[Списки, кортежи и словари 1](#_Toc154481717)

[Список 1](#_Toc154481718)

[Кортежи 15](#_Toc154481719)

[Диапазоны 18](#_Toc154481720)

[Словари 19](#_Toc154481721)

[Множества 26](#_Toc154481722)

[List comprehension 29](#_Toc154481723)

[Упаковка и распаковка 31](#_Toc154481724)

[Упаковка и распаковка в параметрах функций 35](#_Toc154481725)

**Списки, кортежи и словари**

**Список**

Для работы с наборами данных Python предоставляет такие встроенные типы как списки, кортежи и словари.

Список (list) представляет тип данных, который хранит набор или последовательность элементов. Во многих языках программирования есть аналогичная структура данных, которая называется массив.

### Создание списка

Для создания списка применяются квадратные скобки **[]**, внутри которых через запятую перечисляются элементы списка. Например, определим список чисел:

|  |  |
| --- | --- |
| 1 | numbers = [1, 2, 3, 4, 5] |

Подобным образом можно определять списки с данными других типов, например, определим список строк:

|  |  |
| --- | --- |
| 1 | people = ["Tom", "Sam", "Bob"] |

Также для создания списка можно использовать функцию-конструктор **list()**:

|  |  |
| --- | --- |
| 1  2 | numbers1 = []  numbers2 = list() |

Оба этих определения списка аналогичны - они создают пустой список.

Список необязательно должен содержать только однотипные объекты. Мы можем поместить в один и тот же список одновременно строки, числа, объекты других типов данных:

|  |  |
| --- | --- |
| 1 | objects = [1, 2.6, "Hello", True] |

Для проверки элементов списка можно использовать стандартную функцию print, которая выводит содержимое списка в удобочитаемом виде:

|  |  |
| --- | --- |
| 1  2  3  4  5 | numbers = [1, 2, 3, 4, 5]  people = ["Tom", "Sam", "Bob"]    print(numbers)  # [1, 2, 3, 4, 5]  print(people)   # ["Tom", "Sam", "Bob"] |

Конструктор list может принимать набор значений, на основе которых создается список:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers1 = [1, 2, 3, 4, 5]  numbers2 = list(numbers1)  print(numbers2)  # [1, 2, 3, 4, 5]    letters = list("Hello")  print(letters)      # ['H', 'e', 'l', 'l', 'o'] |

Если необходимо создать список, в котором повторяется одно и то же значение несколько раз, то можно использовать символ звездочки \*, то есть фактически применить операцию умножения к уже существующему списку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | numbers = [5] \* 6   # 6 раз повторяем 5  print(numbers)      # [5, 5, 5, 5, 5, 5]    people = ["Tom"] \* 3    # 3 раза повторяем "Tom"  print(people)           # ["Tom", "Tom", "Tom"]    students = ["Bob", "Sam"] \* 2   # 2 раза повторяем "Bob", "Sam"  print(students)                 # ["Bob", "Sam", "Bob", "Sam"] |

### Обращение к элементам списка

Для обращения к элементам списка надо использовать индексы, которые представляют номер элемента в списка. Индексы начинаются с нуля. То есть первый элемент будет иметь индекс 0, второй элемент - индекс 1 и так далее. Для обращения к элементам с конца можно использовать отрицательные индексы, начиная с -1. То есть у последнего элемента будет индекс -1, у предпоследнего - -2 и так далее.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | people = ["Tom", "Sam", "Bob"]  # получение элементов с начала списка  print(people[0])   # Tom  print(people[1])   # Sam  print(people[2])   # Bob    # получение элементов с конца списка  print(people[-2])   # Sam  print(people[-1])   # Bob  print(people[-3])   # Tom |

Для изменения элемента списка достаточно присвоить ему новое значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | people = ["Tom", "Sam", "Bob"]    people[1] = "Mike"  # изменение второго элемента  print(people[1])    # Mike  print(people)       # ["Tom", "Mike", "Bob"] |

### Разложение списка

Python позволяет разложить список на отдельные элементы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | people = ["Tom", "Bob", "Sam"]    tom, bob, sam = people    print(tom)      # Tom  print(bob)      # Bob  print(sam)      # Sam |

В данном случае переменным tom, bob и sam последовательно присваиваются элементы из списка people. Однако следует учитывать, что количество переменных должно быть равно числу элементов присваиваемого списка.

### Перебор элементов

Для перебора элементов можно использовать как цикл for, так и цикл while.

Перебор с помощью цикла **for**:

|  |  |
| --- | --- |
| 1  2  3 | people = ["Tom", "Sam", "Bob"]  for person in people:      print(person) |

Здесь будет производиться перебор списка people, и каждый его элемент будет помещаться в переменную person.

Перебор также можно сделать с помощью цикла **while**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | people = ["Tom", "Sam", "Bob"]  i = 0  while i < len(people):      print(people[i])    # применяем индекс для получения элемента      i += 1 |

Для перебора с помощью функции **len()** получаем длину списка. С помощью счетчика i выводит по элементу, пока значение счетчика не станет равно длине списка.

### Сравнение списков

Два списка считаются равными, если они содержат один и тот же набор элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers1 = [1, 2, 3, 4, 5]  numbers2 = list([1, 2, 3, 4, 5])  if numbers1 == numbers2:      print("numbers1 equal to numbers2")  else:      print("numbers1 is not equal to numbers2") |

В данном случае оба списка будут равны.

### Получение части списка

Если необходимо получить какую-то определенную часть списка, то мы можем применять специальный синтаксис, который может принимать следующие формы:

* list[:end]: через параметр end передается индекс элемента, до которого нужно копировать список
* list[start:end]: параметр start указывает на индекс элемента, начиная с которого надо скопировать элементы
* list[start:end:step]: параметр step указывает на шаг, через который будут копироваться элементы из списка. По умолчанию этот параметр равен 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]    slice\_people1 = people[:3]   # с 0 по 3  print(slice\_people1)   # ["Tom", "Bob", "Alice"]    slice\_people2 = people[1:3]   # с 1 по 3  print(slice\_people2)   # ["Bob", "Alice"]    slice\_people3 = people[1:6:2]   # с 1 по 6 с шагом 2  print(slice\_people3)   # ["Bob", "Sam", "Bill"] |

Можно использовать отрицательные индексы, тогда отсчет будет идти с конца, например, -1 - предпоследний, -2 - третий сконца и так далее.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]    slice\_people1 = people[:-1]   # с предпоследнего по нулевой  print(slice\_people1)   # ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]    slice\_people2 = people[-3:-1]   # с третьего с конца по предпоследний  print(slice\_people2)   # [ "Sam", "Tim"] |

### Методы и функции по работе со списками

Для управления элементами списки имеют целый ряд методов. Некоторые из них:

* **append(item)**: добавляет элемент item в конец списка
* **insert(index, item)**: добавляет элемент item в список по индексу index
* **extend(items)**: добавляет набор элементов items в конец списка
* **remove(item)**: удаляет элемент item. Удаляется только первое вхождение элемента. Если элемент не найден, генерирует исключение ValueError
* **clear()**: удаление всех элементов из списка
* **index(item)**: возвращает индекс элемента item. Если элемент не найден, генерирует исключение ValueError
* **pop([index])**: удаляет и возвращает элемент по индексу index. Если индекс не передан, то просто удаляет последний элемент.
* **count(item)**: возвращает количество вхождений элемента item в список
* **sort([key])**: сортирует элементы. По умолчанию сортирует по возрастанию. Но с помощью параметра key мы можем передать функцию сортировки.
* **reverse()**: расставляет все элементы в списке в обратном порядке
* **copy()**: копирует список

Кроме того, Python предоставляет ряд встроенных функций для работы со списками:

* **len(list)**: возвращает длину списка
* **sorted(list, [key])**: возвращает отсортированный список
* **min(list)**: возвращает наименьший элемент списка
* **max(list)**: возвращает наибольший элемент списка

### Добавление и удаление элементов

Для добавления элемента применяются методы append(), extend и insert, а для удаления - методы remove(), pop() и clear().

Использование методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | people = ["Tom", "Bob"]    # добавляем в конец списка  people.append("Alice")  # ["Tom", "Bob", "Alice"]  # добавляем на вторую позицию  people.insert(1, "Bill")  # ["Tom", "Bill", "Bob", "Alice"]  # добавляем набор элементов ["Mike", "Sam"]  people.extend(["Mike", "Sam"])      # ["Tom", "Bill", "Bob", "Alice", "Mike", "Sam"]  # получаем индекс элемента  index\_of\_tom = people.index("Tom")  # удаляем по этому индексу  removed\_item = people.pop(index\_of\_tom)     # ["Bill", "Bob", "Alice", "Mike", "Sam"]  # удаляем последний элемент  last\_item = people.pop()     # ["Bill", "Bob", "Alice", "Mike"]  # удаляем элемент "Alice"  people.remove("Alice")      # ["Bill", "Bob", "Mike"]  print(people)       # ["Bill", "Bob", "Mike"]  # удаляем все элементы  people.clear()  print(people)       # [] |

### Проверка наличия элемента

Если определенный элемент не найден, то методы remove и index генерируют исключение. Чтобы избежать подобной ситуации, перед операцией с элементом можно проверять его наличие с помощью ключевого слова **in**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | people = ["Tom", "Bob", "Alice", "Sam"]    if "Alice" in people:      people.remove("Alice")  print(people)       # ["Tom", "Bob", "Sam"] |

Выражение if "Alice" in people возвращает True, если элемент "Alice" имеется в списке people. Поэтому конструкция if "Alice" in people может выполнить последующий блок инструкций в зависимости от наличия элемента в списке.

### Удаление с помощью del

Python также поддерживает еще один способ удаления элементов списка - с помощью оператора **del**. В качестве параметра этому оператору передается удаляемый элемент или набор элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | people = ["Tom", "Bob", "Alice", "Sam", "Bill", "Kate", "Mike"]    del people[1]   # удаляем второй элемент  print(people)   # ["Tom", "Alice", "Sam", "Bill", "Kate", "Mike"]  del people[:3]   # удаляем  по четвертый элемент не включая  print(people)   # ["Bill", "Kate", "Mike"]  del people[1:]   # удаляем  со второго элемента  print(people)   # ["Bill"] |

### Изменение подсписка

Для изменения подсписка - набора элементов в списке можно использовать вышерассмотренный синтаксис [start:end]:

|  |  |
| --- | --- |
| 1  2  3 | nums = [10, 20, 30, 40, 50]  nums[1:4]=[11, 22]  print(nums)     # [10, 11, 22, 50] |

Здесь выражение nums[1:4] фактически обращается к подсписку [20, 30, 40]. Присвоение этому подсписку списка [11, 22] позволяет заменить элемента с 1 по 4 индекс не включая на элементы [11, 22]. И после изменения получим список [10, 11, 22, 50]

### Подсчет вхождений

Если необходимо узнать, сколько раз в списке присутствует тот или иной элемент, то можно применить метод count():

|  |  |
| --- | --- |
| 1  2  3  4 | people = ["Tom", "Bob", "Alice", "Tom", "Bill", "Tom"]    people\_count = people.count("Tom")  print(people\_count)      # 3 |

### Сортировка

Для сортировки по возрастанию применяется метод **sort()**:

|  |  |
| --- | --- |
| 1  2  3  4 | people = ["Tom", "Bob", "Alice", "Sam", "Bill"]    people.sort()  print(people)      # ["Alice", "Bill", "Bob", "Sam", "Tom"] |

Если необходимо отсортировать данные в обратном порядке, то мы можем после сортировки применить метод reverse():

|  |  |
| --- | --- |
| 1  2  3  4  5 | people = ["Tom", "Bob", "Alice", "Sam", "Bill"]    people.sort()  people.reverse()  print(people)      # ["Tom", "Sam", "Bob", "Bill", "Alice"] |

При сортировке фактически сравниваются два объекта, и который из них "меньше", ставится перед тем, который "больше". Понятия "больше" и "меньше" довольно условны. И если для чисел все просто - числа расставляются в порядке возрастания, то для строк и других объектов ситуация сложнее. В частности, строки оцениваются по первым символам. Если первые символы равны, оцениваются вторые символы и так далее. При чем цифровой символ считается "меньше", чем алфавитный заглавный символ, а заглавный символ считается меньше, чем строчный.

Таким образом, если в списке сочетаются строки с верхним и нижним регистром, то мы можем получить не совсем корректные результаты, так как для нас строка "bob" должна стоять до строки "Tom". И чтобы изменить стандартное поведение сортировки, мы можем передать в метод sort() в качестве параметра функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | people = ["Tom", "bob", "alice", "Sam", "Bill"]    people.sort()       # стандартная сортировка  print(people)      # ["Bill", "Sam", "Tom", "alice", "bob"]    people.sort(key=str.lower)  # сортировка без учета регистра  print(people)      # ["alice", "Bill", "bob", "Sam", "Tom"] |

Кроме метода sort мы можем использовать встроенную функцию **sorted**, которая имеет две формы:

* sorted(list): сортирует список list
* sorted(list, key): сортирует список list, применяя к элементам функцию key

|  |  |
| --- | --- |
| 1  2  3  4 | people = ["Tom", "bob", "alice", "Sam", "Bill"]    sorted\_people = sorted(people, key=str.lower)  print(sorted\_people)      # ["alice", "Bill", "bob", "Sam", "Tom"] |

При использовании этой функции следует учитывать, что эта функция не изменяет сортируемый список, а все отсортированные элементы она помещает в новый список, который возвращается в качестве результата.

### Минимальное и максимальное значения

Встроенный функции Python min() и max() позволяют найти минимальное и максимальное значения соответственно:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [9, 21, 12, 1, 3, 15, 18]  print(min(numbers))     # 1  print(max(numbers))     # 21 |

### Копирование списков

При копировании списков следует учитывать, что списки представляют изменяемый (mutable) тип, поэтому если обе переменных будут указывать на один и тот же список, то изменение одной переменной, затронет и другую переменную:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | people1 = ["Tom", "Bob", "Alice"]  people2 = people1  people2.append("Sam")   # добавляем элемент во второй список  # people1 и people2 указывают на один и тот же список  print(people1)   # ["Tom", "Bob", "Alice", "Sam"]  print(people2)   # ["Tom", "Bob", "Alice", "Sam"] |

Это так называемое "поверхностное копирование" (shallow copy). И, как правило, такое поведение нежелательное. И чтобы происходило копирование элементов, но при этом переменные указывали на разные списки, необходимо выполнить глубокое копирование (deep copy). Для этого можно использовать метод **copy()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | people1 = ["Tom", "Bob", "Alice"]  people2 = people1.copy()    # копируем элементы из people1 в people2  people2.append("Sam")   # добавляем элемент ТОЛЬКО во второй список  # people1 и people2 указывают на разные списки  print(people1)   # ["Tom", "Bob", "Alice"]  print(people2)   # ["Tom", "Bob", "Alice", "Sam"] |

### Соединение списков

Для объединения списков применяется операция сложения (+):

|  |  |
| --- | --- |
| 1  2  3  4 | people1 = ["Tom", "Bob", "Alice"]  people2 = ["Tom", "Sam", "Tim", "Bill"]  people3 = people1 + people2  print(people3)   # ["Tom", "Bob", "Alice", "Tom", "Sam", "Tim", "Bill"] |

### Списки списков

Списки кроме стандартных данных типа строк, чисел, также могут содержать другие списки. Подобные списки можно ассоциировать с таблицами, где вложенные списки выполняют роль строк. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | people = [      ["Tom", 29],      ["Alice", 33],      ["Bob", 27]  ]    print(people[0])         # ["Tom", 29]  print(people[0][0])      # Tom  print(people[0][1])      # 29 |

Чтобы обратиться к элементу вложенного списка, необходимо использовать пару индексов: people[0][1] - обращение ко второму элементу первого вложенного списка.

Добавление, удаление и изменение общего списка, а также вложенных списков аналогично тому, как это делается с обычными (одномерными) списками:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | people = [      ["Tom", 29],      ["Alice", 33],      ["Bob", 27]  ]    # создание вложенного списка  person = list()  person.append("Bill")  person.append(41)  # добавление вложенного списка  people.append(person)    print(people[-1])         # ["Bill", 41]    # добавление во вложенный список  people[-1].append("+79876543210")    print(people[-1])         # ["Bill", 41, "+79876543210"]    # удаление последнего элемента из вложенного списка  people[-1].pop()  print(people[-1])         # ["Bill", 41]    # удаление всего последнего вложенного списка  people.pop(-1)    # изменение первого элемента  people[0] = ["Sam", 18]  print(people)            # [ ["Sam", 18], ["Alice", 33], ["Bob", 27]] |

Перебор вложенных списков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | people = [      ["Tom", 29],      ["Alice", 33],      ["Bob", 27]  ]    for person in people:      for item in person:          print(item, end=" | ") |

Консольный вывод:

Tom | 29 | Alice | 33 | Bob | 27 |

**Вопрос 1**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | languages = ["Java", "Python", "Javascript", "C++", "C#"]  for lang in languages[:]:      languages.remove(lang)    print(languages) |

Варианты ответов:

1. ["Java", "Python", "Javascript", "C++", "C#"]
2. []
3. Будет сгенерировано исключение
4. Ничего из выше перечисленного

Ответ

Правильный вариант: [] (список без элементов)

Выражение languages[:] возвращает все элементы списка languages. В цикле перебираем все эти элементы и удаляем из списка languages.

**Вопрос 2**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | try:      letters=["a","b"]      print(letters[2])  except (ValueError, IndexError):      print("Произошла ошибка") |

Варианты ответов:

1. Index Error
2. Syntax Error
3. 2
4. Произошла ошибка

Ответ

Правильный вариант: Произошла ошибка. Поскольку список letters имеет только два элемента, то при обращении по индексу 2 программа сгенерирует ошибку IndexError, которая будет перехвачена в выражении except. Соответственно будет выполняться выражение print("Произошла ошибка")

**Вопрос 3**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [1, 2, 3, 4]  for idx, item in enumerate(numbers): del item  print(numbers) |

Варианты ответов:

1. [1, 3]
2. []
3. [2, 4]
4. [1, 2, 3, 4]
5. Ничего из выше перечисленного

Ответ

Правильный вариант: [1, 2, 3, 4]

Функция enumerate(numbers) возвращает перебираемый объект, где каждый элемент можно представить в виде кортежа, который состоит из элемента из списка numbers и его индекса:

|  |  |
| --- | --- |
| 1 | [(0, 1), (1, 2), (2, 3), (3, 4)] |

Далее этот объект перебирается и раскладывается на две переменных: idx (индекс) и item (само число из списка numbers)

|  |  |
| --- | --- |
| 1 | for idx, item in enumerate(numbers) |

В цикле удаляем объект item:

|  |  |
| --- | --- |
| 1 | del item |

Это выражение затронет значение в объекте, который возвращается из функции enumerate(numbers), однако изначальный список numbers останется неизменным.

**Вопрос 4**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3 | li = ["A"]  li.extend("BCD")  print(li) |

Варианты ответов:

1. ["A", "BCD"]
2. ["A", "B", "C", "D"]
3. ["ABCD"]
4. Программа сгенерирует ошибку

Ответ

Правильный вариант: ["A", "B", "C", "D"]

Метод extend() добавляет в конец списка набор элементов, который передается в качестве параметра. Метод extend() рассматривает переданный в него параметр как набор объектов, поэтому в данном случае в реальности мы могли бы написать li.extend(["B","C","D"]). И в итоге элементы списка ["B","C","D"] были бы добавлены в конец списка ["A"]

**Вопрос 5**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def func(x, y=None):      if y is None:          y = [1]      y.extend(x)      return y    print(func([0], [2, 4, 5])) |

Варианты ответов:

1. [2, 4, 5, [0]]
2. [1, 2, 4, 5, [0]]
3. [0, 2, 4, 5]
4. [0, 1, 2, 4, 5]
5. [2, 4, 5, 0]
6. Программа сгенерирует ошибку

Ответ

Правильный вариант: [2, 4, 5, 0]

Здесь функция добавляет в конец списка y элементы из списка x. Поскольку параметр y здесь определен - это список [2, 4, 5], то вызов y.extend(x) добавляет в конец этого списка элементы из списка [0].

**Вопрос 6**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [1, 2, 3, 4, 5]  print(2.0 in numbers, end=" ")  print("2" in numbers) |

Варианты ответов:

1. True True
2. False False
3. True False
4. False True
5. Программа сгенерирует ошибку

Ответ

Правильный вариант: True False

Оператор **in** возвращает True, если объект имеется в некоторой коллекции объектов. Если данные представляют разные типы данные, то применяются неявные преобразования. Если один элемент операции представляет float, а другой int, то значение типа int приводится к типу float. Поэтому для числа 3.0 возвратится True. В случае со строкой "2" неявное преобразование невозможно, поэтому возвращается False.

**Вопрос 7**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2 | numbers = [1, 2, 3, 4, 5]  print(numbers[:4].pop( )) |

Варианты ответов:

1. [1, 2, 3, 4]
2. 5
3. [1, 2, 3, 5]
4. 4
5. 1

Ответ

Правильный вариант: 4

Выражение **numbers[:4]** выбирает первые 4 элемента списка, а метод pop() извлекает последний из них.

**Вопрос 8**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3 | items = [5, "t", 5.6, 3, "Python"]  print(3.0 in items, end=" ")  print(bool(items[-2:-4])) |

Варианты ответов:

1. True True
2. False False
3. True False
4. False True

Ответ

Правильный вариант: True False

Выражение 3.0 in items возвращает True, потому что в списке items есть число 3. А выражение bool(items[-2:-4]) возвращает False, потому что выражение items[-2:-4] возвращает пустой список [], а пустой список соответствует значению False..

**Вопрос 9**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2 | numbers = [5, 8, 12, 15, 5, 18, 11]  print(numbers[:].count(5)) |

Варианты ответов:

1. 0
2. 1
3. 2
4. 3
5. 5
6. 7

Ответ

Правильный вариант: 2

Выражение [start:end] выбирает из списка подсписок с индекса start по индекс end. Если один из параметров (или оба параметра) не определены, то по умолчанию для start равно 0, а end представляет конечный индекс списка. Соответственно выражение [:] выбирает все элементы списка. Затем метод .count(5) вычисляет количество значений "5" в полученном списке. Поскольку в полученном списке есть два числа со значением 5, то возвращается число 2.

**Вопрос 10**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2 | li = ["in", "as", "if", "be"]  print(li[1:][:2]) |

Варианты ответов:

1. "in", "as", "if", "be"
2. "as", "if"
3. "in", "as"
4. "in", "as", "if"
5. "if", "be"
6. "as", "if", "be"

Ответ

Правильный вариант: ["as", "if"]

Выражение li[1:] выбирает из списка подсписок с индекса 1. То есть мы получим список ["as", "if", "be"]

Затем из полученного списка выбираем все элементы по 2-й индекс не включая: li[1:][:2].

**Вопрос 11**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2  3 | li = [10, 20, 30, 40]  li[1:3]=[50]  print(li) |

Варианты ответов:

1. [10, 50, 30, 40]
2. [10, 50, 50, 40]
3. [50, 50]
4. [10, 50, 40]
5. [50]

Ответ

Правильный вариант: [10, 50, 40]

Выражение li[1:3] выбирает подсписок [20, 30]

Затем этому подсписку присваивается список [50]. То есть в исходном списке мы заменяем подсписок [20, 30] на [50].

**Вопрос 12**

Что выведет на консоль следующая программа:

|  |  |
| --- | --- |
| 1  2 | arr = []  print(arr \* 2) |

Варианты ответов:

1. []
2. [][]
3. None
4. Будет ошибка

Ответ

Когда к списку применяется оператор умножения \*, то он создает новый список с несколькими копиями элементов исходного списка. Когда вы умножаете список на число, элементы в списке повторяются столько раз. В примере выше, поскольку умножаемый список пуст, результатом будет пустой список [].

Если бы список содержал элементы, например, если arr равен [2, 3], то результатом выражения print(arr\*2) был бы список [2, 3, 2, 3].

## Кортежи

Кортеж (tuple) представляет последовательность элементов, которая во многом похожа на список за тем исключением, что кортеж является неизменяемым (immutable) типом. Поэтому мы не можем добавлять или удалять элементы в кортеже, изменять его.

Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделенные запятыми:

|  |  |
| --- | --- |
| 1  2 | tom = ("Tom", 23)  print(tom)     # ("Tom", 23) |

Также для определения кортежа мы можем просто перечислить значения через запятую без применения скобок:

|  |  |
| --- | --- |
| 1  2 | tom = "Tom", 23  print(tom)     # ("Tom", 23) |

Если вдруг кортеж состоит из одного элемента, то после единственного элемента кортежа необходимо поставить запятую:

|  |  |
| --- | --- |
| 1 | tom = ("Tom",) |

Для создания кортежа из другого набора элементов, например, из списка, можно передать список в функцию **tuple()**, которая возвратит кортеж:

|  |  |
| --- | --- |
| 1  2  3 | data = ["Tom", 37, "Google"]  tom = tuple(data)  print(tom)      # ("Tom", 37, "Google") |

С помощью встроенной функции **len()** можно получить длину кортежа:

|  |  |
| --- | --- |
| 1  2 | tom = ("Tom", 37, "Google")  print(len(tom))     # 3 |

### Обращение к элементам кортежа

Обращение к элементам в кортеже происходит также, как и в списке, по индексу. Индексация начинается также с нуля при получении элементов с начала списка и с -1 при получении элементов с конца списка:

|  |  |
| --- | --- |
| 1  2  3  4 | tom = ("Tom", 37, "Google", "software developer")  print(tom[0])       # Tom  print(tom[1])       # 37  print(tom[-1])      # software developer |

Но так как кортеж - неизменяемый тип (immutable), то мы не сможем изменить его элементы. То есть следующая запись работать не будет:

|  |  |
| --- | --- |
| 1 | tom[1] = "Tim" |

При необходимости мы можем разложить кортеж на отдельные переменные:

|  |  |
| --- | --- |
| 1  2  3  4  5 | name, age, company, position = ("Tom", 37, "Google", "software developer")  print(name)         # Tom  print(age)          # 37  print(position)     # software developer  print(company)     # Google |

### Получение подкортежей

Как и в списках, можно получить часть кортежа в виде другого кортежа

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | tom = ("Tom", 37, "Google", "software developer")    # получем подкортеж с 1 по 3 элемента (не включая)  print(tom[1:3])     # (37, "Google")    # получем подкортеж с 0 по 3 элемента (не включая)  print(tom[:3])     # ("Tom", 37, "Google")    # получем подкортеж с 1 по послдений элемент  print(tom[1:])     # (37, "Google", "software developer") |

### Кортеж как параметр и результат функций

Особенно удобно использовать кортежи, когда необходимо возвратить из функции сразу несколько значений. Когда функция возвращает несколько значений, фактически она возвращает в кортеж:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def get\_user():      name = "Tom"      age = 22      company = "Google"      return name, age, company      user = get\_user()  print(user)     # ("Tom", 37, "Google") |

При передаче кортежа в функцию с помощью оператора \* его можно разложить на отдельные значения, которые передаются параметрам функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def print\_person(name, age, company):      print(f"Name: {name}  Age: {age}  Company: {company}")    tom = ("Tom", 22)  print\_person(\*tom, "Microsoft")     # Name: Tom  Age: 22  Company: Microsoft    bob = ("Bob", 41, "Apple")  print\_person(\*bob)      # Name: Bob  Age: 41  Company: Apple |

### Перебор кортежей

Для перебора кортежа можно использовать стандартные циклы **for** и **while**. С помощью цикла for:

|  |  |
| --- | --- |
| 1  2  3 | tom = ("Tom", 22, "Google")  for item in tom:      print(item) |

С помощью цикла while:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | tom = ("Tom", 22, "Google")    i = 0  while i < len(tom):      print(tom[i])      i += 1 |

### Проверка наличия значения

Как для списка с помощью выражения элемент in кортеж можно проверить наличие элемента в кортеже:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | user = ("Tom", 22, "Google")  name = "Tom"  if name in user:      print("Пользователя зовут Tom")  else:      print("Пользователь имеет другое имя") |

## Диапазоны

Диапазоны или range представляют неизменяемый последовательный набор чисел. Для создания диапазов применяетя **range**, которая имеет следующие формы:

* **range(stop)**: возвращает все целые числа от 0 до stop
* **range(start, stop)**: возвращает все целые числа в промежутке от start (включая) до stop (не включая).
* **range(start, stop, step)**: возвращает целые числа в промежутке от start (включая) до stop (не включая), которые увеличиваются на значение step

Примеры вызовов функции range:

|  |  |
| --- | --- |
| 1  2  3  4 | range(5)            # 0, 1, 2, 3, 4  range(1, 5)         # 1, 2, 3, 4  range(2, 10, 2)     # 2, 4, 6, 8  range(10, 2, -2)    # 10 8 6 4 |

Диапазоны чаще всего применяются в циклах **for**. Например, выведем последовательно все числа от 0 до 4:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for i in range(5):      print(i, end=" ")    # Консольный вывод  # 0, 1, 2, 3, 4 |

Другой пример выведем таблицу умножения:

|  |  |
| --- | --- |
| 1  2  3  4 | for i in range(1, 10):      for j in range(1, 10):          print(i \* j, end="\t")      print("\n") |

1 2 3 4 5 6 7 8 9

2 4 6 8 10 12 14 16 18

3 6 9 12 15 18 21 24 27

4 8 12 16 20 24 28 32 36

5 10 15 20 25 30 35 40 45

6 12 18 24 30 36 42 48 54

7 14 21 28 35 42 49 56 63

8 16 24 32 40 48 56 64 72

9 18 27 36 45 54 63 72 81

Если нам необходим последовательный список чисел, то для его создания удобно использовать функцию **range**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = list(range(10))  print(numbers)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(2, 10))  print(numbers)      # [2, 3, 4, 5, 6, 7, 8, 9]  numbers = list(range(10, 2, -2))  print(numbers)      # [10, 8, 6, 4] |

Преимуществом диапазонов над стандартными списками и кортежами является то, что диапазон всегда будет занимать одно и то же небольшое количество памяти вне зависимости от того, какой набор чисел представляет этот диапазон. В действительности диапазон хранит только начальное, конечное значение и приращение.

## Словари

**Словарь** (dictionary) в языке Python хранит коллекцию элементов, где каждый элемент имеет уникальный ключ и ассоциированое с ним некоторое значение.

Определение словаря имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | dictionary = { ключ1:значение1, ключ2:значение2, ....} |

В фигурных скобках через запятую определяется последовательность элементов, где для каждого элемента сначала указывается ключ и через двоеточие его значение.

Определим словарь:

|  |  |
| --- | --- |
| 1 | users = {1: "Tom", 2: "Bob", 3: "Bill"} |

В словаре users в качестве ключей используются числа, а в качестве значений - строки. То есть элемент с ключом 1 имеет значение "Tom", элемент с ключом 2 - значение "Bob" и т.д.

Другой пример:

|  |  |
| --- | --- |
| 1 | emails = {"tom@gmail.com": "Tom", "bob@gmai.com": "Bob", "sam@gmail.com": "Sam"} |

В словаре emails в качестве ключей используются строки - электронные адреса пользователей и в качестве значений тоже строки - имена пользователей.

Но необязательно ключи и строки должны быть однотипными. Они могу представлять разные типы:

|  |  |
| --- | --- |
| 1 | objects = {1: "Tom", "2": True, 3: 100.6} |

Мы можем также вообще определить пустой словарь без элементов:

|  |  |
| --- | --- |
| 1 | objects = {} |

или так:

|  |  |
| --- | --- |
| 1 | objects = dict() |

### Преобразование списков и кортежей в словарь

Несмотря на то, что словарь и список - непохожие по структуре типы, но тем не менее существует возможности для отдельных видов списков преобразования их в словарь с помощью встроенной функции **dict()**. Для этого список должен хранить набор вложенных списков. Каждый вложенный список должен состоять из двух элементов - при конвертации в словарь первый элемент станет ключом, а второй - значением:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users\_list = [      ["+111123455", "Tom"],      ["+384767557", "Bob"],      ["+958758767", "Alice"]  ]  users\_dict = dict(users\_list)  print(users\_dict)      # {"+111123455": "Tom", "+384767557": "Bob", "+958758767": "Alice"} |

Подобным образом можно преобразовать в словарь двухмерные кортежи, которые в свою очередь содержать кортежи из двух элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users\_tuple = (      ("+111123455", "Tom"),      ("+384767557", "Bob"),      ("+958758767", "Alice")  )  users\_dict = dict(users\_tuple)  print(users\_dict) |

### Получение и изменение элементов

Для обращения к элементам словаря после его названия в квадратных скобках указывается ключ элемента:

|  |  |
| --- | --- |
| 1 | dictionary[ключ] |

Например, получим и изменим элементы в словаре:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    # получаем элемент с ключом "+11111111"  print(users["+11111111"])      # Tom    # установка значения элемента с ключом "+33333333"  users["+33333333"] = "Bob Smith"  print(users["+33333333"])   # Bob Smith |

Если при установки значения элемента с таким ключом в словаре не окажется, то произойдет его добавление:

|  |  |
| --- | --- |
| 1 | users["+4444444"] = "Sam" |

Но если мы попробуем получить значение с ключом, которого нет в словаре, то Python сгенерирует ошибку KeyError:

|  |  |
| --- | --- |
| 1 | user = users["+4444444"]    # KeyError |

И чтобы предупредить эту ситуацию перед обращением к элементу мы можем проверять наличие ключа в словаре с помощью выражения **ключ in словарь**. Если ключ имеется в словаре, то данное выражение возвращает True:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | key = "+4444444"  if key in users:      user = users[key]      print(user)  else:      print("Элемент не найден") |

Также для получения элементов можно использовать метод **get**, который имеет две формы:

* get(key): возвращает из словаря элемент с ключом key. Если элемента с таким ключом нет, то возвращает значение None
* get(key, default): возвращает из словаря элемент с ключом key. Если элемента с таким ключом нет, то возвращает значение по умолчанию default

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    user1 = users.get("+55555555")  print(user1)    # Alice  user2 = users.get("+33333333", "Unknown user")  print(user2)    # Bob  user3 = users.get("+44444444", "Unknown user")  print(user3)    # Unknown user |

### Удаление

Для удаления элемента по ключу применяется оператор **del**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    del users["+55555555"]  print(users)    # { "+11111111": "Tom", "+33333333": "Bob"} |

Но стоит учитывать, что если подобного ключа не окажется в словаре, то будет выброшено исключение KeyError. Поэтому опять же перед удалением желательно проверять наличие элемента с данным ключом.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }    key = "+55555555"  if key in users:      del users[key]      print(f"Элемент с ключом {key} удален")  else:      print("Элемент не найден") |

Другой способ удаления представляет метод **pop()**. Он имеет две формы:

* pop(key): удаляет элемент по ключу key и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то генерируется исключение KeyError
* pop(key, default): удаляет элемент по ключу key и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то возвращается значение default

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }  key = "+55555555"  user = users.pop(key)  print(user)     # Alice    user = users.pop("+4444444", "Unknown user")  print(user)     # Unknown user |

Если необходимо удалить все элементы, то в этом случае можно воспользоваться методом **clear():**

|  |  |
| --- | --- |
| 1 | users.clear() |

### Копирование и объединение словарей

Метод **copy()** копирует содержимое словаря, возвращая новый словарь:

|  |  |
| --- | --- |
| 1  2  3 | users = {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice"}  students = users.copy()  print(students)     # {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice"} |

Метод **update()** объединяет два словаря:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {"+1111111": "Tom", "+3333333": "Bob"}    users2 = {"+2222222": "Sam", "+6666666": "Kate"}  users.update(users2)    print(users)    # {"+1111111": "Tom", "+3333333": "Bob", "+2222222": "Sam", "+6666666": "Kate"}  print(users2)   # {"+2222222": "Sam", "+6666666": "Kate"} |

При этом словарь users2 остается без изменений. Изменяется только словарь users, в который добавляются элементы другого словаря. Но если необходимо, чтобы оба исходных словаря были без изменений, а результатом объединения был какой-то третий словарь, то можно предварительно скопировать один словарь в другой:

|  |  |
| --- | --- |
| 1  2 | users3 = users.copy()  users3.update(users2) |

### Перебор словаря

Для перебора словаря можно воспользоваться циклом for:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }  for key in users:      print(f"Phone: {key}  User: {users[key]} ") |

При переборе элементов мы получаем ключ текущего элемента и по нему можем получить сам элемент.

Другой способ перебора элементов представляет использование метода **items()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {      "+11111111": "Tom",      "+33333333": "Bob",      "+55555555": "Alice"  }  for key, value in users.items():      print(f"Phone: {key}  User: {value} ") |

Метод items() возвращает набор кортежей. Каждый кортеж содержит ключ и значение элемента, которые при переборе мы тут же можем получить в переменные key и value.

Также существуют отдельно возможности перебора ключей и перебора значений. Для перебора ключей мы можем вызвать у словаря метод **keys()**:

|  |  |
| --- | --- |
| 1  2 | for key in users.keys():      print(key) |

Правда, этот способ перебора не имеет смысла, так как и без вызова метода keys() мы можем перебрать ключи, как было показано выше.

Для перебора только значений мы можем вызвать у словаря метод **values()**:

|  |  |
| --- | --- |
| 1  2 | for value in users.values():      print(value) |

### Комплексные словари

Кроме простейших объектов типа чисел и строк словари также могут хранить и более сложные объекты - те же списки, кортежи или другие словари:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | users = {      "Tom": {          "phone": "+971478745",          "email": "tom12@gmail.com"      },      "Bob": {          "phone": "+876390444",          "email": "bob@gmail.com",          "skype": "bob123"      }  } |

В данном случае значение каждого элемента словаря в свою очередь представляет отдельный словарь.

Для обращения к элементам вложенного словаря соответственно необходимо использовать два ключа:

|  |  |
| --- | --- |
| 1  2  3 | old\_email = users["Tom"]["email"]  users["Tom"]["email"] = "supertom@gmail.com"  print(users["Tom"])     # { phone": "+971478745", "email": "supertom@gmail.com } |

Но если мы попробуем получить значение по ключу, который отсутствует в словаре, Python сгенерирует исключение KeyError:

|  |  |
| --- | --- |
| 1 | tom\_skype = users["Tom"]["skype"]   # KeyError |

Чтобы избежать ошибки, можно проверять наличие ключа в словаре:

|  |  |
| --- | --- |
| 1  2  3  4  5 | key = "skype"  if key in users["Tom"]:      print(users["Tom"]["skype"])  else:      print("skype is not found") |

Во всем остальном работа с комплексными и вложенными словарями аналогична работе с обычными словарями.

## Множества

Множество (set) представляют еще один вид набора, который хранит только уникальные элементы. Для определения множества используются фигурные скобки, в которых перечисляются элементы:

|  |  |
| --- | --- |
| 1  2 | users = {"Tom", "Bob", "Alice", "Tom"}  print(users)    # {"Alice", "Bob", "Tom"} |

Обратите внимание, что несмотря на то, что функция print вывела один раз элемент "Tom", хотя в определении множества этот элемент содержится два раза. Все потому что множество содержит только **уникальные** значения.

Также для определения множества может применяться функция **set()**, в которую передается список или кортеж элементов:

|  |  |
| --- | --- |
| 1  2  3 | people = ["Mike", "Bill", "Ted"]  users = set(people)  print(users)    # {"Mike", "Bill", "Ted"} |

Функцию set удобно применять для создания пустого множества:

|  |  |
| --- | --- |
| 1 | users = set() |

Для получения длины множества применяется встроенная функция **len()**:

|  |  |
| --- | --- |
| 1  2 | users = {"Tom", "Bob", "Alice"}  print(len(users))       # 3 |

### Добавление элементов

Для добавления одиночного элемента вызывается метод **add()**:

|  |  |
| --- | --- |
| 1  2  3 | users = set()  users.add("Sam")  print(users) |

### Удаление элементов

Для удаления одного элемента вызывается метод **remove()**, в который передается удаляемый элемент. Но следует учитывать, что если такого элемента не окажется в множестве, то будет сгенерирована ошибка. Поэтому перед удалением следует проверять на наличие элемента с помощью оператора **in**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | users = {"Tom", "Bob", "Alice"}    user = "Tom"  if user in users:      users.remove(user)  print(users)    # {"Bob", "Alice"} |

Также для удаления можно использовать метод **discard()**, который не будет генерировать исключения при отсутствии элемента:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | users = {"Tom", "Bob", "Alice"}    users.discard("Tim")    # элемент "Tim" отсутствует, и метод ничего не делает  print(users)    #  {"Tom", "Bob", "Alice"}    users.discard("Tom")    # элемент "Tom" есть, и метод удаляет элемент  print(users)    #  {"Bob", "Alice"} |

Для удаления всех элементов вызывается метод **clear()**:

|  |  |
| --- | --- |
| 1 | users.clear() |

### Перебор множества

Для перебора элементов можно использовать цикл for:

|  |  |
| --- | --- |
| 1  2  3  4 | users = {"Tom", "Bob", "Alice"}    for user in users:      print(user) |

При переборе каждый элемент помещается в переменную user.

### Операции с множествами

С помощью метода **copy()** можно скопировать содержимое одного множества в другую переменную:

|  |  |
| --- | --- |
| 1  2  3 | users = {"Tom", "Bob", "Alice"}  students = users.copy()  print(students)     # {"Tom", "Bob", "Alice"} |

### Объединение множеств

Метод **union()** объединяет два множества и возвращает новое множество:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}    users3 = users.union(users2)  print(users3)   # {"Bob", "Alice", "Sam", "Kate", "Tom"} |

### Пересечение множеств

Пересечение множеств позволяет получить только те элементы, которые есть одновременно в обоих множествах. Метод **intersection()** производит операцию пересечения множеств и возвращает новое множество:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}    users3 = users.intersection(users2)  print(users3)   # {"Bob"} |

Вместо метода intersection мы могли бы использовать операцию логического умножения:

|  |  |
| --- | --- |
| 1  2  3  4 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}    print(users & users2)   # {"Bob"} |

В этом случае мы получили бы тот же результат.

Модификация метода - **intersection\_update()** заменяет пересеченными элементами первое множество:

|  |  |
| --- | --- |
| 1  2  3  4 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}  users.intersection\_update(users2)  print(users)   # {"Bob"} |

### Разность множеств

Еще одна операция - разность множеств возвращает те элементы, которые есть в первом множестве, но отсутствуют во втором. Для получения разности множеств можно использовать метод **difference** или операцию вычитания:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}    users3 = users.difference(users2)  print(users3)           # {"Tom", "Alice"}  print(users - users2)   # {"Tom", "Alice"} |

Отдельная разновидность разности множеств - симметрическая разность производится с помощью метода **symmetric\_difference()** или с помощью операции ^. Она возвращает все элементы обоих множеств за исключением общих:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | users = {"Tom", "Bob", "Alice"}  users2 = {"Sam", "Kate", "Bob"}    users3 = users.symmetric\_difference(users2)  print(users3)   # {"Tom", "Alice", "Sam", "Kate"}    users4 = users ^ users2  print(users4)   # {"Tom", "Alice", "Sam", "Kate"} |

### Отношения между множествами

Метод **issubset** позволяет выяснить, является ли текущее множество подмножеством (то есть частью) другого множества:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}    print(users.issubset(superusers))   # True  print(superusers.issubset(users))   # False |

Метод **issuperset**, наоборот, возвращает True, если текущее множество является надмножеством (то есть содержит) для другого множества:

|  |  |
| --- | --- |
| 1  2  3  4  5 | users = {"Tom", "Bob", "Alice"}  superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}    print(users.issuperset(superusers))   # False  print(superusers.issuperset(users))   # True |

### frozen set

Тип **frozen set** является видом множеств, которое не может быть изменено. Для его создания используется функция **frozenset**:

|  |  |
| --- | --- |
| 1 | users = frozenset({"Tom", "Bob", "Alice"}) |

В функцию frozenset передается набор элементов - список, кортеж, другое множество.

В такое множество мы не можем добавить новые элементы, как и удалить из него уже имеющиеся. Собственно поэтому frozen set поддерживает ограниченный набор операций:

* len(s): возвращает длину множества
* x in s: возвращает True, если элемент x присутствует в множестве s
* x not in s: возвращает True, если элемент x отсутствует в множестве s
* s.issubset(t): возвращает True, если t содержит множество s
* s.issuperset(t): возвращает True, если t содержится в множестве s
* s.union(t)

: возвращает объединение множеств s и t

* s.intersection(t): возвращает пересечение множеств s и t
* s.difference(t): возвращает разность множеств s и t
* s.copy(): возвращает копию множества s

## List comprehension

Функциональность **list comprehension** предоставляет более краткий и лаконичный синтаксис для создания списков на основе других наборов данных. Она имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | newlist = [expression for item in iterable (if condition)] |

Синтаксис list comprehension состоит из следующих компонентов:

* iterable: перебираемый источник данных, в качестве которого может выступать список, множество, последовательность, либо даже функция, которая возвращает набор данных, например, range()
* item: извлекаемый из источника данных элемент
* expression: выражение, которое возвращает некоторое значение. Это значение затем попадает в генерируемый список
* condition: условие, которому должны соответствовать извлекаемые из источника данных элементы. Если элемент НЕ удовлетворяет условию, то он НЕ выбирается. Необязательный параметр.

Рассмотрим небольшой пример. Допустим, нам надо выбрать из списка все числа, которые больше 0. В обшем случае мы бы могли написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | numbers = [-3, -2, -1, 0, 1, 2, 3]  positive\_numbers = []  for n in numbers:      if n > 0:          positive\_numbers.append(n)    print(positive\_numbers)     # [1, 2, 3] |

Теперь изменим этот код, применив list comprehension:

|  |  |
| --- | --- |
| 1  2  3  4 | numbers = [-3, -2, -1, 0, 1, 2, 3]  positive\_numbers = [n for n in numbers if n > 0]    print(positive\_numbers)     # [1, 2, 3] |

Выражение [n for n in numbers if n > 0] говорит выбрать из списка numbers каждый элемент в переменную n, если n больше 0 и возврать n в результирующий список.

### источник данных iterable

В качестве источника данных iterable может использоваться любой перебираемый объект, например, другой список, словарь и т.д. Например, функция range() возвращает все числя нуля до указанного порога не включая:

|  |  |
| --- | --- |
| 1  2 | numbers = [n for n in range(10)]  print(numbers)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] |

Нередко данная конструкция применяется, чтобы создать из словаря список. Например, выберем из словаря все ключи:

|  |  |
| --- | --- |
| 1  2  3 | dictionary = {"red": "красный", "blue": "синий", "green": "зеленый"}  words = [word for word in dictionary]  print(words)    # ['red', 'blue', 'green'] |

### Возвращение результата

Параметр expression представляет выражение, которое возвращает некоторое значение. Это значение затем помещается в генерируемый список. В примерах выше это был текущий элемент, который извлекается из источника данных:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [-3, -2, -1, 0, 1, 2, 3]  new\_numbers = [n for n in numbers]  print(new\_numbers)      # [-3, -2, -1, 0, 1, 2, 3] |

Так, в данном случае параметр expression представляет непосредственно извлекаемый из списка numbers элемент n. Но это могут быть и более сложные значения. Например, возвратим удвоенное значение числа:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [-3, -2, -1, 0, 1, 2, 3]  new\_numbers = [n \* 2 for n in numbers]  print(new\_numbers)      # [-6, -4, -2, 0, 2, 4, 6] |

Здесь expression представляет выражение n \* 2

Это могут быть и более сложные выражения:

|  |  |
| --- | --- |
| 1  2  3 | numbers = [-3, -2, -1, 0, 1, 2, 3]  new\_numbers = [n \* 2 if n > 0 else n for n in numbers]  print(new\_numbers)      # [-3, -2, -1, 0, 2, 4, 6] |

Здесь параметр expression представляет выражение n \* 2 if n > 0 else n. В данном случае мы говорим возвратить значение n \* 2, если n > 0, иначе возвратить n.

В expression можно производить различные трансформации с данными. Например, возвратим также из словаря значение по ключу:

|  |  |
| --- | --- |
| 1  2  3 | dictionary = {"red": "красный", "blue": "синий", "green": "зеленый"}  words = [f"{key}: {dictionary[key]}" for key in dictionary]  print(words)    # ['red: красный', 'blue: синий', 'green: зеленый'] |

### Условие

Условие - параметр condition определяет фильтр для выбора элементов из источника данных. Применим условие для конкретизации выборки, например, выберем только четные числа:

|  |  |
| --- | --- |
| 1  2 | numbers = [n for n in range(10) if n % 2 == 0]  print(numbers)      # [0, 2, 4, 6, 8] |

Выберем только те ключи из словаря, длина которых больше 3:

|  |  |
| --- | --- |
| 1  2  3 | dictionary = {"red": "красный", "blue": "синий", "green": "зеленый"}  words = [f"{key}: {dictionary[key]}" for key in dictionary if len(key) > 3]  print(words)    # ['blue: синий', 'green: зеленый'] |

## Упаковка и распаковка

### Распаковка

**Распаковка** (**unpacking**, также называемая **Деструктуризация**) представляет разложение коллекции (кортежа, списка и т.д.) на отдельные значения.

Так, как и многие языки программирования, Python поддерживает концепцию множественного присваивания. Например:

|  |  |
| --- | --- |
| 1  2  3 | x, y = 1, 2  print(x)    # 1  print(y)    # 2 |

В данном случае присваивем значения сразу двум переменным. Присвоение идет по позиции: переменная x получает значение 1, а переменная y - значени 2.

Данный пример в действительности уже представляет деструктуризацию или распаковку. Значения 1, 2 фактически являются кортежом, поскольку именно запятые между значениями говорят о том, что это кортеж. И мы также могли бы написать следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | x, y = (1, 2)  print(x)    # 1  print(y)    # 2 |

В любом случае мы имеем дело с деструктуризацией, когда первый элемент кортежа передается первой переменной, второй элемент - второй переменной и так далее. То есть разложение идет по позиции.

Подобным образом можно разложить другие кортежи, например:

|  |  |
| --- | --- |
| 1  2  3  4 | name, age, company = ("Tom", 38, "Google")  print(name)         # Tom  print(age)          # 38  print(company)      # Google |

Только кортежами мы не ограничены и можем "распаковывать" и другие коллекции, например, списки:

|  |  |
| --- | --- |
| 1  2  3  4  5 | people = ["Tom", "Bob", "Sam"]  first, second, third = people  print(first)      # Tom  print(second)     # Bob  print(third)      # Sam |

При разложении словаря переменные получают ключи словаря:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | dictionary = {"red": "красный", "blue": "синий", "green": "зеленый"}  r, b, g = dictionary  print(r)    # red  print(b)    # blue  print(g)    # green  # получаем значение по ключу  print(dictionary[g])    # зеленый |

### Деструктуризация в циклах

Циклы в Python позволяют разложить коллекции на отдельные составляющие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | people = [      ("Tom", 38, "Google"),      ("Bob", 42, "Microsoft"),      ("Sam", 29, "JetBrains")  ]    for name, age, company in people:      print(f"Name: {name}, Age: {age}, Company: {company}") |

Здесь мы перебираем список кортежей people. Каждый кортеж состоит из трех элементов, соответственно при переборе мы можем их передать в переменные name, age и company.

Другой пример - функция **enumerate()**. Она принимает в качестве параметра коллекцию, создает для каждого элемента кортеж и возвращает набор из подобных кортежей. Каждый кортеж содержит индекс, который увеличивается с каждой итерацией:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | people = ["Tom", "Bob", "Sam"]  for index, name in enumerate(people):      print(f"{index}.{name}")    # результат  # 0.Tom  # 1.Bob  # 2.Sam |

### Игнорирование значений

Если какой-то элемент коллекции не нужен, то обычно для него определяется переменная с именем \_ (прочерк):

|  |  |
| --- | --- |
| 1  2  3  4 | person =("Tom", 38, "Google")  name, \_, company = person  print(name)     # Tom  print(company)  # Google |

Здесь нам не важен второй элемент кортежа, поэтому для него определяем переменную \_. Хотя в реальности \_ - такое же действительное имя, как name и company:

|  |  |
| --- | --- |
| 1  2 | name, \_, company = person  print(\_)     # 38 |

### Упаковка значений и оператор \*

Оператор **\*** упаковывает значение в коллекцию. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | num1=1  num2=2  num3=3  \*numbers,=num1,num2,num3  print(numbers)  #[1, 2, 3] |

Здесь мы упаковываем значения из кортежа (num1,num2,num3) в список numbers. Причем, чтобы получить список, после numbers указывается запятая.

Как правило, упаковка применяется для сбора значений, которые остались после присвоения результатов деструктуризации. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | head, \*tail = [1, 2, 3, 4, 5]    print(head)  # 1  print(tail)  # [2, 3, 4, 5] |

Здесь переменная head в соответствии с позицией получае первый элемент списка. Все остальные элементы передаются в переменную tail. Таким образом, переменная tail будет представлять список из оставшихся элементов.

Аналогичным образом можно получить все кроме последнего:

|  |  |
| --- | --- |
| 1  2  3  4 | \*head, tail = [1, 2, 3, 4, 5]    print(head)  # [1, 2, 3, 4]  print(tail)  # 5 |

Или элементы по середине, кроме первого и последнего:

|  |  |
| --- | --- |
| 1  2  3  4  5 | head, \*middle, tail = [1, 2, 3, 4, 5]    print(head)    # 1  print(middle)  # [2, 3, 4]  print(tail)    # 5 |

Или все кроме первого и второго:

|  |  |
| --- | --- |
| 1  2  3  4  5 | first, second, \*other = [1, 2, 3, 4, 5]    print(first)    # 1  print(second)   # 2  print(other)    # [3, 4, 5] |

Вообщем, таким образом мы можем получать различные комбинации элементов коллекции. Причем не только списков, но и кортежей, словарей и других коллекций.

Другой пример - нам надо получить только первый, третий и последний элемент, а остальные элементы нам не нужны. В общем случае мы должны предоставить переменные для всех элементов коллекции. Однако если коллекция имеет 100 элементов, а нам нужно только три, не будем же мы определять все сто переменных. И в этом случае опять же можно применить упаковку:

|  |  |
| --- | --- |
| 1  2  3  4  5 | first, \_, third, \*\_, last = [1, 2, 3, 4, 5, 6, 7, 8]    print(first)   # 1  print(third)   # 3  print(last)    # 8 |

Также можно получить ключи словаря:

|  |  |
| --- | --- |
| 1  2  3  4  5 | red, \*other, green = {"red":"красный", "blue":"синий", "yellow":"желтый", "green":"зеленый"}    print(red)          # red  print(green)        # green  print(other)        # ['blue', 'yellow'] |

### Распаковка и операторы \* и \*\*

Оператор **\*** вместе с оператором **\*\*** также может применяться для распаковки значений. Оператор **\*** используется для распаковки кортежей, списков, строк, множеств, а оператор **\*\*** - для распаковки словарей. Особенно это может быть полезно, когда на основе одних коллекций создаются другие. Например, распаковка кортежей и списков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | nums1 = [1, 2, 3]  nums2 = (4, 5, 6)    # распаковываем список nums1 и кортеж nums2  nums3 = [\*nums1, \*nums2]  print(nums3)        # [1, 2, 3, 4, 5, 6] |

Здесь распаковывем значения из списка nums1 и кортежа nums2 и помещаем их в список nums3.

Подобным образом раскладываются словари, только применяется оператор **\*\***:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | dictionary1 = {"red":"красный", "blue":"синий"}  dictionary2 = {"green":"зеленый", "yellow":"желтый"}    # распаковываем словари  dictionary3 = {\*\*dictionary1, \*\*dictionary2}  print(dictionary3)  # {'red': 'красный', 'blue': 'синий', 'green': 'зеленый', 'yellow': 'желтый'} |

## Упаковка и распаковка в параметрах функций

Одной из распространенных сфер, где применяются упаковка и распаковка - это параметры функций. Так, в определениях различных функций нередко можно увидеть, что они принимают такие параметры как \*args и \*\*kwargs.

Термины args и kwargs — это соглашения по программированию на Python, в реальности вместо них можно использовать любые именования. **\*args** представляет параметры, которые передаются по позиции. А **\*\*kwargs** означает параметры, которые передаются по имени. обозначает аргументы ключевого слова.

Оператор **\*** применяется с любым итерируемым объектом (например, кортежем, списком и строками). Тогда как оператор **\*\*** можно использовать только со словарями.

### \*args

Оператор **\*** позволяет передать в функцию несколько значений, и все они будут упакованы в кортеж:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def fun(\*args):      # обращаемся к первому элементу кортежа      print(args[0])        # выводим весь кортеж      print(args)    fun("Python", "C++", "Java", "C#") |

Здесь функция fun принимает кортеж значений. При вызове мы можем передать ей различное количество значений. Так, в примере выше передается четыре строки, которые образуют кортеж. Консольный вывод программы:

Python

('Python', 'C++', 'Java', 'C#')

Благодаря такой возможности мы можем передавать в функцию переменное количество значений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def sum(\*args):      result = 0      for arg in args:          result += arg      return result    print(sum(1, 2, 3))         # 6  print(sum(1, 2, 3, 4))      # 10  print(sum(1, 2, 3, 4, 5))   # 15 |

### Оператор \*\*

Оператор **\*\*** упаковывает аргументы, переданные по имени, в словарь. Имена параметров служат ключами. Например, определим функцию, которая просто будет выводить все переданные параметры

|  |  |
| --- | --- |
| 1  2  3  4  5 | def fun(\*\*kwargs):      print(kwargs)   # выводим словарь на консоль    fun(name="Tom", age="38", company="Google")  fun(language="Python", version="3.11") |

Консольный вывод программы:

{'name': 'Tom', 'age': '38', 'company': 'Google'}

{'language': 'Python', 'version': '3.11'}

Поскольку аргументы передаются в функцию в виде словаря, то внутри функции через ключи мы можем получить их значения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def fun(\*\*kwargs):      for key in kwargs:          print(f"{key} = {kwargs[key]}")    fun(name="Tom", age="38", company="Google") |

Консольный вывод программы:

name = Tom

age = 38

company = Google

### Распаковка аргументов

Выше было описано, как операторы **\*** и **\*\*** применяются для упаковки аругментов в кортеж и словарь соответственно. Но эти же операторы могут использоваться для распаковки.

#### Оператор \* и распаковка

Сначала рассмотрим ситуацию, где это может пригодиться.Пусть мы передаем в функцию кортеж:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def sum(\*args):    result = 0    for arg in args:      result += arg    return result    numbers = (1, 2, 3, 4, 5)  print(sum(numbers)) |

Здесь в вызов функции sum передается кортеж. Параметр \*args по сути тоже представляет кортеж, и кажется, все должно работать. Тем не менее мы столкнемся с ошибкой

TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'

То есть в данном случае кортеж numbers передается как элемент кортежа \*args.

И чтобы элементы кортежа были переданы в кортеж \*args как отдельные значения, необходимо выполнить их распаковку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def sum(\*args):    result = 0    for arg in args:      result += arg    return result    numbers = (1, 2, 3, 4, 5)  # применяем распаковку - \*numbers  print(sum(\*numbers))     # 15 |

Здесь при передачи кортежа numbers в функцию sym применяется распаковка: \*numbers

Другим случаем распаковки может быть ситуация, когда функция принимает несколько параметров, а мы передаем один кортеж или список:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def print\_person(name, age, company):    print(f"Name:{name}, Age: {age}, Company: {company}")    person =("Tom", 38, "Google")  # выполняем распаковку кортежа person  print\_person(\*person)   # Name:Tom, Age: 38, Company: Google |

В данном случае выражение \*person раскладывает кортеж person на отдельные значения, которые передаются параметрам name, age и company.

#### Оператор \*\* и распаковка

Оператор **\*\*** применяется для распаковки словарей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def print\_person(name, age, company):    print(f"Name:{name}, Age: {age}, Company: {company}")    tom ={"name":"Tom", "age":38, "company":"Google"}  # выполняем распаковку словаря tom  print\_person(\*\*tom) # Name:Tom, Age: 38, Company: Google |

Здесь выражение \*\*tom раскладывает словарь на отдельные значения, которые передаются параметрам name, age и company по названию ключей.

### Сочетание параметров

Параметры \*args и \*kwargs могут использоваться в функции вместе с другими параметрами. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def sum(num1, num2, \*nums):      result=num1+num2      for n in nums:          result += n      return result    print(sum(1,2,3))       # 6  print(sum(1,2,3,4))     # 10 |