2. Функция проверяет базовый случай: если `n` равно 0, то возвращается 0.
3. В противном случае, функция возвращает сумму `n` и результата рекурсивного вызова `sum_integers(n - 1)`.
4. Рекурсивный вызов происходит с аргументом 4.
5. Процесс повторяется, пока не достигнут базовый случай (`n == 0`).
6. В итоге, суммируются все числа от 5 до 1, и результат (15) возвращается в основной вызов.
7. С использованием рекурсии мы эффективно вычисляем сумму целых чисел от заданного числа до нуля. Каждый рекурсивный вызов представляет из себя шаг с уменьшением значения `n`.

Числа Фибоначчи

Рекурсия чисел Фибоначчи – это метод вычисления чисел Фибоначчи с использованием рекурсивных вызовов. Числа Фибоначчи представляют собой последовательность, в которой каждое число является суммой двух предыдущих чисел (за исключением первых двух, которые равны 0 и 1).

Пример рекурсивной функции для вычисления чисел Фибоначчи в Python:

```
def fibonacci(n):
    # Базовый случай: если n равно 0 или 1, возвращаем само число
    if n == 0 or n == 1:
        return n
    # Шаг рекурсии: возвращаем сумму двух предыдущих чисел Фибоначчи
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Пример вызова функции
result = fibonacci(5)
print(result)
```

Пошаговое объяснение:

1. Вызывается функция `fibonacci` с аргументом 5.
2. Функция проверяет базовый случай: если `n` равно 0 или 1, возвращается само число (5 не соответствует базовому случаю).
3. В противном случае, функция возвращает сумму двух рекурсивных вызовов: `fibonacci(n - 1)` и `fibonacci(n - 2)`.
4. Рекурсивные вызовы продолжаются до достижения базового случая.
5. Для `fibonacci(5)` рекурсивные вызовы раскрываются до `fibonacci(4)`, `fibonacci(3)`, и так далее.
6. Как только достигнут базовый случай (`n == 0` или `n == 1`), начинается возврат значений и подсчет суммы.
7. Рекурсивные вызовы собираются в обратном порядке, и в конечном итоге возвращается результат для `fibonacci(5)`.

С использованием рекурсии мы можем эффективно вычислить числа Фибоначчи, но следует отметить, что рекурсивный подход может привести к повторным вычислениям и быть менее эффективным для больших значений `n`.

Рекурсия для подсчета символов

Рекурсия для подсчета символов – это метод использования рекурсивных вызовов для подсчета количества определенных символов в строке или последовательности. Рекурсивная функция будет вызывать саму себя для обработки подстроки и пошагового подсчета символов.

Пример рекурсивной функции для подсчета символов в строке в Python:

```
def count_characters(string, char):
    # Базовый случай: если строка пуста, возвращаем 0
    if not string:
        return 0
    # Шаг рекурсии: сравниваем первый символ с искомым и добавляем 1,
    # если совпадает
    else:
        return (1 if string[0] == char else 0) + count_characters(string[1:], char)

# Пример вызова функции
result = count_characters("programming", "m")
print(result)
```

Пошаговое объяснение:

1. Вызывается функция `count_characters` с аргументами "programming" (строка) и "m" (искомый символ).
2. Функция проверяет базовый случай: если строка пуста, возвращается 0 (ничего не осталось для подсчета).
3. В противном случае функция проверяет, совпадает ли первый символ строки с искомым символом.
4. Если совпадает, добавляется 1 к результату, и функция вызывает саму себя для оставшейся части строки `string[1:]`.
5. Рекурсивные вызовы продолжаются до достижения базового случая, и значения возвращаются по цепочке.
6. Рекурсивные вызовы собираются в обратном порядке, и в конечном итоге возвращается общее количество вхождений искомого символа в строке.

С использованием рекурсии мы можем эффективно подсчитать количество определенных символов в строке, разбивая задачу на более простые подзадачи.

Проверка строки на палиндромность

Проверка строки на палиндромность при помощи рекурсии – это метод использования рекурсивных вызовов для определения, является ли заданная строка палиндромом. Палиндром – это строка, которая читается одинаково как с начала, так и с конца, игнорируя пробелы, знаки препинания и регистр.

Пример рекурсивной функции для проверки палиндромности строки в Python:

```
def is_palindrome(s):
    # Приведение строки к нижнему регистру и удаление пробелов
    s = s.lower().replace(" ", "")

    # Базовый случай: если длина строки меньше или равна 1, она считается
    # палиндромом
    if len(s) <= 1:
        return True
    # Шаг рекурсии: сравниваем первый и последний символы и вызываем для
    # сокращенной строки
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])

# Пример вызова функции
result = is_palindrome("A man a plan a canal Panama")
print(result)
```

Пошаговое объяснение:

1. Вызывается функция `is_palindrome` с аргументом "A man a plan a canal Panama".
2. Строка приводится к нижнему регистру и удаляются пробелы, чтобы упростить проверку палиндромности.
3. Базовый случай: если длина строки меньше или равна 1 (игнорируя пробелы), строка считается палиндромом, и функция возвращает `True`.
4. В противном случае функция сравнивает первый и последний символы строки.
5. Если они совпадают, функция вызывает саму себя для сокращенной строки между первым и последним символами (`s[1:-1]`).
6. Рекурсивные вызовы продолжаются до достижения базового случая.
7. Если все рекурсивные вызовы возвращают `True` (совпадение символов), в конечном итоге функция возвращает `True`.

С использованием рекурсии мы можем эффективно проверять строки на палиндромность, разбивая задачу на более простые проверки символов.

Практика 6. Рекурсия. Множества. Словари

Создание словарей

Создание словарей в Python

Словарь в Python представляет собой неупорядоченную коллекцию данных, состоящую из пар ключ-значение. Каждый элемент словаря имеет уникальный ключ, который используется для доступа к соответствующему значению. Словари создаются с использованием фигурных скобок {}, и элементы в словаре указываются в виде пары ключ: значение.

Пример:

```

# Создание словаря с информацией о человеке
person = {
    "name": "John",
    "age": 30,
    "city": "New York",
    "email": "john@example.com"
}

# Вывод исходного словаря
print("Original dictionary:", person)

# Доступ к значению по ключу
name_value = person["name"]
print("Name:", name_value)

# Добавление нового элемента в словарь
person["occupation"] = "Engineer"

# Вывод словаря после добавления нового элемента
print("Dictionary after adding 'occupation':", person)

# Изменение значения по ключу
person["age"] = 31

# Вывод словаря после изменения значения
print("Dictionary after changing 'age':", person)

# Удаление элемента из словаря
removed_value = person.pop("email")

# Вывод словаря после удаления элемента
print("Dictionary after removing 'email':", person)
print("Removed email:", removed_value)

```

Пошаговое объяснение:

1. Создается словарь `person` с данными о человеке, представленными в виде пар ключ-значение.
2. Исходный словарь выводится на экран.
3. Используется доступ по ключу (`person["name"]`) для получения значения имени, которое сохраняется в переменной `name_value`.
4. Добавляется новый элемент `"occupation"` в словарь.
5. Выводится словарь после добавления нового элемента.
6. Изменяется значение ключа `"age"` в словаре.
7. Выводится словарь после изменения значения.

8. Метод pop("email") используется для удаления элемента с ключом "email". Удаленное значение сохраняется в переменной removed_value.
9. Выводится словарь после удаления элемента и выводится удаленное значение "email".
Словари в Python предоставляют удобный способ хранения и обработки данных в формате ключ-значение.

Чтение, добавление и изменение словарей

Чтение значений по ключу

Для чтения значения из словаря используется оператор [], где внутри скобок указывается ключ.

Добавление новых элементов

Для добавления новых элементов в словарь используется оператор [] с новым ключом и значением.

Изменение значений по ключу

Изменение значения в словаре происходит путем обращения к ключу и присваивания нового значения.

Пример:

```
# Исходный словарь с информацией о студенте
```

```
student = {  
    "name": "Alice",  
    "age": 20,  
    "grade": "A",  
    "courses": ["Math", "Physics"]  
}
```

```
# Вывод исходного словаря
```

```
print("Original dictionary:", student)
```

```
# Чтение значения по ключу
```

```
name_value = student["name"]
```

```
print("Name:", name_value)
```

```
# Добавление нового элемента
```

```
student["gender"] = "Female"
```

```
# Вывод словаря после добавления нового элемента
```

```
print("Dictionary after adding 'gender':", student)
```

```
# Изменение значения по ключу
```

```
student["age"] = 21
```

```
# Вывод словаря после изменения значения
```

```
print("Dictionary after changing 'age':", student)
```

```
# Изменение значения списка в словаре
student["courses"].append("Chemistry")

# Вывод словаря после изменения значения списка
print("Dictionary after adding 'Chemistry' to courses:", student)
```

Пошаговое объяснение:

1. Создается словарь student с информацией о студенте.
2. Исходный словарь выводится на экран.
3. С использованием оператора [] и ключа "name" читается значение имени, которое сохраняется в переменной name_value.
4. Добавляется новый элемент "gender" со значением "Female" в словарь.
5. Выводится словарь после добавления нового элемента.
6. Изменяется значение ключа "age" в словаре на 21.
7. Выводится словарь после изменения значения "age".
8. С использованием оператора [] и ключа "courses", добавляется новый предмет "Chemistry" в список курсов.
9. Выводится словарь после изменения значения списка курсов.

Словари в Python позволяют удобно читать, добавлять и изменять данные по ключам, предоставляя гибкую структуру для работы с данными.

Дополнительные операции со словарями в Python.

Проверка наличия ключа (in).

Оператор in используется для проверки наличия ключа в словаре.

Получение значения с использованием метода get().

Метод get(key, default) возвращает значение по ключу. Если ключ отсутствует, возвращается значение по умолчанию (если указано), иначе возвращается None.

Удаление элемента по ключу (del или метод pop()).

Оператор del или метод pop(key) используются для удаления элемента по ключу.

Получение списка ключей и значений.

Методы keys() и values() возвращают списки ключей и значений соответственно.

Пример:

```
# Исходный словарь с информацией о студенте
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A",
    "courses": ["Math", "Physics"]
}
```

```
# Вывод исходного словаря
print("Original dictionary:", student)
```

```
# Проверка наличия ключа в словаре
is_grade_present = "grade" in student
print("Is 'grade' present?", is_grade_present)
```

```

# Получение значения по ключу с использованием метода get()
age_value = student.get("age", "Not available")
print("Age:", age_value)

# Удаление элемента по ключу с использованием оператора del
del student["courses"]

# Вывод словаря после удаления элемента
print("Dictionary after deleting 'courses':", student)

# Получение списка ключей и значений
keys_list = student.keys()
values_list = student.values()

# Вывод списков ключей и значений
print("Keys:", keys_list)
print("Values:", values_list)

```

Пошаговое объяснение:

1. Создается словарь student с информацией о студенте.
2. Исходный словарь выводится на экран.
3. Оператор in используется для проверки наличия ключа "grade" в словаре. 4. Результат сохраняется в переменной is_grade_present.
4. Метод get("age", "Not available") используется для получения значения по ключу "age". Если ключ отсутствует, возвращается значение "Not available".
5. Оператор del используется для удаления элемента "courses" из словаря.
6. Выводится словарь после удаления элемента.
7. Методы keys() и values() используются для получения списков ключей и значений соответственно.
8. Выводятся списки ключей и значений.

Циклы и словари

Циклы позволяют выполнять повторяющиеся действия. В Python, для итерации по элементам словаря, можно использовать циклы for. Словари поддерживают итерацию по ключам, значениям или парам ключ-значение.

Итерация по ключам

Цикл for key in dictionary позволяет итерироваться по ключам словаря.

Итерация по значениям

Цикл for value in dictionary.values() позволяет итерироваться по значениям словаря.

Итерация по парам ключ-значение

Цикл for key, value in dictionary.items() позволяет итерироваться по парам ключ-значение словаря.

Пример:

```

# Исходный словарь с оценками студентов
grades = {
    "Alice": 85,
    "Bob": 90,
    "Charlie": 78,
    "David": 95
}

# Итерация по ключам словаря
print("Iterating over keys:")
for student in grades:
    print(student)

# Итерация по значениям словаря
print("\nIterating over values:")
for grade in grades.values():
    print(grade)

# Итерация по парам ключ-значение словаря
print("\nIterating over key-value pairs:")
for student, grade in grades.items():
    print(f"{student}: {grade}")

```

Пошаговое объяснение:

1. Создается словарь `grades` с оценками студентов.
2. Используется цикл `for student in grades:` для итерации по ключам словаря. В каждой итерации переменная `student` принимает значение одного из ключей.
3. Выводится текущее значение переменной `student`.
4. Используется цикл `for grade in grades.values():` для итерации по значениям словаря. В каждой итерации переменная `grade` принимает значение одного из элементов словаря.
5. Выводится текущее значение переменной `grade`.
6. Используется цикл `for student, grade in grades.items():` для итерации по парам ключ-значение словаря. В каждой итерации переменные `student` и `grade` принимают значения соответствующих ключа и значения.
7. Выводится текущее значение переменных `student` и `grade`.

Практика 7. Рекурсия. Множества. Словари

Пример: Вычисление факториала числа

```

def factorial(n):
    """Рекурсивная функция для вычисления факториала числа."""
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

```



```
# Пример использования функции
number = 5
result = factorial(number)

# Вывод результата
print(f"Факториал числа {number} равен {result}.")
```

Пошаговое объяснение:

1. Функция `factorial` рекурсивно вычисляет факториал числа.
2. Если `n` равно 0 или 1, функция возвращает 1 (базовый случай).
3. В противном случае функция вызывает саму себя с аргументом `n - 1`, умножая результат на текущее значение `n`.
4. Пример использования функции с числом 5.
5. Результат вычисления факториала выводится на экран.

Множества:

Пример: Операции с множествами

```
# Создание множеств
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Объединение множеств
union_set = set1.union(set2)

# Пересечение множеств
intersection_set = set1.intersection(set2)

# Разность множеств
difference_set = set1.difference(set2)

# Вывод результатов
print(f"Объединение: {union_set}")
print(f"Пересечение: {intersection_set}")
print(f"Разность: {difference_set}")
```

Пошаговое объяснение:

1. Создаются два множества, `set1` и `set2`.
2. Используются различные операции с множествами: объединение (`union`), пересечение (`intersection`), разность (`difference`).
3. Результаты операций выводятся на экран.

Словари:

Пример: Управление словарем контактов

```
def add_contact(contacts, name, phone):
```

```

"""Добавление нового контакта в словарь."""
contacts[name] = phone

def search_contact(contacts, name):
    """Поиск контакта по имени."""
    return contacts.get(name, "Контакт не найден")

# Инициализация словаря контактов
contacts_dict = {'John': '123-456', 'Jane': '789-1011', 'Doe': '121-314'}

# Добавление нового контакта
add_contact(contacts_dict, 'Alice', '555-1234')

# Поиск контакта
result = search_contact(contacts_dict, 'Jane')

# Вывод результата
print(result)

```

Пошаговое объяснение:

1. Инициализация словаря `contacts_dict` с начальными контактами.
2. Используется функция `add_contact` для добавления нового контакта в словарь.
3. Используется функция `search_contact` для поиска контакта по имени.
4. Результат добавления и поиска выводится на экран.

Множества:

Пример: Работа с подмножествами

```

def powerset(original_set):
    """Генерация всех подмножеств множества."""
    all_subsets = []
    for i in range(2**len(original_set)):
        subset = [original_set[j] for j in range(len(original_set)) if (i & (1 << j)) > 0]
        all_subsets.append(subset)
    return all_subsets

# Пример использования функции
my_set = {1, 2, 3}
all_subsets = powerset(my_set)

# Вывод результата
print(f"Подмножества множества {my_set}: {all_subsets}")

```

Пошаговое объяснение:

1. Функция `powerset` генерирует все подмножества заданного множества.

2. Используется двоичная маска для определения, какие элементы включаются в каждое подмножество.
3. Пример использования функции для множества {1, 2, 3}.
4. Результат выводится на экран.

Словари:

Пример: Словарь студентов с оценками и вычисление среднего балла

```
def calculate_average_grade(students):
    """Вычисление среднего балла студентов."""
    total_grades = 0
    total_students = len(students)

    for student in students.values():
        total_grades += student['grade']

    average_grade = total_grades / total_students
    return average_grade

# Пример использования функции
students_data = {
    'Alice': {'grade': 90, 'age': 20},
    'Bob': {'grade': 85, 'age': 21},
    'Charlie': {'grade': 95, 'age': 19}
}

average_grade_result = calculate_average_grade(students_data)

# Вывод результата
print(f"Средний балл студентов: {average_grade_result}")
```

Пошаговое объяснение:

1. Функция `calculate_average_grade` вычисляет средний балл студентов в словаре.
2. Итерируется по значениям словаря, суммируя оценки.
3. Результат делится на общее количество студентов.
4. Пример использования функции с данными о студентах.
5. Результат выводится на экран.

Множества:

Пример: Поиск пересечения множеств в большом объеме данных

```
import random

# Генерация случайных данных
set1 = set(random.sample(range(1, 1000000), 500000))
set2 = set(random.sample(range(1, 1000000), 500000))
```

```
def intersection_large_sets(set1, set2):  
    """Нахождение пересечения больших множеств."""  
    return set1.intersection(set2)  
  
# Пример использования функции  
result_intersection = intersection_large_sets(set1, set2)  
  
# Вывод результата (так как данные случайные, результат может быть  
разным)  
print(f"Пересечение множеств: {result_intersection}")
```

Пошаговое объяснение:

1. Генерация двух больших случайных множеств set1 и set2.
2. Функция intersection_large_sets использует метод intersection для нахождения пересечения множеств.
3. Пример использования функции с большими множествами.
4. Результат выводится на экран.

Список использованных источников

1. Щербаков, А. Г., Практикум изучения языка программирования PYTHON. Начальный уровень : учебное пособие / А. Г. Щербаков. — Москва : Русайнс, 2024. — 116 с. — ISBN 978-5-466-07049-1. — URL: <https://book.ru/book/954541> (дата обращения: 24.09.2024). — Текст : электронный.
2. Чернышев, С. А., Алгоритмы и структуры данных на Python : учебное пособие / С. А. Чернышев. — Москва : КноРус, 2024. — 326 с. — ISBN 978-5-406-11683-8. — URL: <https://book.ru/book/949701> (дата обращения: 24.09.2024). — Текст : электронный.
3. Криволапов, С. Я., Математика на Python : учебник / С. Я. Криволапов, М. Б. Хрипунова. — Москва : КноРус, 2025. — 455 с. — ISBN 978-5-406-13759-8. — URL: <https://book.ru/book/955467> (дата обращения: 24.09.2024). — Текст : электронный.
4. Федоров, Д. Ю. Программирование на python : учебное пособие для вузов / Д. Ю. Федоров. — 6-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2025. — 187 с. — (Высшее образование). — ISBN 978-5-534-19666-5. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: <https://urait.ru/bcode/556864> (дата обращения: 24.09.2024).
5. Чернышев, С. А. Основы программирования на Python : учебное пособие для вузов / С. А. Чернышев. — 2-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2024. — 349 с. — (Высшее образование). — ISBN 978-5-534-17139-6. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: <https://urait.ru/bcode/544190> (дата обращения: 24.09.2024).