

ТЕМА 5. БАЗОВЫЕ КОЛЛЕКЦИИ: СТРОКИ И СПИСКИ.

5.1. Базовые коллекции.

Коллекции в программировании — это структуры данных, которые позволяют хранить и организовывать группы элементов. Они предоставляют удобные методы и операции для работы с этими элементами. В языке программирования Python базовые коллекции представляются различными типами данных, такими как:

- **Список (List):** Список представляет упорядоченную коллекцию элементов, которые могут быть различных типов данных. Он может содержать дубликаты элементов, и его можно изменять (добавлять, удалять, изменять элементы). Списки в Python обозначаются квадратными скобками `[]`.
- **Кортеж (Tuple):** Кортеж представляет упорядоченную коллекцию элементов, которые могут быть различных типов данных. В отличие от списка, кортеж является неизменяемым, то есть его элементы не могут быть изменены после создания. Кортежи в Python обозначаются круглыми скобками `()`.
- **Множество (Set):** Множество представляет неупорядоченную коллекцию уникальных элементов. Он не поддерживает индексацию элементов, и порядок элементов может меняться при итерации. Множество в Python обозначается фигурными скобками `{}` или с помощью функции `set()`.
- **Словарь (Dictionary):** Словарь представляет неупорядоченную коллекцию пар ключ-значение. Каждый элемент в словаре имеет уникальный ключ, и к нему можно обратиться по ключу для получения значения. Словари в Python обозначаются фигурными скобками `{}` и используются для хранения и доступа к данным с помощью ключей.

Эти базовые коллекции могут быть использованы для различных целей в программировании, в зависимости от требуемых операций и характеристик данных. Python также предлагает более продвинутые коллекции и структуры данных, такие как `namedtuple`, `deque`, `defaultdict`, `OrderedDict` и другие, которые могут быть использованы для решения конкретных задач.

5.2. Множества.

Множество (Set) в Python представляет собой неупорядоченную коллекцию уникальных элементов. Ключевая особенность множества состоит в том, что оно не допускает дубликатов элементов. Множество также поддерживает операции над множествами, такие как объединение, пересечение и разность.

Основные свойства и операции множеств в Python:

- **Уникальность элементов:** Множество гарантирует, что каждый элемент в нем будет уникальным. Если вы добавите элемент, который уже присутствует в множестве, он будет проигнорирован.
- **Неупорядоченность:** Элементы в множестве не имеют определенного порядка. При итерации по множеству элементы могут быть возвращены в случайном порядке.
- **Добавление элементов:** Элементы множества могут быть добавлены с помощью метода `add()`. Если элемент уже присутствует в множестве, он будет проигнорирован.
- **Удаление элементов:** Элементы множества могут быть удалены с помощью метода `remove()` или `discard()`. Метод `remove()` вызовет

исключение, если элемент не найден в множестве, в то время как метод *discard()* не вызывает исключение.

- Операции над множествами: Множества поддерживают операции объединения (*union()*), пересечения (*intersection()*), разности (*difference()*), симметрической разности (*symmetric_difference()*) и проверки подмножества (*issubset()* и *issuperset()*).

Пример использования множества в Python представлен на рисунке 5.1.



```
1 # Создание множества
2 my_set = {1, 2, 3, 4, 5}
3
4 # Добавление элемента
5 my_set.add(6)
6
7 # Удаление элемента
8 my_set.remove(3)
9
10 # Проверка наличия элемента
11 if 4 in my_set:
12     print("Элемент 4 присутствует в множестве")
13
14 # Операции над множествами
15 set1 = {1, 2, 3}
16 set2 = {3, 4, 5}
17
18 union_set = set1.union(set2) # Объединение множеств
19 intersection_set = set1.intersection(set2) # Пересечение множеств
20 difference_set = set1.difference(set2) # Разность множеств
21
22 print(union_set) # Вывод: {1, 2, 3, 4, 5}
23 print(intersection_set) # Вывод: {3}
24 print(difference_set) # Вывод: {1, 2}
25
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Элемент 4 присутствует в множестве
{1, 2, 3, 4, 5}
{3}
{1, 2}
Process finished with exit code 0
```

Рис. 5.1. Пример использования множества в Python

Множества в Python реализованы с использованием хэш-таблицы, что обеспечивает эффективность операций добавления, удаления и проверки наличия элементов. Вот как работают некоторые основные операции с множествами:

- Добавление элементов: При добавлении элемента в множество, хэш-функция вычисляет хэш-значение этого элемента. Затем хэш-значение используется для определения позиции элемента в хэш-таблице. Если в этой позиции уже есть элементы, выполняется проверка на равенство существующего элемента и нового элемента. Если элементы различаются,

создается цепочка элементов с одинаковыми хэш-значениями (так называемая коллизия). Если элемента с таким же хэш-значением еще нет, он добавляется в множество.

- Проверка наличия элементов: при проверке наличия элемента в множестве, хэш-функция сначала вычисляет хэш-значение данного элемента. Затем это хэш-значение используется для поиска элемента в хэш-таблице. Если элемент найден, происходит проверка на равенство элемента с элементом из множества. Если они совпадают, возвращается значение `'True'`, что указывает на наличие элемента в множестве.
- Удаление элементов: при удалении элемента из множества, хэш-функция вычисляет хэш-значение этого элемента. Затем это хэш-значение используется для поиска элемента в хэш-таблице. Если элемент найден, происходит проверка на равенство элемента с элементом из множества. Если они совпадают, элемент удаляется из множества.
- Операции над множествами: при выполнении операций над множествами, таких как объединение, пересечение и разность, используются хэш-таблицы обоих множеств для быстрого определения наличия и сравнения элементов. Результаты операций также сохраняются в новом множестве, чтобы гарантировать уникальность элементов.

Множества в Python обладают высокой эффективностью благодаря использованию хэш-таблиц. Они позволяют эффективно выполнять операции добавления, удаления и проверки наличия элементов, а также операции над множествами. Кроме того, множества обеспечивают уникальность элементов, что может быть полезно во многих задачах программирования.

Также в Python существует *frozenset()* - неизменяемое «замороженное» (immutable) множество.

Основное отличие *frozenset()* от обычного *set()* заключается в том, что *frozenset()* не может быть изменен после создания. Это означает, что нельзя добавлять, удалять или изменять элементы в *frozenset()*.

Основные свойства *frozenset()*:

- Неизменяемость: Элементы в *frozenset()* не могут быть изменены после его создания. Это делает *frozenset()* неизменяемым и гарантирует его неизменность во всем коде.
- Уникальность элементов: Как и обычное *set()*, *frozenset()* гарантирует, что каждый элемент в нем будет уникальным. Нельзя создать *frozenset()* с дубликатами элементов.
- Хэширование: *frozenset()* поддерживает хэширование, что позволяет использовать его в качестве ключей в словарях и элементов в других множествах.

Пример использования *frozenset()* представлен на рисунке 5.2.

```
main.py x
1 # Создание frozenset
2 my_frozenset = frozenset([1, 2, 3, 4, 5])
3
4 # Попытка изменить frozenset (вызовет ошибку)
5 # my_frozenset.add(6)
6 # my_frozenset.remove(3)
7
8 # Проверка наличия элемента в frozenset
9 if 4 in my_frozenset:
10     print("Элемент 4 присутствует в frozenset")
11
12 # Использование frozenset в качестве ключа в словаре
13 my_dict = {my_frozenset: "Значение"}
14 print(my_dict) # Вывод: {frozenset({1, 2, 3, 4, 5}): 'Значение'}
15
/Users/mihail/PycharmProjects/Lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
Элемент 4 присутствует в frozenset
{frozenset({1, 2, 3, 4, 5}): 'Значение'}

Process finished with exit code 0
```

Рис. 5.2. Пример использования неизменяемого множества в Python

Замороженные множества (`frozenset`) полезны, когда требуется использовать неизменяемую коллекцию элементов или использовать множества в качестве ключей в словарях или элементов в других множествах.

5.3. Списки.

Список (`List`) в Python представляет собой упорядоченную коллекцию элементов, которые могут быть различных типов данных. Список является изменяемым (*mutable*), что означает, что его элементы могут быть изменены после создания. Каждый элемент списка имеет свой индекс, начиная с 0 для первого элемента.

Основные свойства и операции списков в Python:

- Создание списка: Список можно создать, заключив элементы в квадратные скобки `[]` и разделив их запятыми.
Например: `my_list = [1, 2, 3, "four", 5.0]`.
- Доступ к элементам: Элементы списка можно получить с помощью индексации.
Например, `my_list[0]` вернет первый элемент списка. Индексация также может быть отрицательной, где `-1` обозначает последний элемент списка.
- Длина списка: Функция `len()` используется для определения количества элементов в списке.
Например, `len(my_list)` вернет количество элементов в `my_list`.
- Изменение элементов: Элементы списка можно изменять, присваивая новое значение по индексу.
Например, `my_list[2] = 10` изменит третий элемент списка на 10.
- Добавление элементов: Метод `append()` используется для добавления элемента в конец списка.
Например, `my_list.append(6)` добавит элемент 6 в конец списка.

- Удаление элементов: Методы `remove()` и `pop()` используются для удаления элементов из списка. `remove()` удаляет первое вхождение указанного значения, а `pop()` удаляет элемент по индексу и возвращает его значение.
- Срезы списка: С помощью срезов (slicing) можно получить подсписок из списка.
Например, `my_list[1:4]` вернет элементы с индексами от 1 до 3 (не включительно).
- Операции над списками: Списки поддерживают операции объединения (+), повторения (*), проверки наличия элемента (in), итерации и другие.

Пример использования списков в Python представлен на рисунке 5.3.

```

main.py ×
1  # Создание списка
2  my_list = [1, 2, 3, 4, 5]
3
4  # Доступ к элементам
5  print(my_list[0]) # Вывод: 1
6  print(my_list[-1]) # Вывод: 5
7
8  # Изменение элемента
9  my_list[2] = 10
10
11 # Добавление элемента
12 my_list.append(6)
13
14 # Удаление элемента
15 my_list.remove(4)
16
17 # Срез списка
18 print(my_list[1:4]) # Вывод: [2, 10, 5]
19
20 # Длина списка
21 print(len(my_list)) # Вывод: 5
22
23 # Операции над списками
24 new_list = my_list + [7, 8, 9] # Объединение списков
25 repeated_list = my_list * 3 # Повторение списка
26 print(new_list) # Вывод: [1, 2, 10, 5, 6, 7, 8, 9]
27 print(repeated_list) # Вывод: [1, 2, 10, 5, 6, 1, 2, 10, 5, 6, 1, 2, 10, 5, 6]
28
29 # Проверка наличия элемента в списке
30 if 3 in my_list:
31     print("Элемент 3 присутствует в списке")
32
33 # Итерация по списку
34 for item in my_list:
35     print(item)

```

```

/Users/mihail/PycharmProjects/Lab_1/venv/bin/python /Users/mihail/PycharmProjects/Lab_1/venv/bin/python
1
5
[2, 10, 5]
5
[1, 2, 10, 5, 6, 7, 8, 9]
[1, 2, 10, 5, 6, 1, 2, 10, 5, 6, 1, 2, 10, 5, 6]
1
2
10
5
6
Process finished with exit code 0

```

Рис. 5.2. Пример использования списков в Python

Индексация списков

В Python списки индексируются с помощью целых чисел, начиная с 0 для первого элемента списка. Индексы используются для доступа к определенным элементам списка или для выполнения операций с отдельными элементами.

Вот некоторые основные операции индексации списков:

- Получение элемента по индексу: для получения элемента списка по его индексу используется квадратные скобки с указанием индекса элемента. Например, `my_list[0]` возвращает первый элемент списка, `my_list[1]` - второй элемент и так далее.
- Отрицательная индексация: можно использовать отрицательные значения индексов для доступа к элементам списка с конца. Индекс -1 обозначает последний элемент списка, -2 обозначает предпоследний элемент и так далее. Например, `my_list[-1]` возвращает последний элемент списка, `my_list[-2]` - предпоследний элемент и так далее.
- Изменение элемента по индексу: элемент списка можно изменить, присвоив новое значение по его индексу. Например, `my_list[2] = 10` изменит третий элемент списка на 10.
- Использование срезов (slicing): с помощью срезов можно получить подсписок из списка, включая несколько элементов. Срезы задаются внутри квадратных скобок и имеют следующий формат: `[start:stop:step]`. *start* - индекс элемента, с которого начинается срез (включительно), *stop* - индекс элемента, на котором срез заканчивается (не включая его), а *step* - шаг, определяющий, какие элементы будут выбраны из среза. Если не указаны *start*, *stop* или *step*, используются значения по умолчанию. Например, `my_list[1:4]` вернет подсписок, начиная со второго элемента и заканчивая четвертым элементом.

Примеры использования индексации списков представлен на рисунке 5.3

```
main.py x
1 my_list = [1, 2, 3, 4, 5]
2
3 print(my_list[0]) # Вывод: 1
4 print(my_list[2]) # Вывод: 3
5 print(my_list[-1]) # Вывод: 5
6
7 my_list[2] = 10
8 print(my_list) # Вывод: [1, 2, 10, 4, 5]
9
10 subset = my_list[1:4]
11 print(subset) # Вывод: [2, 10, 4]
12
13 reversed_list = my_list[::-1]
14 print(reversed_list) # Вывод: [5, 4, 10, 2, 1]
15
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
1
3
5
[1, 2, 10, 4, 5]
[2, 10, 4]
[5, 4, 10, 2, 1]
Process finished with exit code 0
```

Рис. 5.3. Примеры использования индексации списков в Python

Получение части списка

Выражение `a[1:5]` используется для получения подсписка из списка `a`. Оно создает новый список, содержащий элементы с индексами от 1 до 4 (не включая 5). Этот процесс называется срезом (*slicing*) списка.

Общий синтаксис для среза списка в Python: `a[start:stop:step]`.

- *start* - индекс элемента, с которого начинается срез (включительно).
- *stop* - индекс элемента, на котором срез заканчивается (не включая его).
- *step* - шаг, определяющий, какие элементы будут выбраны из среза. По умолчанию шаг равен 1.

В контексте выражения `a[1:5]`, *start* равно 1, *stop* равно 5, а *step* равно 1 (по умолчанию). Это означает, что подсписок будет состоять из элементов с индексами 1, 2, 3 и 4.

Пример получения подсписка из списка представлен на рисунке 5.4.

В этом примере *subset* будет содержать элементы списка `a` с индексами от 1 до 4.

Обратите внимание, что индексы в Python начинаются с 0. Поэтому элемент с индексом 1 - это второй элемент в списке. `a[1:5]` возвращает подсписок без первого элемента. Если вам нужно включить первый элемент, вы можете использовать `a[0:5]`.

```
main.py ×
1 a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 subset = a[1:5]
3 print(subset) # Вывод: [1, 2, 3, 4]
4
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
[1, 2, 3, 4]

Process finished with exit code 0
```

Рис. 5.4. Пример получения подсписка из списка

Методы создания списков

В Python существует несколько способов создания списков. Перечислим некоторые из них:

- Литерал списка: можно создать список, перечислив его элементы в квадратных скобках [] и разделив их запятыми (рисунок 5.5.)

```
main.py ×
1 my_list = [1, 2, 3, 4, 5]
2
```

Рис. 5.5. Литерал списка

- Функция *list()*: можно создать список, используя функцию *list()*, которая принимает итерируемый объект в качестве аргумента и создает список из его элементов. (рисунок 5.6.)

```
4 my_list1 = list(range(1, 6))
```

Рис. 5.6. Функция *list()*

- Списковое включение (List comprehension): Списковое включение - это способ создания списка на основе другого списка или другого итерируемого объекта с помощью компактного синтаксиса. Например, создание списка квадратов чисел от 1 до 5 (рисунок 5.7.)

```
7 my_list2 = [x ** 2 for x in range(1, 6)]
```

Рис. 5.7. Списковое включение

- Метод *append()*: можно создать пустой список и затем добавлять элементы в него с помощью метода *append()* (рисунок 5.8.)

```
10 my_list3 = []
11 my_list3.append(1)
12 my_list3.append(2)
13 my_list3.append(3)
```

Рис. 5.8. Метод *append()*

- Генераторы списков (List comprehensions): генераторы списков - это более компактный способ создания списков с помощью выражения и итерации. Они позволяют создавать списки на основе условий и преобразований

элементов. Например, создание списка четных чисел от 1 до 10 (рисунок 5.9.)

```
6 my_list4 = [x for x in range(1, 11) if x % 2 == 0]
```

Рис. 5.9. Генераторы списков

- Метод *split()*: Метод *split()* может быть использован для разделения строки на список элементов на основе разделителя (рисунок 5.10.)

```
9 my_string = "apple, banana, cherry"
10 my_list = my_string.split(", ")
```

Рис. 5.10. Метод *split()*

Обращения к элементам

В Python для обращения к элементам списка используется индексация. Индексы начинаются с 0 для первого элемента списка и идут последовательно до N-1, где N - количество элементов в списке.

Рассмотрим примеры обращения к спискам

- Получение элемента по индексу: используйте квадратные скобки [] и указывайте индекс элемента, который вы хотите получить (рисунок 5.11.)

```
main.py x
1 my_list = [10, 20, 30, 40, 50]
2 first_element = my_list[0] # Получение первого элемента
3 third_element = my_list[2] # Получение третьего элемента
4 last_element = my_list[-1] # Получение последнего элемента
```

Рис. 5.11. Получение элемента по индексу

- Изменение значения элемента: Вы можете изменить значение элемента списка, обратившись к нему по индексу и присвоив новое значение (рисунок 5.12.)

```
main.py x
1 my_list = [10, 20, 30, 40, 50]
2 my_list[1] = 25 # Изменение второго элемента на 25
```

Рис. 5.12. Изменение значения элемента

- Обращение к под спискам (срезам): с помощью срезов (*slicing*) можно получить подсписок из исходного списка. Срезы задаются с помощью двоеточия : внутри квадратных скобок [] (рисунок 5.13.)

```
main.py x
1 my_list = [10, 20, 30, 40, 50]
2 subset = my_list[1:4] # Получение подсписка с индексами от 1 до 3
```

Рис. 5.13. Обращение к под спискам

- Проверка наличия элемента в списке: с помощью оператора *in* можно проверить, содержится ли определенный элемент в списке. (рисунок 5.14.)

```
main.py x
1 my_list = [10, 20, 30, 40, 50]
2 if 30 in my_list:
3     print("Элемент 30 присутствует в списке")
4
```

Рис. 5.14. Проверка наличия элемента в списке

Обратите внимание, что попытка обратиться к несуществующему индексу или выход за пределы списка вызовет ошибку *IndexError*. Убедитесь, что индексы, которые вы используете, находятся в допустимом диапазоне.

Разложение списков

Разложение списков (*Unpacking*) в Python - это процесс присвоения элементов списка переменным. При разложении списка количество переменных должно быть равно количеству элементов в списке.

Рассмотрим пример разложения списка (рисунок 5.15.)

В этом примере список *my_list* содержит 3 элемента. При разложении списка мы указываем 3 переменные *a*, *b* и *c*, и значения из списка автоматически присваиваются этим переменным.

```
main.py x
1 my_list = [10, 20, 30]
2 a, b, c = my_list # Разложение списка
3
4 print(a) # Вывод: 10
5 print(b) # Вывод: 20
6 print(c) # Вывод: 30
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
10
20
30
Process finished with exit code 0
```

Рис. 5.15. Пример разложения списка

Разложение списков также может быть использовано с помощью оператора *** для присваивания нескольким переменным одновременно. Это позволяет присвоить одной переменной оставшиеся элементы списка после присваивания остальным переменным (рисунок 5.16.)

В этом примере переменная *a* получает первый элемент списка, а переменная *rest* получает оставшиеся элементы, упакованные в новый список.

```
main.py x
1 my_list = [10, 20, 30, 40, 50]
2 a, *rest = my_list # Разложение списка с использованием оператора *
3
4 print(a) # Вывод: 10
5 print(rest) # Вывод: [20, 30, 40, 50]
6
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
10
[20, 30, 40, 50]
Process finished with exit code 0
```

Рис. 5.16. Пример разложения списка с помощью оператора *

Разложение списков также может быть использовано для обмена значениями между переменными. (рисунок 5.17.)

В этом примере значения переменных *a* и *b* меняются местами с помощью разложения списков.

```
main.py x
1 a = 10
2 b = 20
3 a, b = b, a # Обмен значениями
4
5 print(a) # Вывод: 20
6 print(b) # Вывод: 10
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
20
10
Process finished with exit code 0
```

Рис. 5.17. Пример разложение для обмена значениями между переменными.

Разложение списков в Python является мощным инструментом для работы с элементами списков и упрощает присваивание значений переменным из списков.

Перебор списков через циклы

Для перебора элементов списка в Python можно использовать различные циклы, такие как цикл *for* и цикл *while*.

Пример использования цикла *for* для перебора элементов представлен на рисунке 5.18.

В этом примере каждый элемент списка *my_list* присваивается переменной *item*, и затем этот элемент выводится на экран. Результатом будет вывод каждого элемента списка по порядку

```
main.py ×
1 my_list = [1, 2, 3, 4, 5]
2
3 for item in my_list:
4     print(item)
5

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
1
2
3
4
5

Process finished with exit code 0
```

Рис. 5.18. Пример использования цикла *for* для перебора элементов

Также можно использовать индексы элементов списка вместе с циклом *for*, если вам требуется не только значение элемента, но и его индекс (Рисунок 5.19.)

В этом примере функция *enumerate()* используется для получения пары значений (индекс, элемент) из списка *my_list*. Затем пары значений разбираются в переменные *index* и *item*, и выводится информация об индексе и значении элемента.

```
main.py ×
1 my_list = [1, 2, 3, 4, 5]
2
3 for index, item in enumerate(my_list):
4     print(f"Индекс: {index}, Значение: {item}")
5

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
Индекс: 0, Значение: 1
Индекс: 1, Значение: 2
Индекс: 2, Значение: 3
Индекс: 3, Значение: 4
Индекс: 4, Значение: 5

Process finished with exit code 0
```

Рис. 5.19. Пример использования индексов элемента списка

Если вы предпочитаете использовать цикл *while*, вы можете использовать индексы и длину списка для перебора элементов (Рисунок 5.20.)

В этом примере цикл *while* выполняется до тех пор, пока индекс не станет равным длине списка *len(my_list)*. Внутри цикла выводится элемент списка по текущему индексу, а затем индекс увеличивается на 1.

```
main.py ×
1 my_list = [1, 2, 3, 4, 5]
2 index = 0
3
4 while index < len(my_list):
5     print(my_list[index])
6     index += 1
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/L
1
2
3
4
5

Process finished with exit code 0
```

Рис. 5.20. Пример использования индексов элемента списка

Оба подхода - использование цикла *for* и цикла *while* - позволяют перебирать элементы списка и выполнять операции с ними. Выбор конкретного цикла зависит от вашего предпочтения и требований вашей программы.

Сравнение списков

Для сравнения списков в Python можно использовать операторы сравнения, такие как `==`, `!=`, `<`, `>`, `<=` и `>=`. Они позволяют сравнивать списки на равенство, неравенство и отношение (Рисунок 5.21.)

```
main.py ×
1 list1 = [1, 2, 3]
2 list2 = [1, 2, 3]
3 list3 = [4, 5, 6]
4
5 # Сравнение на равенство
6 print(list1 == list2) # Вывод: True
7
8 # Сравнение на неравенство
9 print(list1 != list3) # Вывод: True
10
11 # Сравнение на отношение (поэлементно)
12 print(list1 < list3) # Вывод: True
13 print(list1 > list2) # Вывод: False
14
15 # Сравнение на отношение (лексикографически)
16 list4 = [1, 2, 3, 4]
17 list5 = [1, 2, 3, 5]
18 print(list4 < list5) # Вывод: True
19
```

```

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/1
True
True
True
False
True

Process finished with exit code 0

```

Рис. 5.21. Пример сравнения списка

В примере выше *list1* и *list2* содержат одни и те же элементы, поэтому их сравнение с помощью оператора `==` дает результат *True*. Сравнение *list1* и *list3* дает результат *False*, так как они содержат разные элементы.

Операторы `<` и `>` сравнивают списки поэлементно. Они сравнивают элементы списков попарно, начиная с первых элементов. Если все попарные сравнения элементов дадут *True*, то результат будет *True*. Если же хотя бы одно сравнение будет *False*, результат будет *False*. В примере выше `list1 < list3` дает результат *True*, так как первый элемент *list1* (1) меньше первого элемента *list3* (4).

Операторы сравнения списков также могут использоваться для сортировки списков и определения их порядка при сортировке.

Обратите внимание, что для корректного сравнения списков они должны быть одного типа (например, оба списка должны содержать только числа или только строки). Сравнение списков разных типов может дать непредсказуемые результаты.

Функции списков

В Python списки обладают множеством встроенных функций и методов, которые облегчают работу с ними и позволяют выполнять различные операции. Вот некоторые из наиболее часто используемых функций и методов списков:

- `len(list)`: возвращает количество элементов в списке (Рисунок 5.22.)

```

main.py x
1 my_list = [1, 2, 3, 4, 5]
2 length = len(my_list) # Длина списка: 5
3

```

Рис. 5.22. Функция `len(list)`

- `sum(list)`: возвращает сумму всех элементов в числовом списке (Рисунок 5.23.)

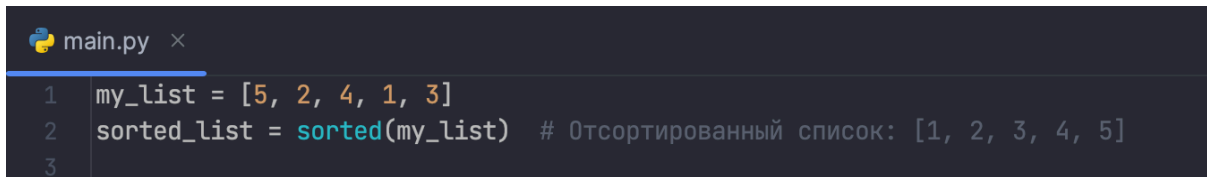
```

main.py x
1 my_list = [1, 2, 3, 4, 5]
2 total = sum(my_list) # Сумма элементов списка: 15
3

```

Рис. 5.23. Функция `sum(list)`

- `sorted(list)`: возвращает новый отсортированный список (Рисунок 5.24.)



```

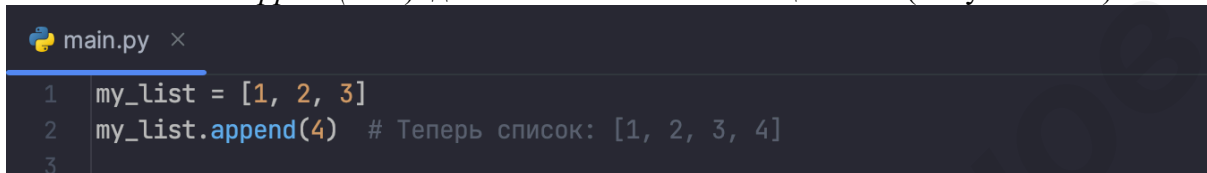
main.py x
1 my_list = [5, 2, 4, 1, 3]
2 sorted_list = sorted(my_list) # Отсортированный список: [1, 2, 3, 4, 5]
3

```

Рис. 5.24. Функция *sorted(list)*:

Методы списков:

- *list.append(item)*: добавляет элемент в конец списка (Рисунок 5.25.)



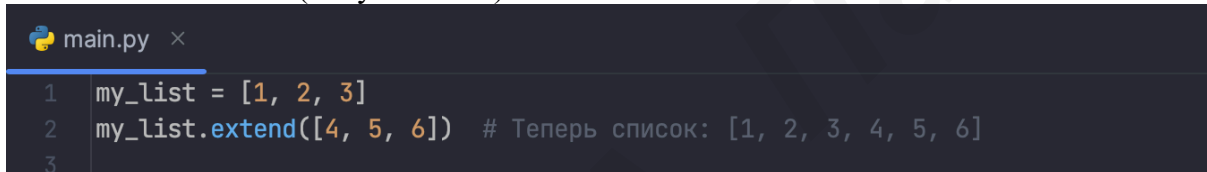
```

main.py x
1 my_list = [1, 2, 3]
2 my_list.append(4) # Теперь список: [1, 2, 3, 4]
3

```

Рис. 5.25. Метод *list.append(item)*

- *list.extend(iterable)*: добавляет элементы из итерируемого объекта в конец списка (Рисунок 5.26.)



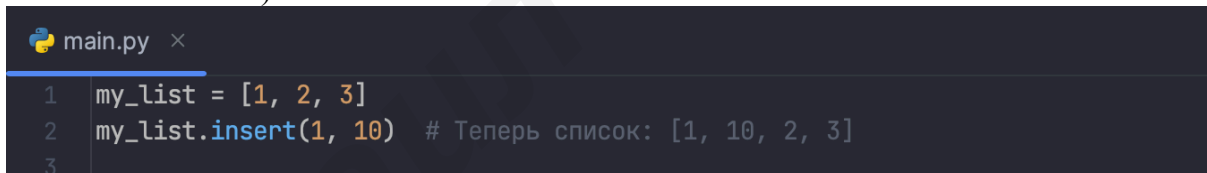
```

main.py x
1 my_list = [1, 2, 3]
2 my_list.extend([4, 5, 6]) # Теперь список: [1, 2, 3, 4, 5, 6]
3

```

Рис. 5.26. Метод *list.extend(iterable)*

- *list.insert(index, item)*: вставляет элемент по указанному индексу (Рисунок 5.27.)



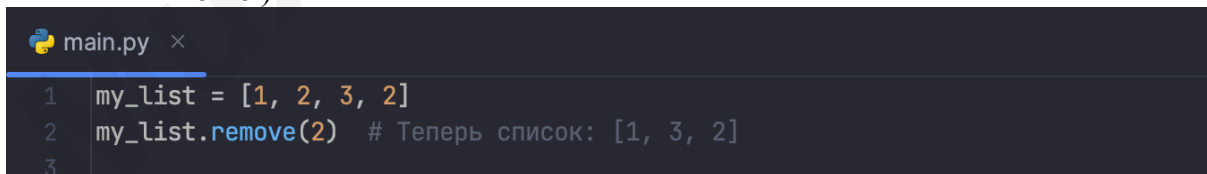
```

main.py x
1 my_list = [1, 2, 3]
2 my_list.insert(1, 10) # Теперь список: [1, 10, 2, 3]
3

```

Рис. 5.27. Метод *list.insert(index, item)*

- *list.remove(item)*: удаляет первое вхождение элемента из списка (Рисунок 5.28.)



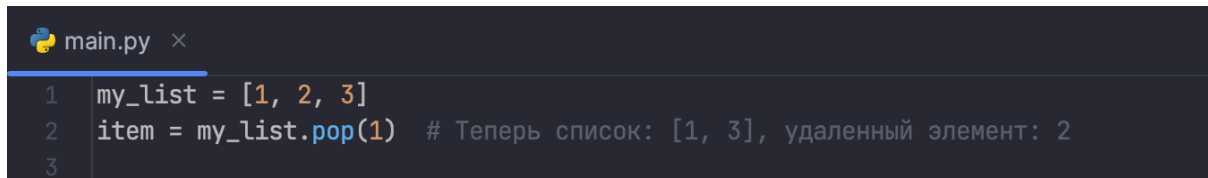
```

main.py x
1 my_list = [1, 2, 3, 2]
2 my_list.remove(2) # Теперь список: [1, 3, 2]
3

```

Рис. 5.28. Метод *list.remove(item)*

- *list.pop(index)*: удаляет и возвращает элемент по указанному индексу. Если индекс не указан, удаляется и возвращается последний элемент (Рисунок 5.29.)



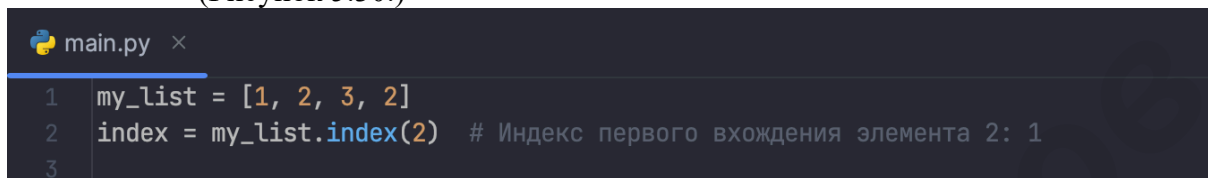
```

1 my_list = [1, 2, 3]
2 item = my_list.pop(1) # Теперь список: [1, 3], удаленный элемент: 2
3

```

Рис. 5.29. Метод *list.pop(index)*

- *list.index(item)*: возвращает индекс первого вхождения элемента в списке (Рисунок 5.30.)



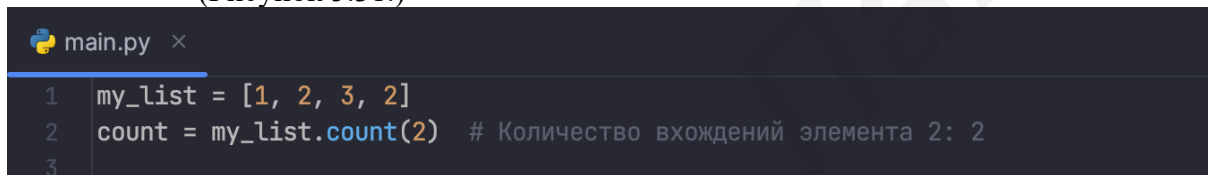
```

1 my_list = [1, 2, 3, 2]
2 index = my_list.index(2) # Индекс первого вхождения элемента 2: 1
3

```

Рис. 5.30. Метод *list.index(item)*

- *list.count(item)*: возвращает количество вхождений элемента в списке (Рисунок 5.31.)



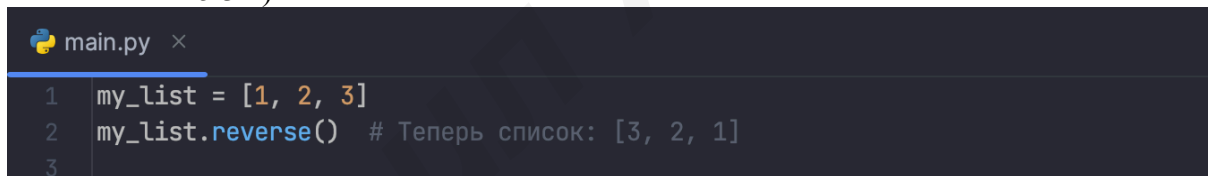
```

1 my_list = [1, 2, 3, 2]
2 count = my_list.count(2) # Количество вхождений элемента 2: 2
3

```

Рис. 5.31. Метод *list.count(item)*

- *list.reverse()*: изменяет порядок элементов в списке на обратный (Рисунок 5.32.)



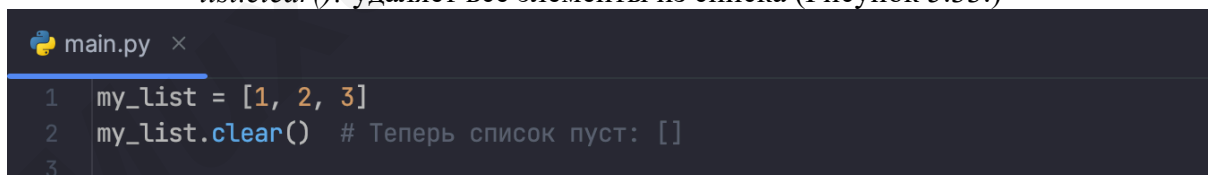
```

1 my_list = [1, 2, 3]
2 my_list.reverse() # Теперь список: [3, 2, 1]
3

```

Рис. 5.32 Метод *list.reverse()*

- *list.clear()*: удаляет все элементы из списка (Рисунок 5.33.)



```

1 my_list = [1, 2, 3]
2 my_list.clear() # Теперь список пуст: []
3

```

Рис. 5.33. Метод *list.clear()*

Сортировка списков

В Python существует несколько способов сортировки списков. Вот некоторые из них:

- Метод *sort()*: метод *sort()* сортирует список на месте, изменяя порядок элементов в исходном списке. Сортировка выполняется в возрастающем порядке (от наименьшего к наибольшему) для чисел или в лексикографическом порядке для строк (Рисунок 5.34.)


```
main.py ×
1 my_list = [3, 1, 4, 2, 5]
2 my_list.sort()
3 print(my_list) # Вывод: [1, 2, 3, 4, 5]
4

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
[1, 2, 3, 4, 5]

Process finished with exit code 0
```

Рис. 5.34. Метод `sort()`

- Встроенная функция `sorted()`: функция `sorted()` создает новый отсортированный список на основе исходного списка, не изменяя его. Отсортированный список возвращается в результате вызова функции (Рисунок 5.35.)

```
main.py ×
1 my_list = [3, 1, 4, 2, 5]
2 sorted_list = sorted(my_list)
3 print(sorted_list) # Вывод: [1, 2, 3, 4, 5]
4

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
[1, 2, 3, 4, 5]

Process finished with exit code 0
```

Рис. 5.35. Встроенная функция `sorted()`

- Параметр `key` в функции сортировки: параметр `key` позволяет указать функцию, которая будет применяться к каждому элементу списка для получения значения, по которому будет производиться сортировка (Рисунок 5.36.)

```
main.py ×
1 my_list = ["apple", "banana", "cherry"]
2 sorted_list = sorted(my_list, key=len) # Сортировка по длине строк
3 print(sorted_list) # Вывод: ["apple", "cherry", "banana"]
4

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
['apple', 'banana', 'cherry']

Process finished with exit code 0
```

Рис. 5.36. Параметр `key` в функции сортировки

- Параметр `reverse` в функции сортировки: параметр `reverse` позволяет указать, следует ли выполнять сортировку в обратном порядке (по убыванию) (Рисунок 5.37.)

```
main.py ×
1 my_list = [5, 2, 4, 1, 3]
2 my_list.sort(reverse=True) # Сортировка в обратном порядке
3 print(my_list) # Вывод: [5, 4, 3, 2, 1]
4
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
[5, 4, 3, 2, 1]

Process finished with exit code 0
```

Рис. 5.37. Параметр *reverse* в функции сортировки

- Метод *sorted()* с оператором сравнения (lambda-функция): можно использовать метод *sorted()* в сочетании с лямбда-функцией для определения кастомной функции сортировки (Рисунок 5.38.)

```
main.py ×
1 my_list = ["apple", "banana", "cherry"]
2 sorted_list = sorted(my_list, key=lambda x: x[1]) # Сортировка по второму сим
3 print(sorted_list) # Вывод: ["banana", "apple", "cherry"]
4
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
['banana', 'cherry', 'apple']

Process finished with exit code 0
```

Рис. 5.38. Метод *sorted()*

Это лишь некоторые из способов сортировки списков в Python. Выбор конкретного способа зависит от ваших требований и предпочтений.

Минимальное и максимальное значения в списке

В Python для нахождения минимального и максимального значения в списке можно использовать функции *min()* и *max()* соответственно. Обе функции принимают в качестве аргумента список и возвращают соответствующее значение.

Пример использования функций *min()* и *max()* (Рисунок 5.39.)

```
main.py ×
1 my_list = [5, 2, 4, 1, 3]
2
3 min_value = min(my_list) # Наименьшее значение в списке
4 max_value = max(my_list) # Наибольшее значение в списке
5
6 print(min_value) # Вывод: 1
7 print(max_value) # Вывод: 5
8
```

```

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/venv/bin/python
1
5
Process finished with exit code 0

```

Рис. 5.39. Пример использования функций *min()* и *max()*:

В этом примере функция *min()* возвращает наименьшее значение в списке *my_list*, а функция *max()* возвращает наибольшее значение. Оба значения затем выводятся на экран.

Обратите внимание, что функции *min()* и *max()* могут использоваться не только для списков чисел, но и для списков строк и других типов данных, поддерживающих сравнение.

Если список содержит объекты более сложного типа, например, словари или пользовательские классы, можно использовать параметр *key* в функциях *min()* и *max()* для указания функции, которая будет применяться к каждому элементу списка для определения значения, по которому будет производиться сравнение.

Пример использования параметра *key* для нахождения минимального и максимального значения в списке словарей по определенному ключу (Рисунок 5.40.)

```

main.py x
1 my_list = [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}, {'name': 'Charlie', 'age': 20}]
2
3 min_value = min(my_list, key=lambda x: x['age']) # Наименьший словарь по ключу 'age'
4 max_value = max(my_list, key=lambda x: x['age']) # Наибольший словарь по ключу 'age'
5
6 print(min_value) # Вывод: {'name': 'Charlie', 'age': 20}
7 print(max_value) # Вывод: {'name': 'Bob', 'age': 30}
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/venv/bin/python
{'name': 'Charlie', 'age': 20}
{'name': 'Bob', 'age': 30}
Process finished with exit code 0

```

Рис. 5.40. Пример использования параметра *key*

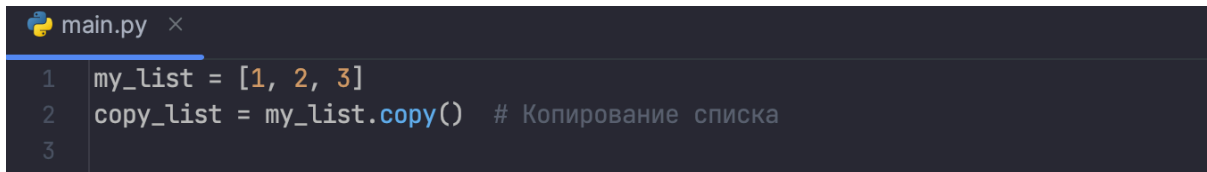
В этом примере мы используем лямбда-функцию, чтобы указать ключ *'age'* для сравнения словарей в списке. Функция *min()* находит словарь с наименьшим значением по ключу *'age'*, а функция *max()* находит словарь с наибольшим значением по ключу *'age'*. Оба словаря затем выводятся на экран.

Копирование списков

В Python для копирования и объединения списков можно использовать различные методы и операторы.

Копирование списка:

- Метод *copy()*: метод *copy()* создает и возвращает копию списка (Рисунок 5.41.)



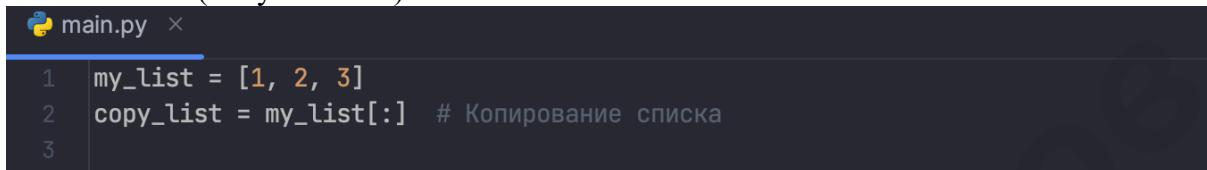
```

main.py x
1 my_list = [1, 2, 3]
2 copy_list = my_list.copy() # Копирование списка
3

```

Рис. 5.41. Метод *copy()*

- Оператор `[:]`: можно использовать срез `[:]` для создания копии списка (Рисунок 5.42.)



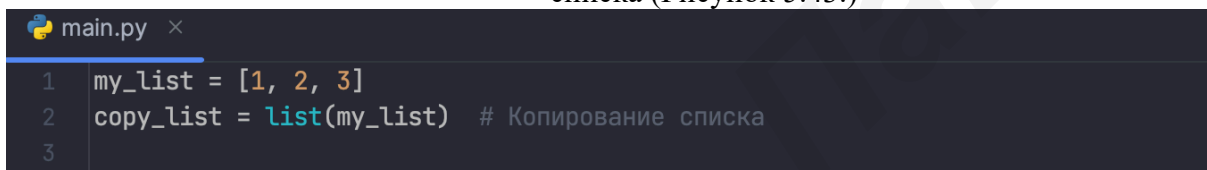
```

main.py x
1 my_list = [1, 2, 3]
2 copy_list = my_list[:] # Копирование списка
3

```

Рис. 5.42. Оператор `[:]`

- Функция *list()*: можно использовать функцию *list()* для создания копии списка (Рисунок 5.43.)



```

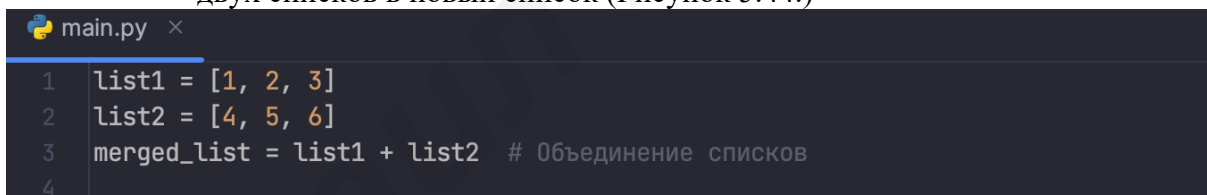
main.py x
1 my_list = [1, 2, 3]
2 copy_list = list(my_list) # Копирование списка
3

```

Рис. 5.43. Функция *list()*

Объединение (конкатенация) списков

- Оператор `+`: оператор `+` используется для объединения (конкатенации) двух списков в новый список (Рисунок 5.44.)



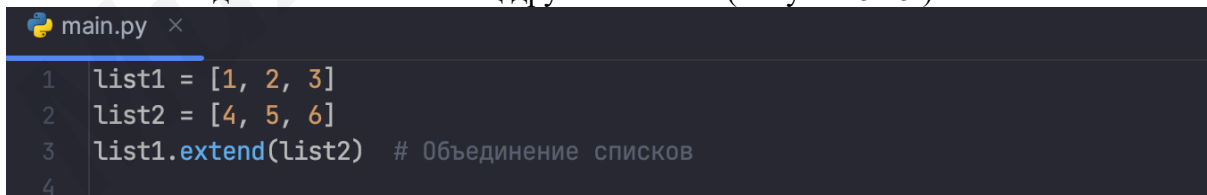
```

main.py x
1 list1 = [1, 2, 3]
2 list2 = [4, 5, 6]
3 merged_list = list1 + list2 # Объединение списков
4

```

Рис. 5.44. Оператор `+`

- Метод *extend()*: метод *extend()* используется для добавления элементов из одного списка в конец другого списка (Рисунок 5.45.)



```

main.py x
1 list1 = [1, 2, 3]
2 list2 = [4, 5, 6]
3 list1.extend(list2) # Объединение списков
4

```

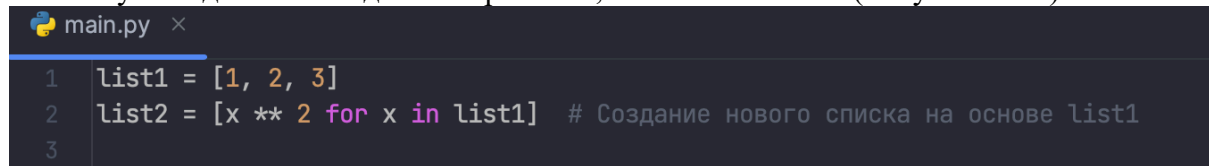
Рис. 5.45. Метод *extend()*

Создание нового списка на основе двух или более списков:

Списковое включение (List comprehension): Списковое включение позволяет создать новый список на основе существующих списков и применить к ним операции или фильтры.

Обратите внимание, что при копировании списка обычным присваиванием (`copy_list = my_list`) создается ссылка на существующий список, а не его копия. Если изменить

копию, изменения могут повлиять и на оригинальный список. Чтобы избежать этого, используйте один из методов копирования, описанных выше (Рисунок 5.46.)



```
main.py ×
1 list1 = [1, 2, 3]
2 list2 = [x ** 2 for x in list1] # Создание нового списка на основе list1
3
```

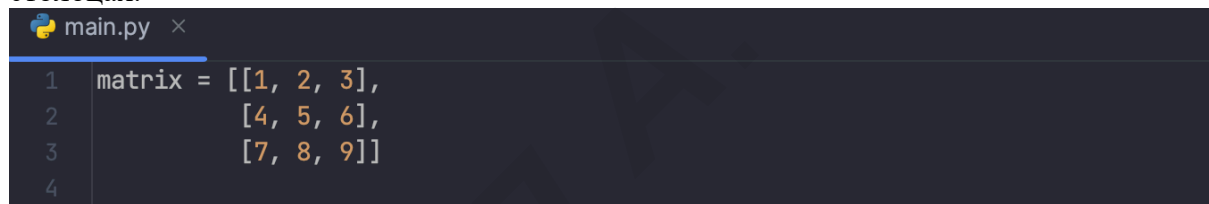
Рис. 5.46. Метод *extend()*

Также учтите, что объединение списков оператором `+` или методом `extend()` изменяет один из исходных списков или создает новый список. Если вам нужно сохранить исходные списки неизменными, создайте новый список на основе их значений.

Списки списков

В Python списки списков (также известные как вложенные списки) представляют собой структуру данных, в которой элементы списка являются также списками. Такая структура позволяет хранить и организовывать данные в виде двумерной матрицы или таблицы.

Пример создания списка списков представлен на рисунке 5.47. В этом примере *matrix* - это список списков. Каждый внутренний список представляет строку матрицы, а элементы внутренних списков представляют значения в соответствующих строках и столбцах.

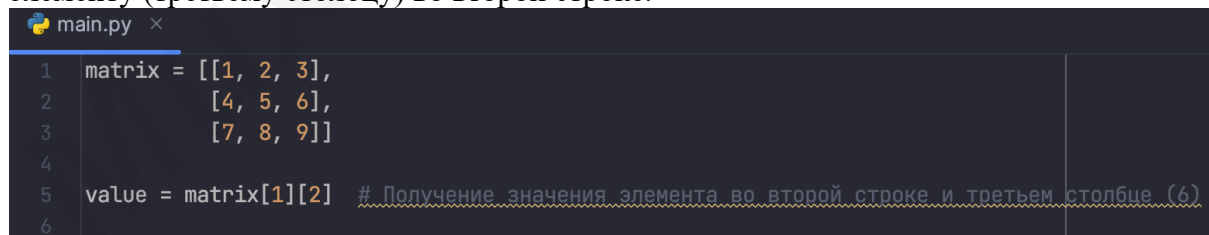


```
main.py ×
1 matrix = [[1, 2, 3],
2           [4, 5, 6],
3           [7, 8, 9]]
4
```

Рис. 5.47. Пример создания списка списков

Для доступа к элементам списка списков используется двойная индексация как показано на рисунке 5.48.

Первый индекс обозначает строку, а второй - столбец. В этом примере *matrix[1]* обращается к второй строке списка списков, а *matrix[1][2]* обращается к третьему элементу (третьему столбцу) во второй строке.

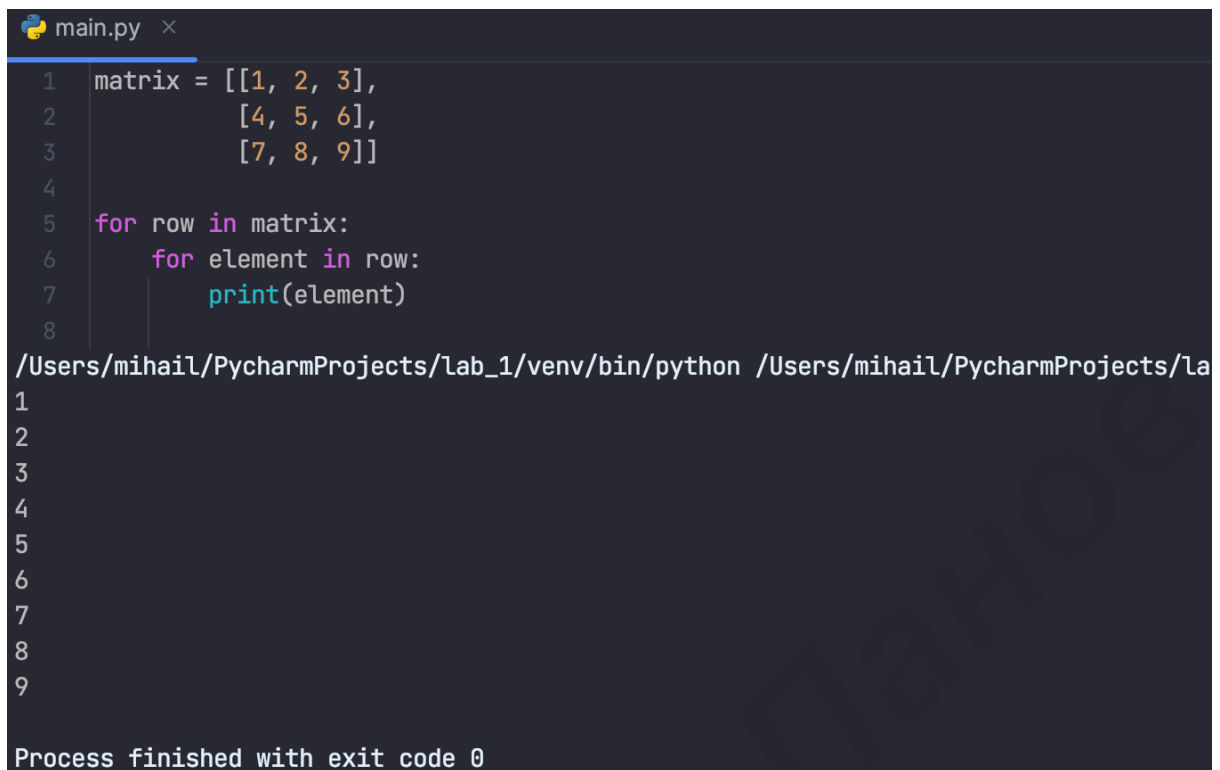


```
main.py ×
1 matrix = [[1, 2, 3],
2           [4, 5, 6],
3           [7, 8, 9]]
4
5 value = matrix[1][2] # Получение значения элемента во второй строке и третьем столбце (6)
6
```

Рис. 5.48. Пример доступа к элементам списка списков через двойную индексацию

Также можно использовать циклы для перебора элементов списка списков. Например, чтобы вывести все значения матрицы (Рисунок 5.49)

В этом примере внешний цикл перебирает строки (внутренние списки) матрицы, а внутренний цикл перебирает элементы в каждой строке и выводит их на экран.



```
main.py x
1 matrix = [[1, 2, 3],
2           [4, 5, 6],
3           [7, 8, 9]]
4
5 for row in matrix:
6     for element in row:
7         print(element)
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/venv/bin/python
1
2
3
4
5
6
7
8
9
Process finished with exit code 0
```

Рис. 5.49. Пример использования цикла для перебора элементов списка списков

Списки списков в Python предоставляют удобный способ хранения и обработки двумерных данных, таких как матрицы, таблицы или сетки. Используя индексацию и циклы, можно легко работать с элементами вложенных списков.

Списки в Python широко используются для хранения и управления коллекциями элементов. Они обладают гибкостью и позволяют изменять содержимое списка в процессе выполнения программы. Списки позволяют выполнять различные операции, такие как добавление, удаление, изменение, доступ к элементам и другие.