

Типы данных

01 Компьютер. Процесс. Память. РуObject

Компьютер

это вычислительное устройство, которое может принимать информацию, обрабатывать ее и по какой-то инструкции и выдавать результат (тоже в виде информации)

В компьютере информация представлена в цифровом виде. Цифровое представление информации основано на использовании двоичной системы счисления, которая использует только два символа - 0 и 1, отсюда и второе название представления информации в компьютере - **двоичный код**

В компьютере информация хранится и обрабатывается в форме битов (бинарных единиц) и байтов (групп битов)

Бит

это наименьшая единица информации и может представлять либо 0, либо

Байт

состоит из 8 битов и может представлять 256 различных значений (от 0 до 255)

Компьютер работает с информацией в следующей последовательности

01 Ввод информации

Пользователь вводит информацию с помощью вводных устройств, таких как клавиатура, мышь, сканер или микрофон. Или информация может быть получена от других устройств или сетей

02 Хранение информации

Информация, полученная от пользователя или других устройств, сохраняется в оперативной памяти (RAM) компьютера или на постоянных носителях информации, таких как жесткий диск или SSD

Random Access Memory (RAM)

это запоминающее устройство, позволяющее быстро считывать и записывать информацию. Без электрической энергии оперативная память не может ничего хранить

03 Обработка информации

Компьютер обрабатывает информацию посредством выполнения программного кода. Центральный процессор (CPU) выполняет операции и инструкции, указанные в программе. Это может включать такие операции, как арифметические вычисления, логические операции, сравнения и условия

Оперативная память, которая выделяется процессу, называется контекстом процесса. Эту область памяти можно разделить на несколько отдельных частей, которые можно выделить в блоки

Первый блок – это код, инструкция, которая должна выполняться на процессоре. Остальные области: `heap`, `free`, `stack`. В `heap` и `stack` будут храниться все наши объекты, с которым будем работать. `Free` – это еще не занятая область контекста процесса, в которую можно записывать новые данные

04 Вывод информации

Результаты обработки информации выводятся на устройства вывода, такие как монитор, принтер или динамики. Это может быть в виде текста, изображений, звуков или других форматов

PyObject

это базовый класс, который наследуется абсолютно каждым объектом в Python

Контекст процесса

- Блок с адресом ячейки памяти, в которой записаны данные
- Блок с описанием типа, в котором хранится ссылка на тип
- Тип хранит в себе описание структуры хранимых данных и операций над ним
- Счетчик ссылок

PyObject является типом объекта в Python. Он представляет общий тип для всех объектов в Python, таких как числа, строки, списки, словари и функции. Конкретный тип объекта, содержащегося в переменной `PyObject`, определяется во время выполнения программы

02

Изменяемые и неизменяемые типы

В Python есть два основных типа данных

Неизменяемые типы данных

immutable types

это типы данных, значения которых невозможно изменить после их создания. Это означает, что при попытке изменить значение неизменяемого типа данных создается новый объект с новым значением, а существующий объект остается неизменным. Примеры неизменяемых типов данных в Python: числа (int, float), строки (str) и кортежи (tuple)

Изменяемые типы данных

mutable types

это типы данных, значения которых могут быть изменены после их создания. В отличие от неизменяемых типов данных, при изменении значения изменяемого типа данных изменяется сам объект, а не создается новый объект. Примеры изменяемых типов данных в Python: списки (list), множества (set) и словари (dict)

03

Списки

Массив

это коллекция элементов одного типа, расположенных в памяти последовательно. В массиве каждый элемент имеет свой индекс, начиная с 0

Массивы могут быть одномерными (векторами), двумерными (матрицами) или многомерными, в зависимости от количества индексов, необходимых для доступа к элементам

Каждый массив имеет имя. В языке программирования Python, имя массива может быть любым допустимым идентификатором. Однако, наиболее распространенное и рекомендуемое соглашение состоит в использовании понятных и описательных имен переменных, которые отражают ее содержимое или назначение

Например, если у вас есть массив с оценками студентов, вы можете назвать его "grades" (оценки) или "student_scores" (оценки_студентов). Если это список имен людей, вы можете использовать имя "names" (имена) или "person_list" (список_людей)

i	—>	0	1	2	3	4	5	6	7	8	9
A[i]		-5	-2	-6	-1	0	4	2	3	-1	-3

A - имя массива

i - индекс элемента

В отличие от языков многих других языков, в Python используется встроенный тип данных "список" (list), который предоставляет функциональность массива и содержит элементы разных типов данных. Все элементы списка хранятся внутри него и могут быть изменены, даже если его длина (количество элементов) изменяется

Динамические массивы | это массивы, размер которых может изменяться во время выполнения программы

Списки в Python | это динамические массивы

Списки являются одним из основных встроенных типов данных в Python и предоставляют мощные возможности для работы с набором элементов

Создание и инициализация списка

Пример

... . . .	
my_list = []	# пустой список
my_list = [1, 2, 3]	# список с элементами
my_list = ['apple', 'banana', 'cherry']	# список со строками
my_list = [1, 'apple', True]	# список с разными типами данных

Длина списка

Для определения длины (количества элементов) списка используется функция len()

Пример

... . . .	
my_list = [1, 2, 3, 4, 5]	
list_length = len(my_list)	
print(list_length) # выводит: 5	

Функция len() принимает список как аргумент и возвращает целое число, представляющее количество элементов в списке

Доступ к элементам списка

Доступ к элементам списка происходит с использованием индексов. Индексы начинаются с 0, при этом отрицательные индексы обозначают обратную нумерацию с конца списка

Пример

```
...
my_list = [1, 2, 3, 4, 5]
print(my_list[0])      # выводит первый элемент: 1
print(my_list[2])      # выводит третий элемент: 3
print(my_list[-1])     # выводит последний элемент: 5
```

Добавление и удаление элементов

Списки в Python являются **изменяемыми (mutable)** объектами, поэтому их можно изменять, добавлять или удалять элементы

Пример

```
...
my_list = [1, 2, 3]
my_list[0] = 10      # изменение значения элемента
print(my_list)       # [10, 2, 3]

my_list.append(4)    # добавление элемента в конец списка
print(my_list)       # [10, 2, 3, 4]

my_list.remove(2)   # удаление элемента из списка
print(my_list)       # [10, 3, 4]
```

Встроенные методы

Списки также поддерживают множество встроенных методов, таких как сортировка, поиск элемента и другие. С помощью срезов (slicing) можно получить подсписки из списка

Пример

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

sub_list = my_list[1:4]  # получение подсписка с индексами от 1 до 3
print(sub_list)          # [2, 3, 4]

sorted_list = sorted(my_list) # сортировка списка
print(sorted_list)          # [1, 2, 3, 4, 5]
```

Списки в Python представляют удобный и гибкий способ работы с набором данных и обеспечивают много возможностей для манипулирования элементами

Объединение списков

В Python существуют несколько способов объединения или конкатенации (соединения) списков.

01 Можно использовать оператор "+", чтобы объединить два списка

Пример

```
...
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print(combined_list) # Выводит: [1, 2, 3, 4, 5, 6]
```

02 Метод extend(): Метод extend() позволяет добавить все элементы одного списка к другому

Пример

```
...
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1) # Выводит: [1, 2, 3, 4, 5, 6]
```

03 Можно использовать оператор "*", чтобы повторить список несколько раз и объединить результат

Пример

```
...
list1 = [1, 2, 3]
repeated_list = list1 * 3
print(repeated_list) # Выводит: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

04 Можно также использовать метод append() для добавления элементов одного списка в конец другого списка

Пример

```
...
list1 = [1, 2, 3]
list2 = [4, 5, 6]
for item in list2:
    list1.append(item)
print(list1) # Выводит: [1, 2, 3, 4, 5, 6]
```

Оператор in

Оператор `in` используется для проверки наличия элемента в списке или другом итерируемом объекте. Возвращает `True`, если элемент присутствует в списке, и `False`, если элемент отсутствует

Пример

```
...
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)    # Выводит: True
print(6 in my_list)    # Выводит: False
```

В первом примере, оператор `3 in my_list` возвращает `True`, потому что значение 3 присутствует в списке `my_list`. Во втором примере, оператор `6 in my_list` возвращает `False`, потому что значение 6 отсутствует в списке `my_list`

Оператор `in` также может использоваться в условных выражениях и циклах для проверки наличия элементов в списке

Срезы

Срезы (slicing) в списках Python позволяют получить подсписок из исходного списка. Срезы выполняются с использованием квадратных скобок `[]` и индексов элементов. Формат среза выглядит следующим образом: `[start:stop:step]`

01 Получение подсписка с помощью стартового и конечного индексов

Пример

```
...
my_list = [1, 2, 3, 4, 5]
sub_list = my_list[1:4]    # Получение подсписка с индексами от 1 до 3
print(sub_list)           # Выводит: [2, 3, 4]
```

В этом примере, с помощью среза `[1:4]` мы получаем подсписок, начиная с элемента с индексом 1 и заканчивая элементом с индексом 3

02 Получение подсписка с определенным шагом

Пример

```
...
my_list = [1, 2, 3, 4, 5]
sub_list = my_list[::2]    # Получение подсписка с шагом 2
print(sub_list)           # Выводит: [1, 3, 5]
```

В этом примере, с помощью среза `[::2]` мы получаем подсписок с каждым вторым элементом в исходном списке

03 Использование отрицательных индексов

Пример

```
...  
my_list = [1, 2, 3, 4, 5]  
sub_list = my_list[-3:-1]    # Получение подсписка с отрицательными индексами  
print(sub_list)            # Выводит: [3, 4]
```

В этом примере, с помощью среза `[-3:-1]` мы получаем подсписок, начиная с третьего элемента с конца и заканчивая вторым элементом с конца

Срезы могут быть комбинированы и адаптированы для разных потребностей. Примеры выше предоставляют основы использования срезов в списках Python, и вы можете экспериментировать с разными значениями индексов и шагов, чтобы получить нужный подсписок

04 Кортежи

Кортеж | это коллекция, которая является итерируемым объектом

То есть по кортежу можно:

- ✓ Итерироваться
- ✓ Обращаться к элементам по индексам
- ✓ Брать срезы

Способы создания кортежей

01 Использование круглых скобок

Пример

```
...  
my_tuple = ()                      # пустой кортеж  
my_tuple = (1, 2, 3)                # кортеж с элементами  
my_tuple = ("apple", "banana", "cherry") # кортеж со строковыми элементами  
my_tuple = (1, "apple", True)       # кортеж с разными типами данных
```

02 Использование запятых без скобок

Пример

```
...  
my_tuple = 1, 2, 3                  # кортеж с элементами  
my_tuple = "apple", "banana", "cherry" # кортеж со строковыми элементами  
my_tuple = 1, "apple", True          # кортеж с разными типами данных
```

Обращение к элементу кортежа

Для доступа к элементам кортежа в Python используется оператор индексации [] или срезы (slicing)

01 Оператор индексации []

Индексы в кортеже начинаются с 0, где 0 обозначает первый элемент, 1 - второй элемент и так далее. Отрицательные индексы обозначают элементы с конца кортежа

Пример

```
...
my_tuple = ("apple", "banana", "cherry")
print(my_tuple[0])          # Выводит: "apple"
print(my_tuple[2])          # Выводит: "cherry"
print(my_tuple[-1])         # Выводит: "cherry"
```

02 Срезы (slicing)

С помощью срезов можно получить подкортеж из исходного кортежа, указав начальный и конечный индексы (включительно) с заданным шагом

Пример

```
...
my_tuple = ("apple", "banana", "cherry", "date", "elderberry")
print(my_tuple[1:4])        # Выводит: ("banana", "cherry", "date")
print(my_tuple[:3])         # Выводит: ("apple", "banana", "cherry")
print(my_tuple[2:])         # Выводит: ("cherry", "date", "elderberry")
print(my_tuple[::-2])       # Выводит: ("apple", "cherry", "elderberry")
```

Обратите внимание, что операции доступа с индексами и срезами возвращают значения в виде нового кортежа

Неизменяемость кортежей

Кортежи в Python являются неизменяемыми (immutable) объектами, что означает, что их элементы не могут быть изменены после создания кортежа. После того, как кортеж создан, его элементы остаются неизменными

Примером неизменяемого кортежа может служить набор координат

Пример

```
...
point = (3, 4)      # Кортеж с координатами (3, 4)
```

Вместо изменения элементов кортежа необходимо создавать новый кортеж с обновленными значениями или выполнять другие операции для создания и изменения новых кортежей

Пример

```
...  
point = (3, 4)      # Исходный кортеж  
new_point = (point[0] + 1, point[1]) # Создание нового кортежа с обновленным значением  
print(new_point)    # Выводит: (4, 4)
```

При попытке изменить элемент кортежа напрямую будет возникать ошибка.

Пример

```
...  
point = (3, 4)  
point[0] = 5      # Ошибка: кортежи не поддерживают присваивание значения элементу
```

Кортежи являются полезным инструментом в Python, особенно в случаях, когда требуется гарантировать, что данные не будут изменены

Множественное присваивание

Множественное присваивание в кортежах в Python позволяет одновременно присвоить несколько значений переменным с использованием кортежей

Пример

```
...  
my_tuple = (1, 2, 3)  
a, b, c = my_tuple  
print(a)  # Выводит: 1  
print(b)  # Выводит: 2  
print(c)  # Выводит: 3
```

В этом примере значения, содержащиеся в кортеже `my_tuple`, были присвоены переменным `a`, `b` и `c`. Каждое значение кортежа было присвоено соответствующей переменной

Сравнение кортежей

Сравнение кортежей выполняется поэлементно с помощью операторов сравнения, таких как `==`, `!=`, `>`, `<`, `≥` и `≤`. Сравнение происходит от первого элемента до последнего, и сравниваются значения элементов на соответствующих позициях

Пример

```
...  
tuple1 = (1, 2, 3)  
tuple2 = (1, 2, 3)  
tuple3 = (4, 5, 6)  
print(tuple1 == tuple2)    # Выводит: True, так как tuple1 и tuple2 содержат одинаковые  
                           # значения на каждой позиции  
print(tuple1 != tuple3)    # Выводит: False, так как tuple1 и tuple3 имеют разные значения  
                           # на одной или нескольких позициях  
print(tuple1 > tuple3)    # Выводит: False, так как tuple1 не является больше tuple3 на  
                           # первой позиции  
print(tuple2 < tuple3)    # Выводит: True, так как tuple2 меньше tuple3 на каждой  
                           # позиции
```

При сравнении кортежей, элементы сравниваются один за другим в порядке их расположения. Если элементы на соответствующих позициях равны, сравнение переходит к следующему элементу. Если все элементы на всех позициях совпадают, кортежи считаются равными. Если хотя бы один элемент на соответствующей позиции не совпадает, сравнение определяется по результату сравнения этих элементов

05

Строки

Строки

это коллекции символов в кодировке Unicode. Строки являются неизменяемым типом данных и подобны кортежам

Основные методы работы со строками

- Поиск подстроки
- Изменение регистра символов

Работа со строками в Python предоставляет множество возможностей для манипулирования, форматирования, изменения и обработки текстовой информации

Создание строки

Строки можно создавать с использованием одинарных ("'), двойных ("""') кавычек или тройных ("""") или """")

Пример

```
...  
my_string = 'Hello, World!'          # Создание строки с использованием  
одинарных кавычек  
my_string = "Hello, World!"         # Создание строки с использованием двойных  
кавычек  
my_string = '''Hello, World!'''     # Создание строки с использованием тройных  
одинарных кавычек  
my_string = """Hello, World!"""      # Создание строки с использованием тройных  
двойных кавычек
```

Строки в Python могут содержать буквы, цифры и специальные символы, а также различные символы Unicode. Одинарные кавычки могут быть использованы внутри строки, заключенной в двойные кавычки, и наоборот

Пример

```
...  
my_string = "I'm learning Python!"    # Использование одинарных кавычек  
внутри двойных кавычек  
my_string = 'He said, "Hello, World!"'  # Использование двойных кавычек внутри  
одинарных кавычек
```

Доступ к символам и срезы

Пример

```
...  
my_string = "Hello, World!"  
print(my_string[0])      # Выводит: "H"  
print(my_string[7:12])   # Выводит: "World"
```

Длина строки

Длина строки определяется при помощи команды `len`

Пример

```
...  
my_string = "Пример строки"  
length = len(my_string)  
print(length)  # Выведет 14
```

Неизменяемый тип

В Python строки являются неизменяемым типом данных. Это означает, что после создания строки ее невозможно изменить

Пример

```
...  
my_string = "Пример строки"  
my_string[0] = "H" # Попытка изменить первый символ на "H"
```

При выполнении этого кода будет вызвано исключение TypeError: 'str' object does not support item assignment (Тип данных 'str' не поддерживает присваивание элемента). Это происходит потому, что строки в Python неизменяемы и не поддерживают присваивание отдельных символов

Перебор символов строки

Символы строки можно перебирать при помощи цикла for

Пример

```
...  
my_string = "Пример строки"  
for char in my_string:  
    print(char)
```

Вывод

```
...  
П  
р  
и  
м  
е  
р  
с  
т  
р  
о  
к  
и
```

В цикле for, мы используем переменную char для хранения каждого символа из строки my_string. В каждой итерации цикла, переменная char будет содержать следующий символ строки. Затем, мы просто выводим текущий символ на экран

Конкатенация строк (сложение)

Пример

```
...  
string1 = "Hello"  
string2 = "World"  
result = string1 + " " + string2 # Конкатенация двух строк с пробелом между ними  
print(result) # Выводит: "Hello World"
```

Метод join()

Метод join() в Python используется для объединения элементов списка (или любой итерируемой коллекции) в одну строку. Он принимает список или итерируемый объект в качестве параметра и возвращает новую строку, состоящую из элементов этого списка, разделенных заданной строкой-разделителем

Пример

```
...  
my_list = ['Пример', 'строки', 'для', 'join']  
separator = ''  
result = separator.join(my_list)  
print(result) # Выведет "Пример строки для join"
```

В этом примере мы объединяем элементы списка my_list с помощью метода join(). Разделителем является строка '', что означает, что каждый элемент списка будет разделен пробелом. Результат сохраняется в переменной result и затем выводится на экран

Различные методы работы со строками

01 upper()

Переводит все символы строки в верхний регистр

Пример

```
...  
my_string = "пример строки"  
result = my_string.upper()  
print(result) # Выведет "ПРИМЕР СТРОКИ"
```

02 lower()

Переводит все символы строки в нижний регистр

Пример

```
...  
my_string = "пример строки"  
result = my_string.lower()  
print(result) # Выведет "ПРИМЕР СТРОКИ"
```

03 capitalize()

Переводит первый символ строки в верхний регистр, а все остальные - в нижний регистр

Пример

```
...  
my_string = "пример СТРОКИ"  
result = my_string.capitalize()  
print(result) # Выведет "Пример строки"
```

04 strip()

Удаляет начальные и конечные пробелы из строки

Пример

```
...
my_string = "    Пример строки    "
result = my_string.strip()
print(result) # Выведет "Пример строки"
```

05 split()

Разбивает строку на список подстрок по указанному разделителю

Пример

```
...
my_string = "Пример строки для разделения"
result = my_string.split()
print(result) # Выведет ['Пример', 'строки', 'для', 'разделения']
my_string = "Пример-строки-для-разделения"
result = my_string.split('-')
print(result) # Выведет ['Пример', 'строки', 'для', 'разделения']
```

06 replace()

Заменяет все вхождения указанной подстроки на другую подстроку

Пример

```
...
my_string = "Пример строки для замены"
result = my_string.replace("строки", "текста")
print(result) # Выведет "Пример текста для замены"
```

07 find()

Ищет первое вхождение подстроки в строку и возвращает индекс или -1, если подстрока не найдена

Пример

```
...
my_string = "Пример строки для поиска"
result = my_string.find("строки")
print(result) # Выведет 7
```

F-строки

F-строки (или форматированные строки) в Python представляют собой удобный способ вставки значений переменных в строку без необходимости использования конкатенации или сложных операций форматирования

Синтаксис F-строки в Python состоит из префикса f перед открывающей кавычкой строки. Внутри F-строки можно включать выражения, заключенные в фигурные скобки {} для вставки значений переменных

Пример

```
ooo  
name = "Алина"  
age = 25  
# Вставка значений переменных в строку  
result = f"Меня зовут {name} и мне {age} лет."  
print(result) # Выведет "Меня зовут Алина и мне 25 лет."  
# Вычисление выражений внутри F-строки  
result = f"Сумма 5 и 3 равна {5 + 3}."  
print(result) # Выведет "Сумма 5 и 3 равна 8."  
# Вставка значений переменных с форматированием  
balance = 3125.67  
result = f"Баланс на счете: ${balance:.2f}"  
print(result) # Выведет "Баланс на счете: $3125.67"
```

Обратите внимание на использование фигурных скобок внутри F-строки для вставки значений переменных. Вы также можете использовать выражения внутри фигурных скобок для выполнения вычислений или форматирования чисел и дат

06

Хэшируемость. Метод hash()

Хеш-функция

hash – «превращать в фарш», «мешанина»

функция, которая сопоставляет объекту целое число фиксированной длины

Хеш-функция используется для обеспечения уникальности и целостности данных, быстрого поиска и проверки целостности файлов

Хеш

значение возвращаемое хеш-функцией. Хеш функция должна возвращать одинаковые хеши для одинаковых объектов

Идеальная хеш-функция

функция, которая обеспечивает уникальный хеш-код для каждого входного значения, не имеет коллизий (совпадений хеш-кодов для разных входных значений)

Хорошая хеш-функция

хеш-функция, которая равномерно распределяет хеши объектов по множеству всех допустимых хешей

Хешируемые объекты

встроенные неизменяемые типы данных, а также другие типы данных, для которых определена функция `hash` в классе

В Python метод `hash()` используется для вычисления хэш-значения (хэш-кода) объекта. Хэш-значение – это числовое представление объекта, которое используется для быстрого и эффективного поиска, сравнения и проверки объектов в структурах данных, таких как словари или множества

Синтаксис метода `hash()` в Python

`hash(object)`, где `object` – объект, для которого вы хотите вычислить хэш-значение

Хеширование целых чисел

Хеш для целых чисел это остаток от деления числа на число $2^{61} - 1$

Пример

```
...  
print(hash(1256))  
print(hash(-1))  
print(hash(4258468))  
print(hash(35735824681486846854))  
print(hash(2305843009213693950))  
print(hash(2305843009213693951))
```

Вывод

```
...  
1256  
-2  
4258468  
1148179543281437589  
2305843009213693950  
0
```

Хеширование вещественных чисел

Пример

```
...  
print(hash(23.64))
```

Вывод

```
...  
1475739525896765463
```

Хеширование кортежей

Значение хеша кортежа может меняться от запуска программы к запуску, так как при вычислении хешей для коллекций используется случайное число, генерируемое при запуске программы

Пример

```
...  
print(hash((4, -2, "Hello", 2.5)))  
print(hash((4, -2, "Hello", 2.5)))
```

Вывод

```
...  
8242004071891041952  
8242004071891041952
```

Хеширование строк

Пример

```
...  
print(hash("Hello"))
```

Вывод

```
...  
-7331187157525728317
```

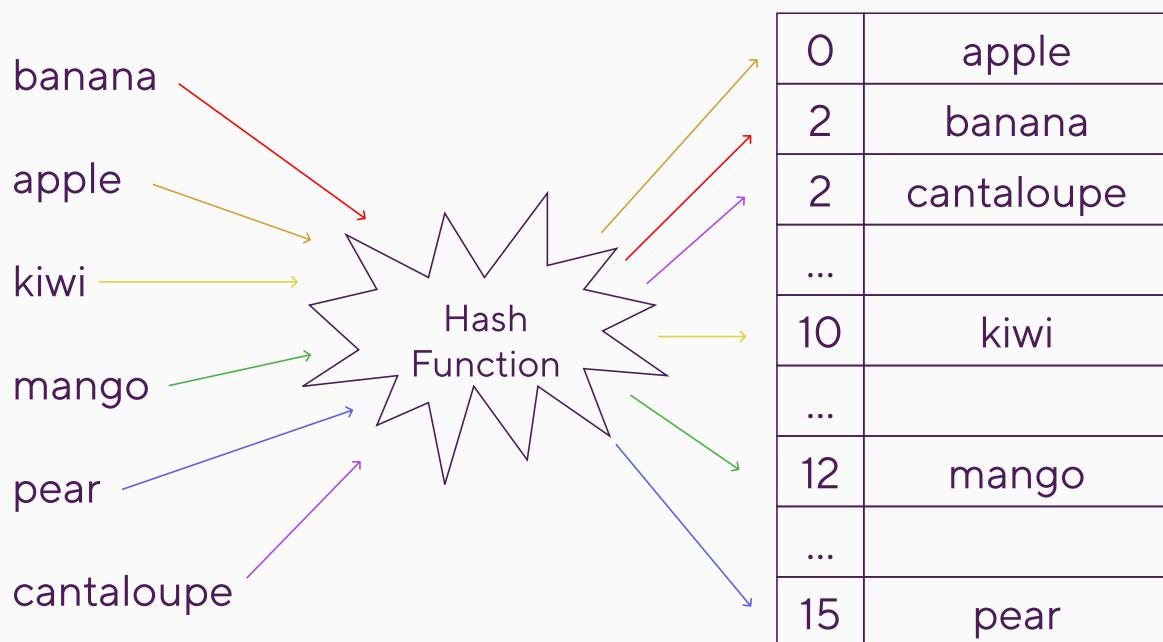
Хеш-таблица

Хеш-таблица

Неупорядоченная коллекция, которая представляет из себя множество ключей, связанных с какими-то данными

Ключи

Хешируемые объекты. Ключи - уникальные объекты для хеш-таблицы

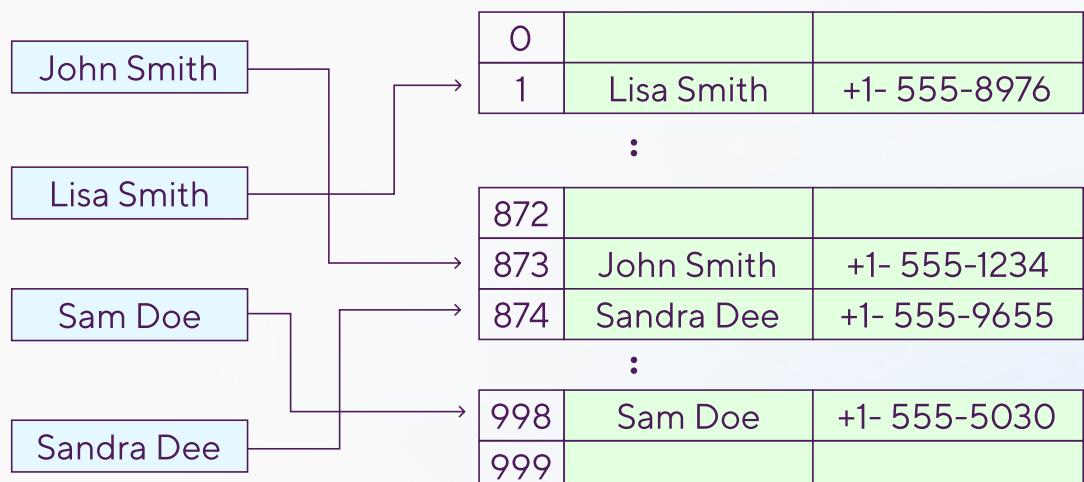


Элементы в таблице хранятся в неупорядоченном виде

keys

Indices

Key-value pairs (records)



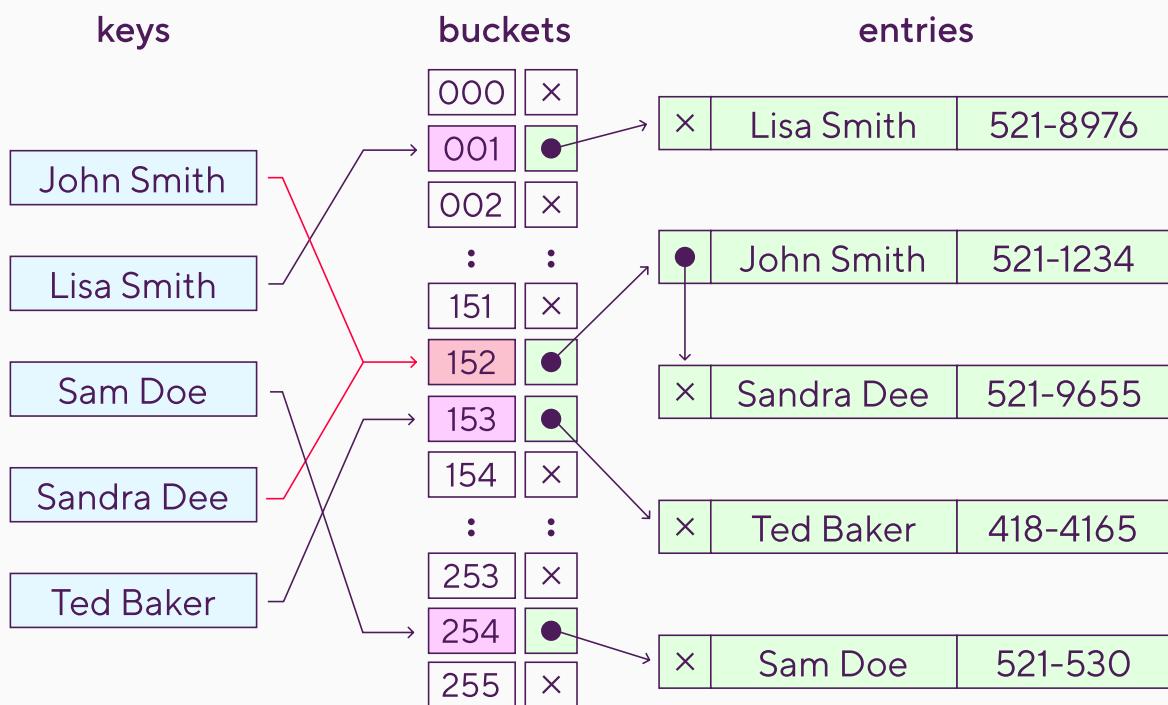
Коллизия

Это ситуация, когда хеши двух разных объектов совпадают

Разрешение коллизий

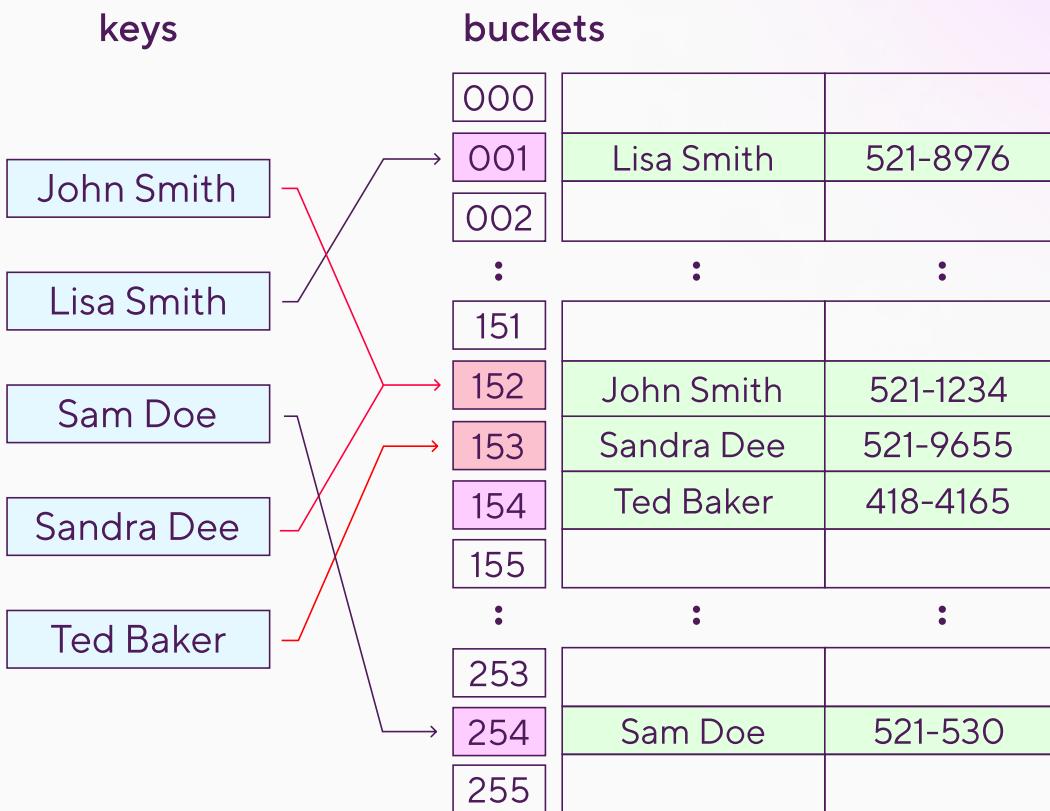
Бакеты

Какая-то структура данных содержащая в себе коллекцию из элементов с одинаковыми ключами, например список. Таким образом по хешу мы вычисляем индекс Бакета, в котором осуществляем поиск нужного нам ключа



Итерирование

Если мы вычисляем индекс по хешу объекта, а на этом месте в динамическом массиве оказывается другой ключ, мы вычисляем функцию от индекса который возвращает какой-то другой индекс и переходим к вычисленному индексу. Если со следующим индексом ситуация повторяется, мы применяем эту функцию вновь. Последний метод реализован в Python, что обеспечивает быструю работу с хеш-таблицами. А именно классами dict, то есть словарь, и set - множество. Свойства и методы этих классов мы обсудим в ближайших лекциях, а сейчас давайте резюмируем всё то, что мы обсудили на этом занятии



07

Множества

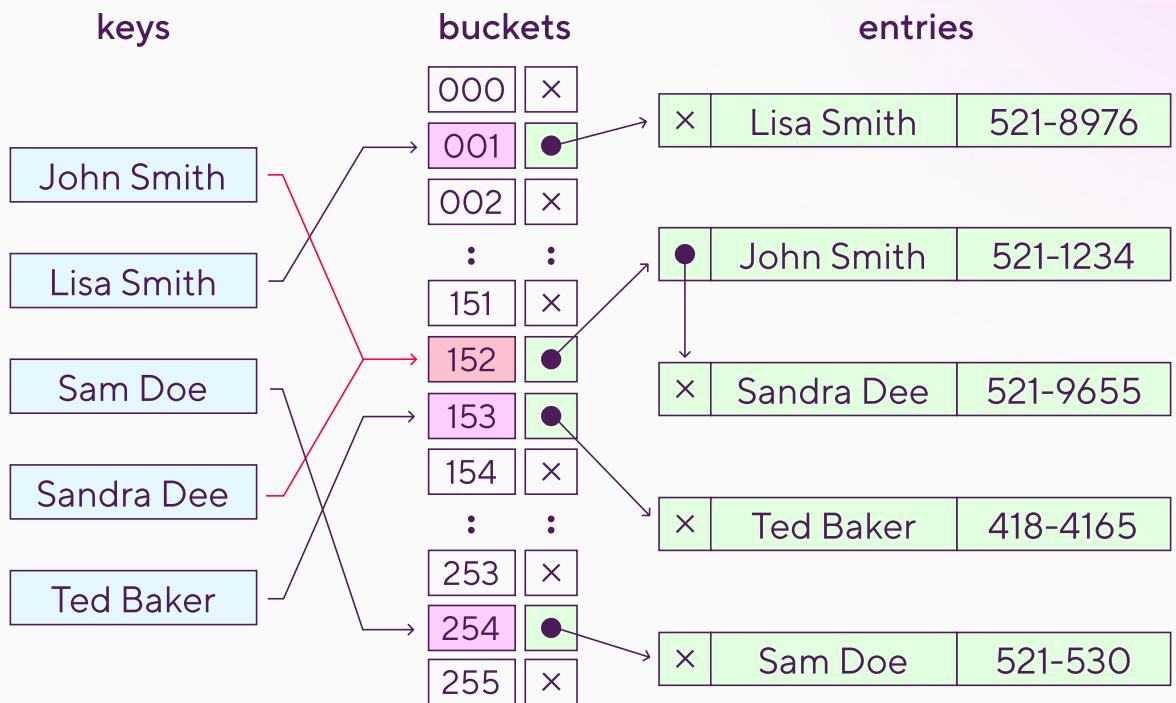
Множество

набор уникальных, то есть не повторяющихся, элементов. С этим понятием вы встречались ранее в школьной математике, в которой выделялось два основных типа множеств – непрерывные и дискретные. Примером непрерывного множества может служить отрезок или вся вещественная прямая

Дискретное множество

набор отдельных элементов. Класс `set` в Python является представлением дискретных множеств с конечным набором элементов

Множество в Python – изменяемый и итерируемый тип данных, который является коллекцией, построенной на хеш-таблицах. Хеш-таблица – это в каком-то смысле отображение ключей на данные. И что касается множеств в Python, то там ключи являются и данными. Поскольку ключи в хеш-таблице не повторяются, то это структура данных отлично подходит для реализации множества



Создание множества

Для того чтобы создать пустое множество, нужно какой-то переменной приравнять выражение `set` с круглыми скобками.
 Для того чтобы создать множество, содержащее уже какие-то элементы, необходимо эти элементы перечислить через запятую и заключить в фигурные скобки

Пример

```
...  

s = {3, 6, 9, -1, 2}  

for x in s:  

    print(x)
```

Вывод

```
...  

2  

3  

6  

9  

-1
```

Вы можете видеть пример кода, на котором мы ввели множество `S`, содержащее элементы 3, 6, 9, -1 и 2, а затем вывели каждый элемент с новой строки с помощью цикла `for`. Заметьте что числа отобразились не в том порядке, в котором они представлены при задании множества. Это обусловлено тем, что хеш-таблицы — это неупорядоченные структуры данных, поэтому элементы могут следовать не в том порядке, в котором мы их добавляли

Вывод в терминал количества элементов, содержащихся в множестве, происходит с помощью метода `len`

Пример

```
... |  
s = {3, 6, 9, -1, 2}  
print(len(s))
```

Вывод

```
... |  
5
```

Добавление элемента

Добавление элемента во множество происходит при помощи метода **add**

Пример

```
... |  
s = {4, 23, 9, -4, 6, 8}  
s.add(11)  
print(s)  
s.add(6)  
print(s)
```

Вывод

```
... |  
{4, 6, 23, 8, 9, 11, -4}  
{4, 6, 23, 8, 9, 11, -4}
```

В коде мы ввели множество S, состоящее из чисел 4, 23, 9, -4, 6 и 8. Затем добавили число 11 и число шесть с помощью метода add. Заметьте, что после добавления числа 6 множество не изменилось, поскольку число 6 уже содержалось в этом множестве

Удаление элемента

У множеств есть три метода для удаления элементов
- это методы **remove**, **discard** и **pop**

Пример

```
... |  
s = {4, 6, 23, 8, 9, 11, -4}  
s.remove(6)  
print(s)  
try:  
    s.remove(6)  
except KeyError:  
    print("there is no 6 in s")
```

Вывод

```
... |  
{4, 23, 8, 9, 11, -4}  
>>> there is no 6 in s
```

Метод **remove** удаляет элемент из множества или генерирует исключение **KeyError**, если такового элемента в множестве нет. В примере кода мы дважды удаляем элемент 6 с помощью метода **remove**. В первый раз мы действительно удаляем элемент, а во второй раз этот метод сгенерирует исключение **KeyError**

Пример

```
...  
s = {4, 23, 8, 9, 11, -4}  
s.discard(9)  
print(s)  
s.discard(9)  
print(s)
```

Вывод

...	
{4, 23, 8, 11, -4}	{4, 23, 8, 11, -4}

С помощью метода `discard` мы пытаемся дважды удалить элемент 9. В первый раз мы действительно его удаляем, а во второй просто ничего не происходит

Пример

```
...  
s = {4, 23, 8, 11, -4}  
print(s.pop())
```

Вывод

...	
{4, 23, 8, 11, -4}	{4, 23, 8, 11, -4}

Также для удаления элементов существует метод `pop`, который удаляет случайный элемент множества

Оператор `in`

Оператор `in` используется для проверки принадлежности элемента к множеству. Принцип такой же как и с другими коллекциями, но в случае с хеш-таблицами нам не нужно проходить по всем элементам коллекции – для того чтобы найти элемент, нужно лишь просто вычислить его хэш для вычисления индекса в таблице. Оператор `in` выполняет всю эту рутину за нас

Пример

```
...  
s = {4, 23, 8, 11, -4}  
print(s.pop())
```

Вывод

...	
True	False

Объединение множеств

Python также предоставляет нам некоторый набор математических операций над множествами. Рассмотрим объединение множеств

Пример

```
...  
a = {3, 4, 0, 11}  
b = {45, -3, 4, 11, 34}  
print(a | b)
```

Вывод

...	
{0, 34, 3, 4, 11, 45, -3}	

В примере мы ввели множество А состоящие из чисел 3 ,4, 0 и 11 а также множество В состоящее из чисел 45, -3, 4, 11 и 34. Далее мы получаем объединение множеств с помощью оператора `ripe` (или вертикальный слэш). Напоминаю что объединение множеств – это множество состоящее из элементов, которые входят в хотя бы в одно из этих множеств

Пересечение множеств

С помощью оператора амперсанд мы получаем пересечение множеств. Напоминаю, что пересечение множеств – это множество состоящее из элементов, которые входят в каждое из этих множеств

Пример

ooo	
a = {3, 4, 0, 11} b = {45, -3, 4, 11, 34} print(a & b)	

Вывод

ooo	
{11, 4}	

Сравнение множеств

К множеству можно применять операции сравнения, они работают по следующему принципу

Пример

ooo	
a = {3, 4, 0, 11} b = {45, -3, 4, 11, 34} print(a < b)	

Вывод

ooo	
False	

Выражение А меньше В, если А и В - множества, означает что А является подмножеством В и при этом не равно множеству В

Пример

ooo	
a = {3, 4, 0, 11} b = {45, -3, 4, 11, 34} print(a <= b)	

Вывод

ooo	
False	

Выражение А меньше либо равно В означает, что А является подмножеством В и возможно А равно В. Равенство множеств означает, что все элементы множества а входят в множество В и все элементы множества В входят в А. Со знаками больше и больше либо равно всё аналогично

Пример

```
...  
a = {3, 4, 0, 11}  
b = {45, -3, 4, 11, 34}  
print(a - b)
```

Вывод

...	
{0, 3}	

Выражение A - B возвращает множество элементов A, которые не являются элементами B

С другими методами и операциями вы можете познакомиться на python.org. А теперь предлагаю обсудить какие типы данных можно использовать в качестве ключей в множествах

08

Словари

Мап

абстрактная структура данных, которая хранит в себе данные в виде пар ключ значение, где каждый ключ уникален

Иногда map называет ещё ассоциативным массивом или словарём. Последнее название и используется в Python. Поскольку эта структура данных абстрактная, то она может иметь множество реализаций. В Python она реализована на хеш-таблицах. Действительно, хеш-таблицы отлично подходят под реализацию этой структуры, поскольку позволяет быстро осуществлять поиск данных по ключу, их модификацию, а также удаление пары ключ-значение

Словарь в Python

это изменяемый тип данных, который является итерируемой по ключу коллекцией

Поскольку и словари, и множества реализованы на хеш-таблицах, то они имеют очень много общего, что мы будем периодически отмечать это на лекции, не обходя стороной и различия

Создание словаря

Для того чтобы создать словарь, заполненный какими-то начальными данными, необходимо пары ключ-значение разделить запятыми и заключить всё в фигурные скобки. Внутри каждой пары ключ и значение разделены двоеточием

Пример

```
...  
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

Для создания пустого словаря необходимо лишь поставить пару фигурных скобок

Пример

```
...  
my_dict = {}
```

Сейчас вы можете видеть демонстрацию создания пустого и непустого словарей. Заметьте, что множества вводятся тоже с помощью фигурных скобок, но поскольку в множествах просто ключи, а в словарях пары ключ-значение, разделённые двоеточием, то проблем с неоднозначностью не возникает в случае непустых множеств и словарей. А с пустыми возникает. Именно поэтому пустое множество вводится с помощью функции конструктора `set` с круглыми скобками. А пустой словарь определяется просто с помощью двух фигурных скобок

Перебор элементов

Поскольку словарь объект итерируемый, то по нему можно проходить с помощью цикла `for`

Пример

```
...  
  
names_ages = {  
    'Peter': 12,  
    'Anna': 14,  
    'Steven': 10,  
    'Jack': 27,  
    'Garry': 20,  
    'Bob': 14,  
    'Alice': 27,  
    'Elizabeth': 14,  
    'Amelia': 30,  
}  
  
for x in names_ages:  
    print(x)
```

Вывод

```
...  
  
Peter  
Anna  
Steven  
Jack  
Garry  
Bob  
Alice  
Elizabeth  
Amelia
```

Мы проходимся по словарю `names_ages`, который содержит пары имя-возраст. Но заметьте, что в таком случае мы пробегаем только по ключам, поскольку словари итерируемые именно по ключу

Оператор `in`

Проверка принадлежности ключу словарю осуществляется с помощью оператора `in` так же как и с множествами

Пример

```
...  
names_ages = {  
    'Peter': 12,  
    'Anna': 14,  
    'Steven': 10,  
    'Jack': 27,  
    'Garry': 20,  
    'Bob': 14,  
    'Alice': 27,  
    'Elizabeth': 14,  
    'Amelia': 30}  
print('Peter' in names_ages)
```

Вывод

```
...  
True
```

Метод get

Метод `get()` в словаре используется для получения значения по ключу. Он принимает два аргумента: ключ и необязательное значение по умолчанию. Если ключ найден в словаре, то метод `get()` возвращает соответствующее значение. Если ключ не найден, то возвращается значение, указанное вторым аргументом (если он указан), или значение `None`, если второй аргумент не указан

Пример

```
...  
student = {'name': 'John', 'age': 20, 'grade': 'A'}  
# Получение значения по ключу  
print(student.get('name')) # Вывод: John  
print(student.get('grade')) # Вывод: A  
# Получение значения по несуществующему ключу  
print(student.get('gender')) # Вывод: None  
# Получение значения по несуществующему ключу с указанием значения по умолчанию  
print(student.get('gender', 'Unknown')) # Вывод: Unknown
```

Метод `get()` намного безопаснее, чем обращение к словарю напрямую с помощью оператора `[]`, потому что не вызывает ошибки `KeyError`, если ключ не найден в словаре

Объединение словарей

Для объединения двух или более словарей в Python можно использовать несколько подходов

01 Оператор распаковки словаря **

```
...  
dict1 = {'a': 1, 'b': 2}  
dict2 = {'c': 3, 'd': 4}  
combined_dict = {**dict1, **dict2}  
print(combined_dict) # Вывод: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

02 Метод update()

```
...  
dict1 = {'a': 1, 'b': 2}  
dict2 = {'c': 3, 'd': 4}  
  
dict1.update(dict2)  
print(dict1) # Вывод: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

03 Метод copy() и оператор update()

```
...  
dict1 = {'a': 1, 'b': 2}  
dict2 = {'c': 3, 'd': 4}  
  
combined_dict = dict1.copy()  
combined_dict.update(dict2)  
print(combined_dict) # Вывод: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

04 Метод copy() и оператор update()

```
...  
dict1 = {'a': 1, 'b': 2}  
dict2 = {'c': 3, 'd': 4}  
dict3 = {'e': 5, 'f': 6}  
combined_dict = {}  
combined_dict.update(dict1, **dict2, **dict3)  
print(combined_dict) # Вывод: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
```

Во всех этих подходах объединение словарей означает добавление всех ключей и значений из одного словаря в другой. Если ключи повторяются, значение из второго словаря заменяет значение в первом словаре.

Итог

В данной теме мы изучили типы данных, познакомились с различными структурами данных, таким как: списки, кортежи, строки, множества, словари. Разобрали методы работы с этими структурами. Познакомились с понятием хешируемости