

Списки (Массивы)

01

Списки, методы списков, срезы

Срезы в Python

Срезы в Python позволяют извлекать подстроки, подсписки или другие срезы из последовательностей, таких как строки, списки и кортежи.

Срезы определяются с использованием оператора [] с указанием начального индекса, конечного индекса и (необязательно) шага

Синтаксис

...	
	[start:stop:step]

Где

- **Start** - индекс элемента, с которого начинается срез (включительно). Если не указан, используется начало последовательности (индекс 0)
- **Stop** - индекс элемента, на котором заканчивается срез (исключая сам этот элемент). Если не указан, используется конец последовательности
- **Step** - определяет шаг среза (необязательно). Положительные значения перечисляют элементы слева направо, отрицательные - справа налево. Если не указан, используется шаг 1

Примеры

```
...
text = "Hello, World!"
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(text[7:12]) # Вывод: World
print(numbers[2:6]) # Вывод: [3, 4, 5, 6]

print(text[:5]) # Вывод: Hello
print(numbers[3:]) # Вывод: [4, 5, 6, 7, 8, 9, 10]

print(text[::-2]) # Вывод: Hlo ol!
print(numbers[::-1]) # Вывод: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

! Важно отметить, что срез не изменяет исходную последовательность, а создает новый объект с выбранными элементами

Метод split()

Метод join()

Метод join() в Python используется для объединения элементов последовательности в одну строку. Он принимает последовательность в качестве аргумента и возвращает новую строку, которая состоит из элементов последовательности, разделенных заданным разделителем

Синтаксис

```
... |  
separator.join(sequence)
```

Где

- **Separator** - строка, которая будет использована в качестве разделителя между элементами последовательности
- **Sequence** - последовательность, элементы которой нужно объединить в строку

Примеры

```
... |  
  
words = ['Hello', 'World']  
sentence = ' '.join(words) # Объединение слов через пробел  
print(sentence) # Вывод: "Hello World"  
  
numbers = ['1', '2', '3', '4', '5']  
num_string = '-'.join(numbers) # Объединение чисел через дефис  
print(num_string) # Вывод: "1-2-3-4-5"
```

Метод join() полезен, когда необходимо объединить элементы последовательности в строку, используя определенный разделитель. Он обратный по отношению к методу split()

02

Генераторы списков

Генераторы в Python

специальный тип функций, которые позволяют создавать итерируемые последовательности значений. Они позволяют создавать элементы по мере необходимости, вместо создания всех элементов сразу, что может быть полезно для работы с большими наборами данных или для работы с бесконечными последовательностями

Генераторы определяются с использованием ключевого слова `yield` внутри функции. Когда функция вызывается, она возвращает генератор-объект, который может быть использован для итерации по последовательности значений, вырабатываемых генератором

Пример

```
...  
def square_generator(n):  
    for i in range(1, n+1):  
        yield i*i  
  
# Использование генератора  
gen = square_generator(5)  
for value in gen:  
    print(value)
```

Вывод

```
...  
1  
4  
9  
16  
25
```

Этот код будет выводить квадраты чисел от 1 до 5

Главное преимущество генераторов заключается в их эффективности и экономии памяти. Они генерируют значения по мере необходимости, что позволяет обрабатывать большие объемы данных без загрузки их в память целиком

Помимо ключевого слова `yield`, в Python также есть выражение-генератор, которое может быть использовано для создания генератора. Это более компактная форма определения генератора, обеспечивающая ещё большую гибкость и экономию времени

Пример

```
...  
gen = (x*x for x in range(1, 6))  
for value in gen:  
    print(value)
```

Пример выше приведет к тому же выводу

03

Алгоритмы сортировки

Сортировка

процесс упорядочивания элементов в заданном порядке. В Python существует несколько способов сортировки данных

Метод sort() для списков

Метод `sort()` используется для сортировки элементов списка прямо на месте. Он меняет порядок элементов в исходном списке

Примеры

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

Функция sorted()

Функция `sorted()` используется для сортировки элементов любой последовательности и возвращает новый список с отсортированными значениями

Примеры

```
...
my_tuple = (3, 1, 4, 2, 5)
sorted_tuple = sorted(my_tuple)
print(sorted_tuple) # Вывод: [1, 2, 3, 4, 5]
```

Параметр key

Оба метод `sort()` и функция `sorted()` также принимают необязательный параметр `key`, который позволяет задать функцию сравнения для определения порядка сортировки

Примеры

```
...
my_list = ['apple', 'banana', 'cherry', 'date']
my_list.sort(key=len)
print(my_list) # Вывод: ['date', 'apple', 'banana', 'cherry']
```

В приведенном примере, для сортировки списка используется длина каждой строки. Это позволяет отсортировать список по возрастанию длины строк

Параметр reverse

Оба метод `sort()` и функция `sorted()` также имеют необязательный параметр `reverse`, который указывает, следует ли сортировать элементы в обратном порядке

Примеры

```
...  
my_list = [3, 1, 4, 2, 5]  
my_list.sort(reverse=True)  
print(my_list) # Вывод: [5, 4, 3, 2, 1]
```

В приведенном примере, список отсортирован в обратном порядке

Все эти методы и функции сортировки могут быть применены к различным типам последовательностей и имеют различные параметры для достижения желаемого результата

Сложность алгоритма

Определение сложности алгоритма важно для анализа его производительности и эффективности. Сложность алгоритма описывает, как меняется количество операций (время выполнения) или используемая память в зависимости от размера входных данных

Существуют два основных типа сложности алгоритмов:

01 Некоторые общие классы временной сложности алгоритмов

- $O(1)$ - постоянная сложность, время выполнения не зависит от размера входных данных
- $O(\log n)$ - логарифмическая сложность, время выполнения растет в логарифмической пропорции к размеру входных данных
- $O(n)$ - линейная сложность, время выполнения растет линейно с размером входных данных
- $O(n^2)$ - квадратичная сложность, время выполнения растет квадратично от размера входных данных
- $O(2^n)$ - экспоненциальная сложность, время выполнения растет экспоненциально с размером входных данных

02 Пространственная сложность

Пространственная сложность определяет, сколько дополнительной памяти или ресурсов требуется для выполнения алгоритма. Она измеряется как количество дополнительной памяти, используемой алгоритмом, по отношению к размеру входных данных

Примеры классов пространственной сложности алгоритмов

- **O(1)** - постоянная сложность, алгоритм не требует дополнительной памяти
- **O(n)** - линейная сложность, алгоритм требует дополнительную память пропорционально размеру входных данных
- **O(n^2)** - квадратичная сложность, алгоритм требует дополнительную память, пропорциональную квадрату размера входных данных

Определение сложности алгоритма помогает выбирать наиболее эффективные алгоритмы для конкретных задач и предсказывать их производительность при изменении размера входных данных. Более эффективные алгоритмы могут значительно сократить время выполнения или использование памяти по сравнению с менее эффективными алгоритмами

Итог

В данной теме мы познакомились с различными методами работы со списками и строками: срезы, методы `split()` и `join()`. Рассмотрели несколько вариантов сортировки элементов списка, а также разобрали такое понятие, как сложность алгоритма