10. Работа со списками

- Списки

- Методы списков

- Генераторы и итераторы

В Python существует четыре типа данных для хранения последовательностей:

- **list** (список) — упорядоченная последовательность, которую можно изменять. Допускаются одинаковые элементы.
- **tuple** (кортеж) — последовательность, которая упорядочена, но не изменяемая. Допускаются одинаковые элементы.
- **dict** (словарь) — неупорядоченная изменяемая последовательность, состоящая из пар ключ, значение. Ключи не дублируются.
- **set** (множество) — неупорядоченная изменяемая последовательность. Одинаковые элементы удаляются.

При выборе типа последовательности полезно знать и понимать свойства каждого из типов. Выбор правильного типа для определенного набора данных помогает сохранить смысл, и это дает повышение эффективности или безопасности.

Списки

Списки – это **упорядоченная** и **изменяемая последовательность**.

Список используется для хранения последовательности различных типов данных. Списки в Python являются мутабельными типами, то есть мы можем изменять их элементы после их создания.

Список можно определить как коллекцию значений или элементов различных типов. Элементы в списке разделяются запятой , и заключаются в квадратные скобки [ ]

```
fruits = ["яблоко", "банан", "вишня"]
print(fruits)
```

Рисунок.1 – Создание списка

```
['яблоко', 'банан', 'вишня']

Process finished with exit code 0
```

Рисунок.1.1 – Результат Рис.1

```
data_list = [49, True, 'Вася', 3.48]
print(data_list)
```

Рисунок.2 – Создание списка с разными типам данных

```
[49, True, 'Вася', 3.48]

Process finished with exit code 0
```

Рисунок.2.1 – Результат Рис.2

Особенности списков (list)

Список имеет следующие характеристики:

- Списки упорядочены.
- К элементу списка можно получить доступ по индексу.
- Списки являются изменяемыми типами.
- Список может хранить количество различных элементов.

Проверим первое утверждение о том, что списки являются упорядоченными.

```
>>> list_a = [41, 0, "Иван", 5.79, "Петр", 25, 197]
>>> list_b = [41, 0, 25, "Иван", 5.79, "Петр", 197]
>>> list_a == list_b
False
```

Рисунок.3 – Сравнение списков

Оба списка состоят из одинаковых элементов, но во втором списке изменена позиция индекса 5-го элемента, что нарушает порядок списков. При сравнении обоих списков возвращается **False**

Списки сохраняют порядок элементов на протяжении всего жизненного цикла. Именно поэтому они являются упорядоченной коллекцией объектов

```
>>> list_a = [41,0, "Иван", 5.79, "Петр", 25, 197]
>>> list_b = [41,0, "Иван", 5.79, "Петр", 25, 197]
>>> list_a == list_b
True
```

Рисунок.3.1 – Сравнение списков

Индексирование и разбивка списков

Индексация обрабатывается так же, как и в случае со строками. Доступ к элементам списка осуществляется с помощью оператора *slice(среза)* [ ]

Индекс начинается с 0 и идет до длины -1. Первый элемент списка хранится по 0-му индексу, второй элемент списка хранится по 1-му индексу и так далее.

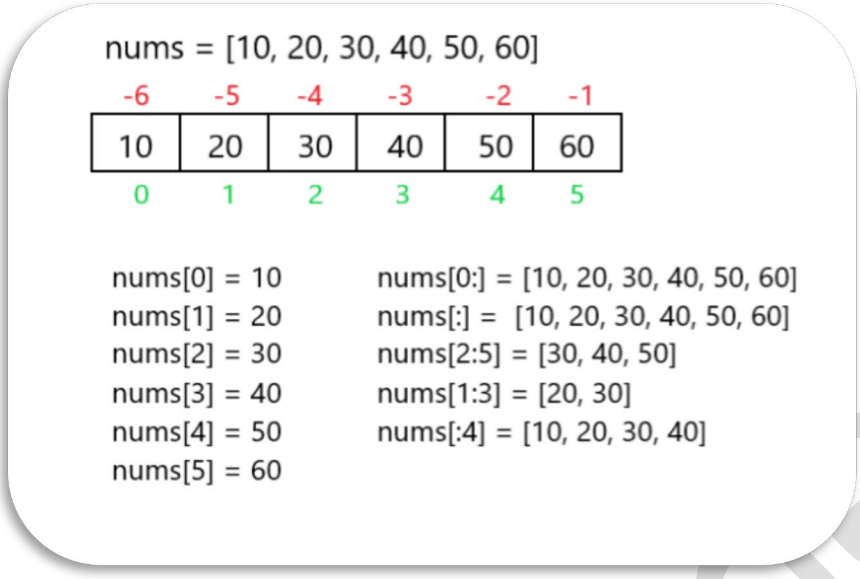

Рисунок.4 – Индексирование и разбивка списков

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> print(fruits[1])
банан
```

Рисунок.4.1 – Доступ к элементам списка

Также поддерживается отрицательная индексация. Отрицательная индексация **начинается с конца**. Иногда её удобнее использовать для получения последнего элемента в списке, потому что не нужно знать длину списка, чтобы получить доступ к последнему элементу

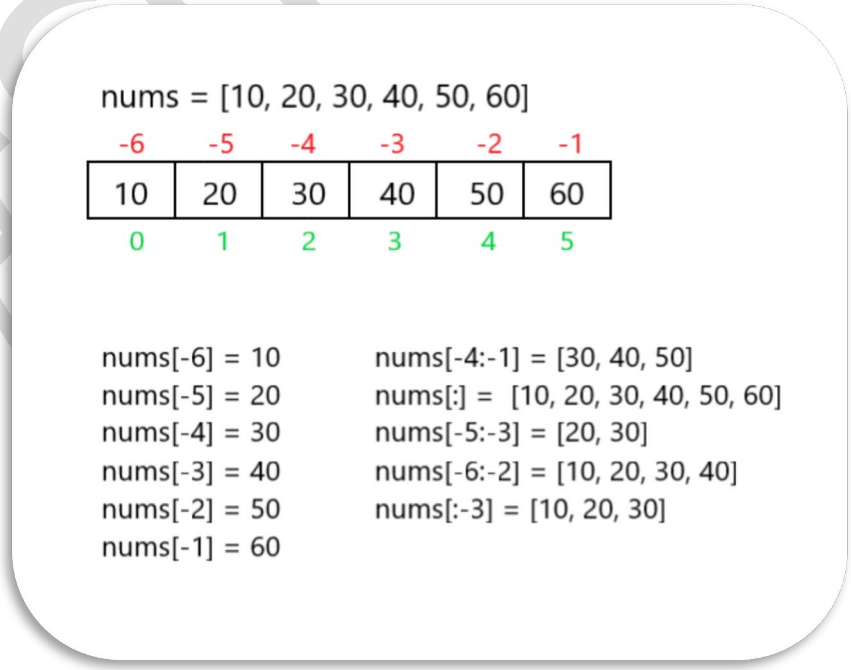

Рисунок.4.2 – Индексирование и разбивка списков

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> print(fruits[-1])
вишня
```

Рисунок.4.3 – Доступ к последнему элементу списка

Мы можем получить подсписок списка, используя следующий синтаксис.

var_list[start:stop:step]

- start обозначает начальную индексную позицию списка.
- stop обозначает последнюю индексную позицию списка.
- step используется для пропуска **n-го** элемента в пределах start:stop.

```
>>> data = ['One', 'Two', 3, 'Four', '5', 'Six', 'Seven']
>>> print(data[0:6])
['One', 'Two', 3, 'Four', '5', 'Six']
>>> print(data[1:6:2])
['Two', 'Four', 'Six']
>>> print(data[-5:-2:2])
[3, '5']
>>> print(data[-6:-3])
['Two', 3, 'Four']
```

Рисунок.5 – Получить подсписок списка

Обновление значений списка

Списки являются наиболее универсальными структурами данных, поскольку они *являются изменяемыми*, и их значения можно *обновлять.*

Python также предоставляет методы list.append() и list.insert(), которые можно использовать для добавления значений в список.

Для того, чтобы изменить значение определенного элемента, ссылайтесь на номер индекса.

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> fruits[1] = "смородина"
>>> print(fruits)
['яблоко', 'смородина', 'вишня']
```

Рисунок.6 – Изменить значение по индексу

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> fruits[-3] = "груша"
>>> print(fruits)
['груша', 'банан', 'вишня']
```

Рисунок.6.1 – Изменить значение по отрицательному индексу

```
>>> nums = [1, 2, 3, 4, 5, 6]
>>> # Добавление нескольких элементов
>>> nums[2:5] = [30, 60, 90]
>>> print(nums)
[1, 2, 30, 60, 90, 6]
```

Рисунок.6.2 – Изменить несколько значений с помощью оператора среза

Встроенные функции списков

Python предоставляет следующие встроенные функции, которые можно использовать со списками.

«Встроенные функции списков» Таблица.1

Функция	Описание
len(list)	Используется для вычисления длины списка.
max(list)	Возвращает максимальный элемент списка.
min(list)	Возвращает минимальный элемент списка.
list(seq)	Он преобразует любую последовательность в список

Методы списков

У списков есть разные методы, которые помогают в программировании

list.index() возвращает индекс первого элемента с определенным значением

l.index() возвращает положение первого индекса, со значением x. В указанном ниже коде, *Рис.7* он возвращает назад 4.

```
>>> fruits = ["яблоко", "виноград", "банан", "вишня", "персик", "абрикос"]
>>> fruits.index("персик")
4
```

*Рисунок.7 – Метод **l.index()***

Вы также можете указать, откуда начинаете поиск *Рис.7.1*

```
>>> fruits = ["яблоко", "виноград", "банан", "вишня", "персик", "абрикос"]
>>> print(fruits.index("персик", 3))
4
>>> print(fruits.index("яблоко", 3))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: 'яблоко' is not in list
```

*Рисунок.7.1 – Метод **l.index()***

list.count() возвращает число элементов с определенным значением

l.count() работает так, как звучит. Он считает количество раз, когда значение появляется в списке.

```
>>> nums = [10, 90, 40, 20, 10, 60, 70, 10, 30, 50, 10]
>>> nums.count(10)
4
```

*Рисунок.8 – Метод **l.count()***

list.reverse() разворачивает список

l.reverse() *"переворачивает"* список, переставляет элементы в обратном порядке, сначала становиться последний элемент, а в конец первый элемент.

```
>>> letters = ["A", "B", "C", "D", "E", "F"]
>>> letters.reverse()
>>> print(letters)
['F', 'E', 'D', 'C', 'B', 'A']
```

*Рисунок.9 – Метод **l.reverse()***

list.sort() сортирует список

l.sort() сортирует и меняет исходный список.

```
>>> nums = [10, 90, 40, 20, 10, 60, 70, 10, 30, 50, 10]
>>> nums.sort()
>>> print(nums)
[10, 10, 10, 10, 20, 30, 40, 50, 60, 70, 90]
```

*Рисунок.10 – Метод **l.sort()**, сортировка списка с наименьшего значения к наибольшему*

Вышеуказанный код *Рис.10* сортирует список чисел от наименьшего к наибольшему. Код, указанный ниже *Рис.10.1*, показывает, как вы можете сортировать список от наибольшего к наименьшему.

```
>>> nums = [10, 90, 40, 20, 10, 60, 70, 10, 30, 50, 10]
>>> nums.sort(reverse=True)
>>> print(nums)
[90, 70, 60, 50, 40, 30, 20, 10, 10, 10, 10]
```

*Рисунок.10.1 – Метод **l.sort()**, сортировка списка с наибольшего значения к наименьшему*

Следует отметить, что вы также можете отсортировать список строк *от А до Я (или A-Z)* и наоборот.

```
>>> fruits = ["яблоко", "виноград", "банан", "вишня", "персик", "абрикос"]
>>> fruits.sort()
>>> print(fruits)
['абрикос', 'банан', 'виноград', 'вишня', 'персик', 'яблоко']
```

Рисунок.10.2 – Сортировка списка строк

list.copy() возвращает копию списка

l.copy() создает мелкую копию списка

```
>>> new_nums = nums.copy()
>>> new_nums[3] = 5000
>>> print(nums)
[3, 6, 9, 12, 15, 18]
>>> print(new_nums)
[3, 6, 9, 5000, 15, 18]
```

*Рисунок.11 – Мелкая копия списка **l.copy()***

Добавление элементов

list.append() добавляет элемент(ы) в конец списка

l.append() добавляет элемент в конец списка. Это происходит на месте.

```
>>> nums = [10, 90, 40, 20, 10, 60, 70, 10, 30, 50, 10]
>>> nums.append(20)
>>> print(nums)
[10, 90, 40, 20, 10, 60, 70, 10, 30, 50, 10, 20]
```

*Рисунок.12 – Метод **l.append()** добавить значение 20 в конец списка*

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> fruits.append("клубника")
>>> print(fruits)
['яблоко', 'банан', 'вишня', 'клубника']
```

*Рисунок.12.1 – Метод **l.append()** добавить строку "клубника" в конец списка*

list.insert() добавляет элемент по индексу

l.insert() вставляет элемент перед указанным индексом

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> fruits.insert(2, "авакадо")
>>> print(fruits)
['яблоко', 'банан', 'авакадо', 'вишня']
```

*Рисунок.13 – Метод **l.insert()** добавление элемента по индексу*

list.extend() добавляет элементы в конец текущего списка

l.extend() расширяет список, добавляя элементы. Преимущество над методом l.append() в том, что вы можете добавлять списки

```
>>> fruits = ["вишня", "персик", "абрикос"]
>>> tropical = ["киви", "апельсин", "кокос"]
>>> fruits.extend(tropical)
>>> print(fruits)
['вишня', 'персик', 'абрикос', 'киви', 'апельсин', 'кокос']
```

*Рисунок.14 – Метод **l.extend()** добавление элементов в конец списка*

```
>>> fruits = ["вишня", "персик", "абрикос"]
>>> tropical = ["киви", "апельсин", "кокос"]
>>> fruits.append(tropical)
>>> print(fruits)
['вишня', 'персик', 'абрикос', ['киви', 'апельсин', 'кокос']]
```

*Рисунок.14.1 – Метод **l.append()** добавление элемента в конец списка*

То же самое, что делает метод l.extend() можно было бы сделать, используя оператор + *Рис.14.2*

```
>>> fruits = ["вишня", "персик", "абрикос"]
>>> tropical = ["киви", "апельсин", "кокос"]
>>> print(fruits + tropical)
['вишня', 'персик', 'абрикос', 'киви', 'апельсин', 'кокос']
```

Рисунок.14.2 – Объединение двух списков

Удаление элементов

Существует несколько методов удаления элементов списка

list.remove() удаляет элементы по значению

l.remove() удаляет определенные элементы

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> fruits.remove("вишня")
>>> print(fruits)
['яблоко', 'банан']
```

*Риснок.15 – Метод **l.remove()** удаление элемента по значению*

list.pop() удаляет элемент по индексу или последний

l.pop() **удаляет элемент** по индексу (или последний элемент, если индекс не указан) **и возвращает его**

```
>>> fruits = ["яблоко", "виноград", "банан", "вишня", "персик", "абрикос"]
>>> last_element = fruits.pop()
>>> print(fruits)
['яблоко', 'виноград', 'банан', 'вишня', 'персик']
>>> print(last_element)
абрикос
```

*Риснок.16 – Метод **l.pop()** удаление элемента*

```
>>> fruits = ["яблоко", "виноград", "банан", "вишня", "персик", "абрикос"]
>>> element_2 = fruits.pop(2)
>>> print(fruits)
['яблоко', 'виноград', 'вишня', 'персик', 'абрикос']
>>> print(element_2)
банан
```

*Риснок.16.1 – Метод **l.pop()** удаление элемента*

Ключевое слово del удаляет определенный индекс

```
>>> fruits = ["яблоко", "банан", "вишня"]
>>> del fruits[0]
>>> print(fruits)
['банан', 'вишня']
```

*Риснок.16.2 – Ключевое слово **del** удаление элемента*

Ключевое слово del может полностью удалить список

```
>>> fruits = ["банан", "вишня", "персик", "абрикос"]
>>> del fruits
>>> print(fruits)  # это вызывает ошибку так, как "fruits" больше не существует.
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'fruits' is not defined
```

*Риснок.16.3 – Ключевое слово **del** удаление списка*

list.clear() удаляет все элементы в списке

l.clear() очищает список

```
>>> fruits = ["виноград", "клубника", "груша"]
>>> fruits.clear()
>>> print(fruits)
[]
```

*Риснок.17 – Метод **l.clear()** удаление всех элементов*

Конструктор list()

Вы так же можете использовать конструктор list() для создания списка

```
>>> fruits = list(("яблоко", "банан", "вишня"))  # обратите внимание на двойные круглые скобки
>>> print(fruits)
['яблоко', 'банан', 'вишня']
>>> apple = list("яблоко")
>>> print(apple)
['я', 'б', 'л', 'о', 'к', 'о']
```

*Рисунок.18 – Конструктор **list()** преобразует последовательность в список*

Простые операции над списками

Таблица.2

Метод	Описание
x in s	**True** если элемент **x** находится в списке **s**
x not in s	**True** если элемент **x** не находится в списке **s**
s1 + s2	Объединение списков **s1** и **s2**
s * n или n * s	Копирует список **s n** раз
len(s)	Длина списка **s**, т.е. количество элементов в **s**
min(s)	Наименьший элемент списка **s**
max(s)	Наибольший элемент списка **s**
sum(s)	Сумма чисел списка **s**
list("Python")	Преобразует последовательность в список

```
nums = [14, -5, 65, 0, 16, -88, 345, 2]
print("Список nums =", nums)
print("16 есть в списке nums - это", 16 in nums)
print("16 нет в списке nums - это", 16 not in nums)

# + объединяет два списка
list_1 = [45, 97]
list_2 = [101, 20]
list_3 = list_1 + list_2
print("list_3 =", list_3)

# * копирует элементы в списке.
list_4 = [10, 20, 30]
list_5 = list_4 * 3
print("list_5 =", list_5)

print("Количество элементов списка:", len(nums))
print("Самый большой элемент списка:", max(nums))
print("Наименьший элемент списка:", min(nums))
print("Сумма чисел в списке:", sum(nums))

text_list = list("Python")
print("text_list =", text_list)
```

Рисунок.19 – Простые операции над списками

```
Список nums = [14, -5, 65, 0, 16, -88, 345, 2]
16 есть в списке nums - это True
16 нет в списке nums - это False
list_3 = [45, 97, 101, 20]
list_5 = [10, 20, 30, 10, 20, 30, 10, 20, 30]
Количество элементов списка: 8
Самый большой элемент списка: 345
Наименьший элемент списка: -88
Сумма чисел в списке: 349
text_list = ['P', 'y', 't', 'h', 'o', 'n']

Process finished with exit code 0
```

Рисунок.19.1 – Результат Рис.19

Генераторы и итераторы

Генераторы списков (List comprehension)

Генераторы списков служат для создания новых списков на основе существующих. Представьте, что имеется список чисел, на основе которого требуется получить новый список, состоящий из всех чисел, *умноженных на 2*, но только при условии, что само *число больше 10*. Генераторы списков подходят для таких задач как нельзя лучше.

```
list_one = [15, 68, 35, 4, 7, 29, 10,]
list_two = [2*i for i in list_one if i > 10]

print("List_one =", list_one)
print("List_two =", list_two)
```

Рисунок.20 – Генератор списка

В этом примере мы создаём новый список, указав операцию, которую необходимо произвести 2 * i, когда выполняется некоторое условие if i > 10 Обратите внимание, что исходный список при этом не изменяется.

Преимущество использования генераторов списков состоит в том, что это заметно сокращает объёмы стандартного кода, необходимого для циклической обработки каждого элемента списка и сохранения его в новом списке.

```
List_one = [15, 68, 35, 4, 7, 29, 10]
List_two = [30, 136, 70, 58]

Process finished with exit code 0
```

Рисунок.20.1 – Результат Рис.20

Итераторы

Когда мы создаём список, то можем считывать его элементы один за другим – это называется итерацией.

```
my_list = [10, 20, 30]
for i in my_list:
    print(i)
```

Рисунок.21 – Итераторы

```
10
20
30

Process finished with exit code 0
```

Рисунок.21.1 – Результат Рис.21

my_list является итерируемым объектом. Когда мы создаём список, используя *генераторное выражение*, мы также создаём итератор

```
# генератор списка
new_my_list = [x * x for x in range(1, 10, 2)]

# итератор
for i in new_my_list:
    print(i)
```

Рисунок.21.2 – Итераторы

```
1
9
25
49
81

Process finished with exit code 0
```

Рисунок.21.3 – Результат Рис.21.2

Всё, к чему можно применить конструкцию for in является итерируемым объектом: *списки, строки, файлы и т.д.*

Это удобно, потому что можно считывать из них значения сколько потребуется – однако все значения хранятся в памяти, а это не всегда желательно, если у нас много значений.

Генераторы

Генераторы – это тоже итерируемые объекты, но прочитать их можно лишь один раз. Это связано с тем, что они не хранят значения в памяти, а генерируют их на лету

```python
# генератор
my_generator = (num * num for num in range(1, 6))

# итератор
for n in my_generator:
    print(n)
```

Рисунок.22 – Генераторы

```
1
4
9
16
25

Process finished with exit code 0
```

Рисунок.22.1 – Результат Рис.22

Всё то же самое, разве что используются круглые скобки вместо квадратных. Но нельзя применить конструкцию for in второй раз, так как генератор может быть использован только единожды: он вычисляет 1, потом забывает про него и вычисляет 4, завершая вычислением 25 – одно за другим.