22. Конспект

Конспект поможет вам вспомнить информацию, которая была в уроке, пригодится для дальнейшего обучения и работы.

Что такое списки

Списки — новый тип данных, а скорее даже агрегат типов данных, куда мы можем складывать разные значения и потом работать с ними по индексам и срезам.

Список — это тип данных list, представляющий собой упорядоченный набор элементов.

Примеры того, как создавать списки:

```
animals = ["deer", "bear", "rabbit", "butterfly"]
numbers = [5, 4, 3, 2, 1, 0]
flags = [True, True, False, True]
```

Чтобы распечатать список, достаточно записать:

```
print(animals)
```

Индексы элементов списка

У каждого элемента списка есть **индекс** — порядковый номер.

Нумерация элементов начинается с нуля, т. е. индекс первого элемента будет равен 0:

```
['P', 'y', 't', 'h', 'o', 'n']
# 0    1    2    3    4    5
```

Запомнить это легко: достаточно представить, что речь идет о количестве шагов, которые надо пройти от начала, чтобы увидеть объект.

Индекс записывается в квадратных скобках.

Так можно получить элемент по номеру:

```
hp_books = ["Философский Камень", "Тайная Комната", "Узник Азкабана"]
```

```
print(hp_books[2])
>>> "Узник Азкабана"
kings = ['Генрих', 'Людовик', 'Фридрих', 'Ричард']
pos = 3
print(kings[pos])
>>>  Ричард
```

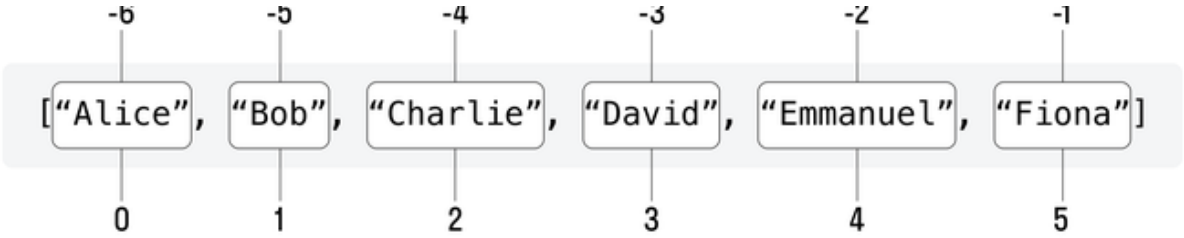
Мы можем заменить элемент в списке, если знаем индекс:

```
flock = ['sheep', 'sheep', 'sheep', 'sheep']
flock[2] = 'wolf'
print(flock)
>>> ['sheep', 'sheep', 'wolf', 'sheep']
```

Отрицательные индексы

Кроме обычных индексов, для элементов списка используются **отрицательные индексы**. Они нужны для того, чтобы получать элементы по их номеру с конца списка.

Обратите внимание: так как числа «минус ноль» не существует, **нумерация начинается от «минус первого» элемента**.



Рассмотрим на примере. У нас есть список:

```
guests = ["Alice", "Bob", "Charlie", "David", "Emmanuel",
"Fiona"]
```

Обратимся к элементам по отрицательному индексу:

```
guests[-1]  # Вернет "Fiona"
guests[-3]  # Вернет "David"
guests[-5]  # Вернет "Bob"
```

Отрицательные индексы можно использовать вместе с переменными:

```
guests = ["Alice", "Bob", "Charlie", "David", "Emmanuel",
"Fiona"]

last = -1
last_but_one = -2

guests[last]  # Вернет "Fiona"
```

```
guests[last_but_one]  # Вернет "Emmanuel"
```

Функция len

Чтобы измерить длину списка, воспользуемся функцией len:

```
list_len = len(["декабрь", "январь", "февраль"])
print(list_len)
>>> 3
```

Еще один пример использования функции:

```
mylist = ['P', 'y', 't', 'h', 'o', 'n']
list_len = len(mylist)
print(list_len)
>>> 6
```

Длина списка и индекс последнего элемента всегда разные и отличаются на единицу!

```
['P', 'y', 't', 'h', 'o', 'n']
# 0    1    2    3    4    5
```

Метод append

Позволяет добавить элемент в конец списка:

```
имя_списка.append(элемент)
```

Элементом может быть число, строка или другой список.

Пример:

```
brhd_ring = ["Арагорн", "Фродо", "Гэндальф", "Гимли"]
brhd_ring.append("Леголас")
print(brhd_ring)
>>> ["Арагорн", "Фродо", "Гэндальф", "Гимли", "Леголас"]
```

Метод extend

Позволяет добавлять множество элементов в конец списка:

```
brhd_ring = ["Арагорн", "Фродо", "Гэндальф", "Гимли"]
brhd_ring.extend(["Леголас", "Боромир"])
print(brhd_ring)
>>> ["Арагорн", "Фродо", "Гэндальф", "Гимли", "Леголас",
"Боромир"]
```

Обратите внимание, что множество элементов записывается в квадратных скобках:

список.extend([элемент1, элемент2])

Срезы

Срезы — это «подсписки» внутри списка, т. е. вырезанная часть списка.

Пример работы со срезами:

```
drinks = ['вода', 'чай', 'кофе', 'какао']

drinks[1:3]
# Вернет ['чай', 'кофе']

drinks[2:]
# Вернет ['кофе', 'какао']
```

Рассмотрим еще примеры использования срезов.

Предположим, у нас есть список:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

В таблице показано, какие срезы возможно применить к списку и какой

результат получится.
Срез**Расшифровка****Вхождение****Результат**letters[:]Все элементыABCDEFGH['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']letters[**3:6**]С элемента 3 до 6 (не включая его)

ABCDEFGH
['D', 'E', 'F']letters[**3:**]Начиная с элемента 3

ABCDEFGH
['D', 'E', 'F', 'G', 'H']letters[**0:6**]До элемента 6 (не включая его)

ABCDEFGH
['A', 'B', 'C', 'D', 'E', 'F']letters[**:6**]До элемента 6 (короткая запись)

ABCDEFGH
['A', 'B', 'C', 'D', 'E', 'F']letters[**0:-1**]Все, кроме последнего

ABCDEFGH
['A', 'B', 'C', 'D', 'E', 'F', 'G']letters[**0:-3**]Все, кроме 3 последних

ABCDEFGH
['A', 'B', 'C', 'D', 'E']letters[**-3:**]Три последних

ABCDEFGH
['F', 'G', 'H']

Срезы и переменные

Вместо числовых значений мы можем использовать **переменные**.

Например, такой код выведет первые 3 элемента:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

pos = 3

slices = letters[:pos]

print(slices)
>>> ['A', 'B', 'C']
```

Возможно использовать сразу **две переменные**:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

start = 2
end = 5

slices = letters[start:end]

print(slices)
>>> ['C', 'D', 'E']
```

Можно использовать **стандартный ввод** для срезов.

Такой код выведет все элементы после элемента с индексом number:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
number = int(input())

slices = letters[number+1:]

print(slices)
```

В Python существуют два основных способа удаления элементов из списка:

1. По **индексу** — когда мы знаем порядковый номер элемента, с помощью оператора del и метода pop.
2. По **значению** — когда мы знаем, что хотим удалить, но не знаем, на какой позиции элемент находится, с помощью метода remove.

Метод remove

Самый простой способ удалить элемент по значению — использовать метод remove().

remove — это метод, а не функция. Поэтому при работе с методом укажите список, затем поставьте точку, напишите remove и в скобках — значение элемента, который хотите удалить.

Пример 1:
```
numbers = [1,2,3]
numbers.remove(2)
print(numbers)
>>> [1, 3]
```

Пример 2:
```
letters = ["a", "b", "c"]
letters.remove("a")
print(letters)
>>> ['b', 'c']
```

Удаляйте только те элементы, которые есть в списке.

Если указанного элемента не существует, интерпретатор выведет ошибку:
```
numbers = [1,2,3]
numbers.remove(5)
```

>>> ValueError: list.remove(x): x not in list

Оператор del

Универсальный оператор, который удаляет переменные любого типа, элементы из списка, а также из тех типов, которые мы будем изучать в будущем: кортежей, словарей и множеств.

Чтобы удалить элемент из списка, мы можем передать оператору del ссылку на определенный элемент, например, получив его по индексу. Обратите внимание, **del — это оператор**. Значит, удаляемый объект мы указываем через пробел — никаких круглых скобок!

Пример 1:
```
numbers = [1,2,3]

del numbers[0]
print(numbers)
>>> [2, 3]
```

Пример 2:
```
letters = ["a", "b", "c"]
del letters[2]
print(letters)
>>> ['a', 'b']
```

Метод pop

Еще один способ удалить элемент, зная его индекс, — метод pop.

Особенности работы метода pop:

1. Кроме удаления элемента по индексу, метод pop возвращает его.

   Вот как это работает:
   ```
   numbers = [1,2,3]

   removed = numbers.pop(0)

   print(removed)

   print(numbers)
   ```

   Выведет:
   ```
   1

   [2, 3]
   ```

2. Если вообще не передавать аргумент этому методу, он удалит последний элемент и вернет его значение.

   Это пригодится нам в будущем при реализации некоторых алгоритмов.

   Пример:
   ```
   numbers = [1,2,3]

   removed = numbers.pop()

   print(removed)

   print(numbers)
   ```

   Выведет:

```
3

[1, 2]
```

Обращаем внимание: если указать несуществующий индекс, то получим сообщение об ошибке:

```
numbers = [1,2,3]
removed = numbers.pop(9)
```

>>> IndexError: pop index out of range

Оператор in

Вы уже сталкивались с оператором **in**, когда знакомились с операторами сравнения на строках.

Оператор in применяется для списков и позволяет проверять вхождение в список элемента, однако принцип работы отличается от строк — мы всегда проверяем **вхождение только одного элемента** и **значение должно совпадать с точностью**.

Примеры:

```
"a" in ["a", "b", "c"]
#  Вернет True

"x" in ["a", "b", "c"]
#  Вернет False
1 in [1, 2, 3]
#  Вернет True

9 in [1, 2 ,3]
#  Вернет False
```

Важные особенности при работе с оператором in

- Когда вы работаете со списками, оператор in **проверяет вхождение целого элемента в список**, а не вхождение подстроки в строку. True вернется только тогда, когда элемент, который мы ищем, содержится в списке целиком.

Например:

```
"a" in ["a", "b", "c"] # Вернет True


"a" in ["aa", "ab", "ac"] # Вернет False
```

- Для оператора сравнения in **важен тип данных — число или строка:**

```
11 in [11, 12, 13] # Вернет True

11 in ["11", "12", "13"]  # Вернет False

"11" in [11, 12, 13] # Вернет False
```

Однако если значения имеют **числовое равенство**, сравнение вернет True.

Пример:

```
1.0 in [0, 1] # Вернет True


1 in [1.0, 0.0] # Вернет True
```

Поскольку внутреннее представление **bool** — число, это также верно для bool:

```
0 in [True, False] # Вернет True


True in [0, 1] # Вернет True
```

При работе со строками помните, что **a** и **A** — это разные значения!

Пример:

```
"a" in ["a", "b", "c"] # Вернет True


"A" in ["a", "b", "c"] # Вернет False
```