Массивы

22. Списки — list

Списки (коллекции) данных и взаимодействие с ними.

Урок

В этом уроке мы разберем один из универсальных и часто используемых типов данных в Python — списки. Списки используются практически во всех предметных областях —- написании игр, создании сайтов, анализе данных и так далее.

Например, в списках можно хранить имена авторизованных пользователей, идентификаторы корзины покупателя или историю заказов. Все эти и многие другие списки можно создать и изменить (добавить или удалить элемент), так как списки являются изменяемыми типами данных в Python.

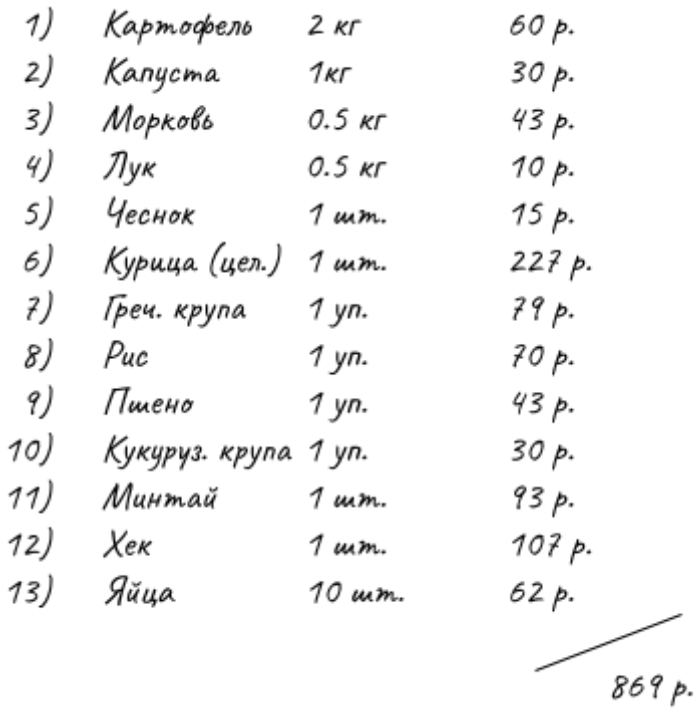
Цель урока
Научиться создавать и использовать списки.

1. Переход от строк к спискам

Списки в Python — это один из универсальных и часто используемых типов данных, который позволяет нам работать с несколькими объектами одновременно.

Строки близки к спискам с точки зрения понимания коллекционирования значений. Вот только в списках могут храниться совершенно разнородные данные любого типа и в любой последовательности. Например, перечисление выигрышных номеров билетов в Бинго или перечень ингредиентов для супа. (В последнем примере не хватает, конечно, информации о дозировке. Но это относится уже к более сложным *вложенным структурам*, о которых мы поговорим позже.)

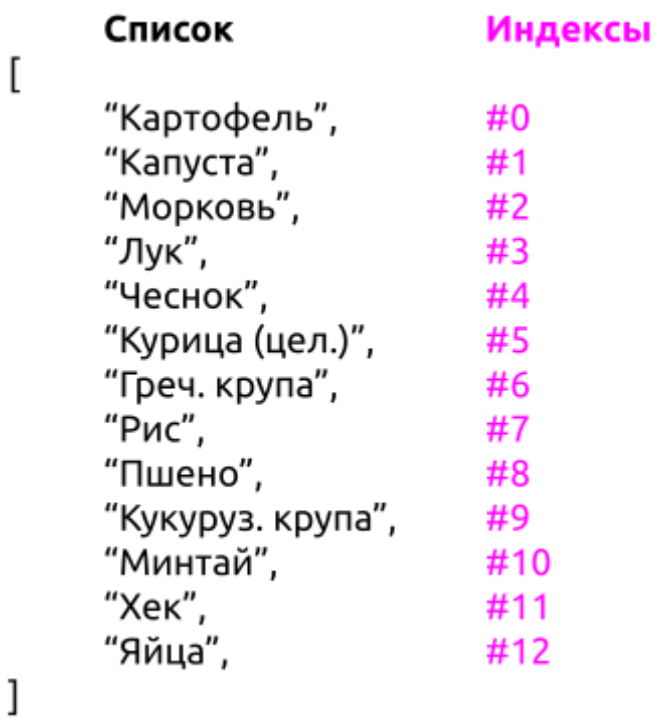
Списки как структуры данных будут полезны во многих случаях. Например, мы собираемся в магазин и составляем список покупок:



Этот список будет обладать теми же свойствами, что и строки: он упорядочен, каждый товар в нем имеет свой порядковый номер.

Теория

Так выглядит список покупок, например, на бумаге, но его можно сохранить в понятном для Python формате. Можно создать переменную особого типа, которая будет содержать в себе не один элемент (например, одно число или одну строку), а сразу список элементов или объектов. Такой тип данных называется **список** (**list**). Например, список наименований покупок будет выглядеть так:



В результате у нас получился список из тех же 13 элементов, хотя его длина условно безгранична.

Ранее мы говорили о том, что строки — это последовательности символов. Символы в строке упорядочены, и мы можем обратиться к каждому символу строки по его индексу:

example-26.1.1.py	
код	`string = "Python"` `print(string[2], string[4], string[5])`
вывод	`t o n`

К слову, подобную строку можно представить в виде списка, и результат обращения к элементам списка будет таким же:

example-26.1.2.py	
код	`list = ["P", "y", "t", "h", "o", "n"]` `print(list[2], list[4], list[5])`
вывод	`t o n`

К каждому отдельному элементу списка, так же, как и в случае строк, можно обращаться по **индексу** — числовому значению порядкового номера для элемента.

Важно!
Нумерация индексов списка начинается с **0**, а не с **1**.

Так же, как и строки, списки поддерживают **отрицательное индексирование**. То есть мы можем обращаться к элементам списка, считая не от начала списка, а от конца. Например, посмотрим на последний элемент списка **list** из предыдущего примера:

example-26.1.3.py

код
```
list = ["P", "y", "t", "h", "o", "n"]
print(list[2], list[4], list[-1])
```

вывод
```
t o n
```

И так же, как и в случае со строками, мы можем использовать **срезы** (**слайсы**) для того, чтобы получить часть списка:

example-26.1.4.py

код
```
list = ["P", "y", "t", "h", "o", "n"]

list_1 = list[2:3]
list_2 = list[4:6]
print(list_1, list_2)

print(list_1[0], list_2[0], list_2[1])
```

вывод
```
['t'] ['o', 'n']
t o n
```

В общем виде формат срезов выглядит следующим образом:

<tuple_name>**[** <*x*> **:** <*y*> **:** <*z*> **]**

где:

- x : int — индекс начала (граница включается);
- y : int — индекс конца (граница исключается);
- z : int — частота (шаг) выбора элементов.

Примечание
В результате среза из списка получается новый список, с которым можно работать, как с обычным списком.

Для того чтобы узнать длину списка, мы можем использовать функцию **len()**. Эта функция, примененная к списку, вернет количество элементов в этом списке:

example-26.1.5.py	
код	`list = ["P", "y", "t", "h", "o", "n"]` `print(len(list))`
вывод	`6`

Примечание
Если список пустой, то есть содержит **0** элементов, то функция **len()** вернет значение **0**.

2. Создание и изменение списков

Как и любой объект, список начинается с его создания. Указывая начальные значения списка, мы его **инициализируем**. Например, у нас всегда есть список продуктов по умолчанию. Но когда нам надо что-то добавить, удалить или заменить какой-то элемент — это и будет изменением списка.

Теория

Списки в Python создаются с помощью квадратных скобок, внутрь которых через запятую помещаются элементы (объекты), которые необходимо хранить в этом списке. Создадим список покупок:

example-26.2.1.py	
код	`shoplist = ["яблоки", "молоко", "говядина", "яйца"]` `print(shoplist)`
вывод	`['яблоки', 'молоко', 'говядина', 'яйца']`

В качестве элементов списка не обязательно должны быть строки. Списки могут содержать элементы любых типов — целые числа, числа с плавающей точкой, булевы

значения, даже другие списки. Так же, как и строка, список может быть пустым, то есть не содержать никаких элементов:

example-26.2.2.py	
код	`empty_list = []` `print(empty_list)`
вывод	`[]`

Список может состоять не только из явных значений, но и из переменных. Создадим, например, три переменные и соберем из них список:

example-26.2.3.py	

код	`a = 5`
	`b = 10`
	`c = 15`
	`lst = [a, b, c]`
	`print(lst)`

вывод	`[5, 10, 15]`

А теперь создадим список из булевых значений:

example-26.2.4.py	
код	`lst = [True, False]` `print(lst)`
вывод	`[True, False]`

Также списки можно создавать из чисел разных типов:

example-26.2.5.py	
код	`lst = [0, 1.0, 2, 3.4]` `print(lst)`
вывод	`[0, 1.0, 2, 3.4]`

Важно!
Элементы в списке разделяются запятой (**,**) а не точкой (**.**).

В отличие от строк, списки являются *изменяемыми* объектами, а значит, мы можем редактировать список множеством способов. Давайте их рассмотрим.

Добавление элементов в конец списка

Начнем с добавления элементов в списки. Предположим, мы хотим добавить сыр к списку покупок. Чтобы добавить объект в *конец* списка, вызовем у списка метод **.append()**:

example-26.2.6.py	

код	`shoplist = ["яблоки", "молоко", "говядина", "яйца"]`
	`print(shoplist)`
	
	`shoplist.append("сыр")`
	`print(shoplist)`

вывод	`['яблоки', 'молоко', 'говядина', 'яйца']` `['яблоки', 'молоко', 'говядина', 'яйца', 'сыр']`

Добавление элементов внутрь списка

Мы можем добавить элемент на *любую позицию* в списке. Для этого используется другой метод списка под названием **.insert()**. Он принимает на вход два параметра:

1) индекс или позиция списка, начиная с которой нужно вставить новые элементы;
2) элементы, которые нужно вставить.

В этом случае все элементы списка, находящиеся правее от указанной позиции, сдвинутся вправо. Давайте, например, добавим в список покупок бананы сразу после яблок, то есть на позицию с индексом 1:

example-26.2.7.py	

код	`shoplist = ["яблоки", "молоко", "говядина", "яйца"]`
	`print(shoplist)`
	
	`shoplist.append("сыр")`
	`print(shoplist)`

	shoplist.insert(1, "бананы") print(shoplist)
вывод	['яблоки', 'молоко', 'говядина', 'яйца'] ['яблоки', 'молоко', 'говядина', 'яйца', 'сыр'] ['яблоки', 'бананы', 'молоко', 'говядина', 'яйца', 'сыр']

Удаление элемента по индексу

Продолжим изменять список и теперь удалим элемент. Для удаления элемента по индексу используется метод **.pop()**, который принимает один *необязательный* аргумент — индекс удаляемого элемента и возвращает значение удаляемого элемента. Удалим из списка покупок первый элемент, то есть элемент с индексом 0:

example-26.2.8.py	
код	shoplist = ["яблоки", "бананы", "молоко", "говядина", "яйца", "сыр"] print(shoplist) first = shoplist.pop(0) print(shoplist) print(first)
вывод	['яблоки', 'бананы', 'молоко', 'говядина', 'яйца', 'сыр'] ['бананы', 'молоко', 'говядина', 'яйца', 'сыр'] яблоки

Если значение аргумента в метод **.pop()** не передано, то метод удаляет последний элемент списка, то есть по умолчанию значение индекса равно последнему в списке:

example-26.2.9.py	
код	shoplist = ["бананы", "молоко", "говядина", "яйца", "сыр"] print(shoplist) shoplist.pop() print(shoplist)
вывод	['бананы', 'молоко', 'говядина', 'яйца', 'сыр']

	`['бананы', 'молоко', 'говядина', 'яйца']`

Удаление элемента по значению

Для удаления элемента по значению используется метод **.remove()**, который принимает на вход один обязательный параметр — значение элемента, который нужно удалить:

example-26.2.10.py	

код	`shoplist = ["бананы", "молоко", "говядина", "яйца"]`
	`shoplist.remove("яйца")`
	`print(shoplist)`

вывод	`['бананы', 'молоко', 'говядина']`

Важно!
Если в списке нет удаляемого элемента, переданного в **.remove()**, будет выведена ошибка: ValueError: list.remove(x): x not in list.

Примечание
Метод **.remove()** удаляет только *первое вхождение* элемента в списке.

Изменение значения по индексу

Для того чтобы изменить любой элемент списка, нам нужно обратиться к нему по индексу и присвоить новое значение. Например, давайте в списке покупок вместо **молоко** напишем **сливки**:

example-26.2.11.py	

код	`shoplist = ["бананы", "молоко", "говядина"]`
	`shoplist[1] = "сливки"`
	`print(shoplist)`

вывод	`['бананы', 'сливки', 'говядина']`

3. Перебор списков с элементами разных типов

Основное применение списков в языке Python — это последовательный перебор элементов списка и их обработка, например: поиск минимальной закупочной цены по товару, сортировка товара по количеству, фильтрация товара по наличию и так далее. Создавать и изменять списки поэлементно мы уже научились, приступим к перебору.

Теория

Давайте на примере списка **shoplist** рассмотрим основные конструкции перебора списка. В Python для последовательного перебора элементов списка чаще всего используются два типа конструкций:

- **for <value> in <list>**;
- **for <index> in range(len(<list>))**.

Конструкция **for <value> in <list>** самая распространенная для перебора списков. В этом случае мы перебираем элементы списка по порядку от начала до конца и обрабатываем их, например выводим на экран или сохраняем в *другой* список:

example-26.3.1.py

код

```
newlist = []
shoplist = ["яблоки", "бананы", "молоко", "говядина", "яйца"]

for elem in shoplist:
    print(elem)
    newlist.append(elem)

print(newlist)
```

вывод

```
яблоки
бананы
молоко
говядина
яйца
['яблоки', 'бананы', 'молоко', 'говядина', 'яйца']
```

Конструкция **for <index> in range(len(<list>))** чаще всего используется, когда нам нужно изменять элементы списка и сохранять их в этот же список. В этом случае мы в цикле перебираем индексы всех элементов списка, чтобы затем использовать эти индексы, например, для изменения элементов:

example-26.3.2.py	

код	`shoplist = ["яблоки", "бананы", "молоко", "говядина", "яйца"]`
	
	`for i in range(len(shoplist)):`
	`    print(i, " - ", shoplist[i])`

вывод	`0  -  яблоки`
	`1  -  бананы`
	`2  -  молоко`
	`3  -  говядина`
	`4  -  яйца`

Мы можем комбинировать цикл перебора списка с условными конструкциями, чтобы обрабатывать не все элементы списка, а только те, которые удовлетворяют какому-то условию для фильтрации. Например, давайте в список **newlist** сохраним только те элементы списка **shoplist**, которые заканчиваются буквой **a**:

example-26.3.3.py	

код	`newlist = []`
	`shoplist = ["яблоки", "бананы", "молоко", "говядина", "яйца"]`
	
	`for elem in shoplist:`
	`    if elem[-1] == "a":`
	`        newlist.append(elem)`
	
	`print(newlist)`

вывод	`['говядина', 'яйца']`

Списки в Python — очень гибкий тип данных, позволяющий хранить объекты любых типов: числа, строки, другие списки и так далее, причем в одном списке могут одновременно храниться элементы разных типов. Следующий пример — это полностью рабочий код на Python:

example-26.3.4.py	
код	`lst = [2, 6.5, "банан", True, 11]` `print(lst)`

вывод	`[2, 6.5, 'банан', True, 11]`

Однако несмотря на то, что такая работа со списками корректна с точки зрения языка, она очень опасна с точки зрения работы реальных систем. По своему смыслу список — это коллекция объектов, к которым можно применить одни и те же операции. И если мы попробуем возвести в степень 2 каждый элемент списка, то получим ошибку:

example-26.3.5.py

код
```
lst = [2, 6.5, "банан", True, 11]
for i in lst:
    print(i ** 2)
```

вывод
```
4
42.25
Traceback (most recent call last):
  File "example-26.3.2.py", line 3, in <module>
    print(i ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and
'int'
```

Первые два элемента списка, являющиеся числами, были корректно возведены в квадрат, но затем интерпретатор Python сообщает нам, что операция возведения в степень неприменима к строкам, и выдает ошибку. Но даже если мы найдем операцию, применимую ко всем типам данных элементов, хранящихся в списке, результат может получиться мало предсказуемым.

Примечание

Хранение в одном списке элементов разных типов данных считается плохой практикой.

А теперь попробуем собрать в одном списке что-нибудь близкое по смыслу, например создадим список булевых значений из явных и неявных истинных значений:

example-26.3.6.py

код
```
lst = [True, "True", 1, 1.0]
print(lst)
```

вывод
```
[True, 'True', 1, 1.0]
```

И то же самое мы можем проделать со списком ложных значений, в любом порядке:

example-26.3.7.py	
код	`lst = ["", 0.0, False, 0]` `print(lst)`
вывод	`['', 0.0, False, 0]`

4. Преобразование строки к списку и списка к строке

При работе с различными типами данных иногда возникает необходимость преобразовать их к списку, например, для удобства и упрощения решения какой-либо задачи. Скажем, мы получили список электронных адресов пользователей через запятую в виде строки, а нам нужно сделать рекламную рассылку по этим адресам. Получается, надо превратить строку в список для рассылки.

Теория

Одним из самых частых преобразований типов при работе с Python является преобразование строки в список. Например, пусть набор покупок записан в одну строку как последовательность слов, а нам нужно получить список покупок:

example-26.4.1.py	

код	`shopstring = "яблоки, банан, говядина, яйца"`
	`print(shopstring)`
	
	`shoplist = shopstring.split()`
	`print(shoplist)`

вывод	`яблоки, банан, говядина, яйца` `['яблоки,', 'банан,', 'говядина,', 'яйца']`

Для того чтобы превратить такую строку в список слов, можно использовать метод строки **.split()**, который принимает два необязательный параметра:

1) разделитель — символ или другая строка, по которой строка будет разбита на несколько, по умолчанию это пробел (" ");
2) количество разделений — сколько будет использовано разделителей для формирования списка, по умолчанию это -1, то есть все найденные разделители.

По умолчанию разделитель равен символу пробела " ", наиболее часто встречающемуся разделителю. Синтаксис будет таким:

<string>**.split(**<separator>, <maxsplit>**)**

Видим, что в предыдущем примере строка была преобразована в список слов по разделяющему пробелу. Но запятые также являются разделяющими символами, и они остались «прилеплены» к словам. Используем другой разделитель, а именно **сочетание запятой и пробела**:

example-26.4.2.py	

код	`shopstring = "яблоки, банан, говядина, яйца"`
	`shoplist = shopstring.split(", ")`
	`print(shoplist)`

вывод	`['яблоки', 'банан', 'говядина', 'яйца']`

Теперь строка преобразована в список правильным образом.

Примечание
В качестве разделителя для метода **.split()** может быть использован любой символ или строка.

А теперь поступим наоборот: склеим список в строку, и поможет нам в этом метод строки **.join()**, который принимает один обязательный параметр:

- список для склеивания, а если быть точнее, то итерируемый объект.

Разделителем является строка, для которой вызван метод **.join()**, то есть синтаксис такой:

<string>**.join(**<list>**)**

Сделаем из списка продуктов строчку продуктов:

example-26.4.3.py	
код	`shoplist = ["яблоки", "банан", "говядина", "яйца"]` `print("lst:", shoplist)`

	shopstring = ", ".join(shoplist) print("str:", shopstring)
вывод	lst: ['яблоки', 'банан', 'говядина', 'яйца'] str: яблоки, банан, говядина, яйца