



Урок 2

Встроенные типы и операции с ними

Последовательности (итераторы). Строки, списки, кортежи, словари, множества. Обход последовательностей в цикле.

[Строки](#)

[Базовые операции со строками](#)

[Функция len](#)

[Методы строк](#)

[Форматирование строк](#)

[Списки](#)

[Методы списков](#)

[Кортежи](#)

[Последовательности \(Обобщение\)](#)

[Обход последовательностей в цикле \(for in\)](#)

[Словари](#)

[Методы словарей](#)

[Множества](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Строки

В предыдущем уроке мы уже пользовались строками, пришло время рассмотреть их подробнее.

Строки в Python - упорядоченные неизменяемые последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме.

К любому символу последовательности можно обратиться по номеру его индекса. Индекс последовательности – порядковый номер элемента последовательности.

Последовательность: В А С Я Номера индексов: [0] [1] [2] [3]

Обратите внимание! Элементов (букв) последовательности 4, а последний индекс равен 3.

Базовые операции со строками

```
print("*****Операции со строками*****")
# 1. Строки можно складывать:
print('Hello' + ' ' + 'world')
# 2. Если строки идут друг за другом, + можно опустить
#(конкатенация строк произойдет автоматически):
print('Hello' ' ' 'world')
# 3. Строки повторять операцией *:
print('Hey!' * 3)
string = 'произвольная строка'
print('string = ', string)
# 4. Получение символа строки по индексу:
# Все элементы строки нумеруются порядковыми индексами (первый индекс НОЛЬ):
print('string[1] --> ', string[1])
# 5. Срезы
# Подстроку можно получить при помощи срезов:
print('string[6:11] --> ', string[6:11])
# Значения по умолчанию: опущенный первый индекс заменяется нулём,
# опущенный второй индекс подменяется размером срезаемой строки.
print('string[6:] --> ', string[6:])
print('string[:11] --> ', string[:11])
# s[3] = 'g' # такая конструкция вызовет ошибку, т.к. строки — неизменяемый объект
# Чересчур большой индекс заменяется на размер строки:
print('string[6:100]')
# Верхняя граница меньше нижней возвращает пустую строку:
print('string[50:] --> ', string[50:])
print('string[6:1] --> ', string[6:1])
# Индексы могут быть отрицательными числами,
# обозначая при этом отсчёт справа налево:
print('string[-1] --> ', string[-1]) # Последний символ
print('string[-2] --> ', string[-2]) # Предпоследний символ
print('string[-2:] --> ', string[-2:]) # Последние два символа
print('string[:-2] --> ', string[:-2]) # Всё, кроме последних двух символов
# Хороший способ понять, как работают срезы –
# думать о них, как об указателях на места между символами:
# +---+---+---+---+---+
# | L | o | r | e | m |
# +---+---+---+---+---+
# 0  1  2  3  4  5
# -5 -4 -3 -2 -1
# 6. Срезы с шагом
# Получаем каждый второй символ для указанного среза
print('string[12:2] --> ', string[12:2])
# Переворачиваем строку задом наперед
print('string[::-1] --> ', string[::-1])
# 7. Длина строки:
print(len(s))
```

Функция len

Работа функции len() очень проста - она принимает любую последовательность в качестве аргумента и возвращает её длину (количество элементов).

Методы строк

Подробно с методами мы познакомимся на 6 уроке. А пока воспринимайте методы, как действия, которые можно применять к различным данным. Метод - это функция, которая вызывается через точку после объекта, над которым нужно произвести некоторое действие.

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
>>> 'иван'.title()
: Иван
```

Метод `.title()` - первую букву переводит в верхний регистр, а все остальные в нижний.

```
>>> name = 'Вася'
>>> name.upper()
: ВАСЯ
```

Метод `.upper()` - переводит все буквы в верхний регистр/

Обратите внимание! Нет разницы, применять метод к значению или к переменной, содержащей это значение. Метод в любом случае применяется к самой строке.

Существуют методы, которые принимают дополнительные аргументы. Например:

```
>>> 'трололошка'.find('ло')
: 3
>>> 'трололошка'.find('ло', 4)
: 5
```

`.find()` - Поиск подстроки в строке. Возвращает номер последнего вхождения или -1. Вторым аргументом принимает индекс начала поиска (по умолчанию поиск производится с начала строки).

Полный список всех методов строк [здесь](#).

Форматирование строк

Довольно часто возникают ситуации, когда нужно сделать строку, подставив в неё некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, данные из файлов и т. д.).

Делать следующим образом - плохой способ, т.к. ухудшает читаемость и запись сложно редактировать.

```
name = 'Иван'
surname = 'Иванов'
print("Welcome, ' + surname + ' ' + name + ', to our conference')
```

Подстановку данных можно сделать с помощью форматирования строк. Форматирование можно сделать с помощью оператора `%`, либо с помощью метода `format`.

```
# Старый способ форматирования
print("Welcome, %s %s, to our conference" % (name, surname))
# Более новый и гибкий метод
print("Welcome, {} {}, to our conference".format(name, surname))
print("Welcome, {1} {0}, to our conference".format(name, surname))
```

Метод `.format()` наиболее гибкий и имеет много возможностей для форматирования.

Фигурными скобками `{ }` указываем места в строке-шаблоне, куда будет выполняться подстановка данных. Цифрами `{0}` `{1}` можем изменить порядок подставляемых данных.

Познакомиться со всеми инструментами форматирования можно [здесь](#). Крайне рекомендую прочитать статью по ссылке, там все довольно просто и доступно.

Списки

Списки – это упорядоченные, неограниченные по размеру коллекции объектов произвольных типов. В отличие от строк, списки являются изменяемыми: они могут модифицироваться как с помощью операций присваивания по индексам, так и с помощью разнообразных методов работы со списками. Поскольку списки являются последовательностями, они поддерживают все операции над последовательностями, которые обсуждались в разделе, посвященном строкам. Единственное отличие состоит в том, что результатом таких операций являются списки, а не строки. Если строки создаются с помощью литералов кавычек `""` / `''`. То списки создаются литералами квадратных скобок `[]`.

list.py Пример, операции со списками/

```
# Список - изменяемая последовательность, элементы которой - любые типы данных
empty_list = [] # пустой список
my_list = [1, 3, 5, 3.45, 'ddd', 's', 333]
print('my_list = ', my_list)
# Т.к. список – это последовательность, к нему применимы те же операции, как к строке.
# Получение элемента по индексу
print(my_list[0]) # получим первый элемент списка
print(my_list[-1]) # последний элемент списка
# Срезы
print(my_list[0:-3]) # 3 последних элемента списка
# Конкатенация
print(my_list[0:3] + [7, 8, 9]) # получим новый список из 6 элементов
# Мультипликация
print([3, '4'] * 3) # размножим список
# В отличие от строк, элементы списка можно изменять:
my_list[2] = 'New'
print('my_list after change =', my_list)
# А также заменять часть элементов с помощью срезов. Заменяем первые 3:
my_list[0:3] = [2, 4, 6]
print(my_list)
# Удалим последние 2
my_list[-2:] = []
print(my_list)
# Вставим несколько элементов внутрь
my_list[3:3] = ['this', 'is', 'some', 'elements']
print(my_list)
# Вставим элемент в начало списка
my_list[:0] = ['first']
print(my_list)
# Как и для строк, встроенная функция len() вернет длину списка:
print(len(my_list))
# Добавить что-то в конец списка можно так:
my_list[len(my_list):] = [100]
print(my_list)
# Но чаще используется более простая конструкция (простое лучше сложного, правда?):
my_list.append(200)
print(my_list)
# Можно создавать списки, содержащие другие списки:
b = [1, 2, 3, [11, 22, 33], 5, 6]
print('b = ', b)
print('b[3][2] =', b[3][2])
# Оператор вхождения in
print('2 in b -->', 2 in b)
print("'2' in b -->', '2' in b)
```

Методы списков

Список является изменяемым объектом, может модифицироваться многими своими методами.

```
>>> lst = [1, -2]
>>> lst.append(4) # добавит элемент в конец списка
>>> lst.pop()     # удалит последний элемент списка и вернет его
>>> lst.pop(1)    # удалит элемент списка с индексом 1
```

Кортежи

Кортеж – неизменяемый список. *Зачем нужны кортежи*, если есть списки?

1. “Защита от дурака”. То есть кортеж защищен от изменений, как намеренных (что плохо), так и случайных (что хорошо).
2. Меньший размер, по сравнению со списками, при одинаковом количестве элементов.

Пустой кортеж можно создать с помощью литерала круглые скобки ().

```
>>> t = ()
```

Но чтобы создать кортеж из одного элемента, необходимо поставить запятую

```
>>> t = (2) # так не годится, получим int
>>> print(t)
: 2
>>> t = (2, ) # а так получим именно кортеж
>>> print(t)
: (2, )
>>> t = 2, # так тоже получим кортеж
>>> print(t)
: (2, )
```

Т.е., по сути, кортеж создают не круглые скобки, а наличие запятой.

К кортежам можно применять те же операции и методы, что и к спискам, за исключением тех, что меняют сам кортеж.

Последовательности (Обобщение)

Мы рассмотрели три типа данных, являющихся последовательностями:

1. Строки.
2. Списки.
3. Кортежи.

Ко всем последовательностям применимы одни и те же операции:

- Обращение по индексу;
- Получение среза;
- Конкатенация;
- Мультипликация.

И многие другие, которые будем рассматривать далее.

Здесь важно понять следующее: для всех последовательностей(если говорить правильно итераторов) есть набор операций, которые работают одинаково в не зависимости от того строка это или список. Тут мы сталкиваемся со следующей важной философией python: “Если что-то летает как утка и крикает как утка - то это утка”. Т.е. не столь важно, каким объектом являются данные, важно,

какие операции к ним можно применить. Подробнее эту особенность мы разберем на уроке “Интерфейсы”.

Обход последовательностей в цикле (for in)

Очень часто возникает ситуация, когда нужно перебирать элементы последовательности. Конечно можно это делать с помощью известного цикла while.

```
# Перебор элементов по индексам
fruits = ['apple', 'banana', 'mango']
i = 0
while len(fruits) > i:
    print('fruit = ', fruits[i])
    i += 1
```

Это плохой и громоздкий способ. Это не python-way (way-путь).

Сравните предыдущий пример с наиболее элегантным циклом for in, созданным для работы специально с последовательностями:

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print('fruit = ', fruit)
```

1. В два раза меньше строк кода.
2. Не нужна переменная счетчик i.
3. Гораздо быстрее работает.

Если вы пришли в python из другого языка программирования, постарайтесь как можно быстрее перебороть желание работать с последовательностями через индексы.

Как работает for in? Очень просто! Переменной fruit при каждой итерации (полном обороте) цикла присваиваются элементы по очереди, когда элементы заканчиваются - цикл завершается.

```
for el in 'Hello':           # со строкой тоже работает
    print('el = ', el)
print()
for t_el in 1, 2, 3, 4, 5, 10: # и с кортежем
    print("-----")
    print('t_el = ', t_el)
print()
```

Словари

Словари в языке Python - это нечто совсем иное, они вообще не являются последовательностями, это то, что известно как отображения.

Отображения - это коллекции объектов, но доступ к ним осуществляется не по определенным смещениям от начала коллекции (индексам), а по ключам. В действительности отображения вообще не подразумевают какого-либо упорядочения элементов по их позиции, они просто отображают ключи на связанные с ними значения.

Словари – единственный тип отображения в наборе базовых объектов Python - также относятся к классу изменяемых объектов: они могут изменяться непосредственно и, в случае необходимости, могут увеличиваться и уменьшаться в размерах, подобно спискам.

Программный код определения словаря заключается в фигурные скобки и состоит из последовательности пар «ключ: значение».

Словари удобно использовать всегда, когда возникает необходимость связать значения с ключами, например, чтобы описать свойства чего-либо:

```
>>> fruit = {"name": "Carrot", "color": "orange", "quantity": 12}
```

Мы можем обращаться к элементам этого словаря по ключам и изменять значения, связанные с ключами. Для доступа к элементам словаря используется тот же синтаксис, который используется для обращения к элементам последовательностей, только в квадратных скобках указывается не смещение относительно начала последовательности, а ключ.

```
>>> fruit["name"]
: Carrot
```

Обращение к несуществующему ключу вызовет ошибку (исключение).

```
>>> fruit["from"]
: ... KeyError: 'from'
```

Добавление значения в словарь происходит присваиванием значения несуществующему ключу.

```
>>> new_dict = {} # создаем пустой словарь
>>> new_dict["new"] = "value"
>>> print(new_dict)
: {"new": "Value"}
```

Присваивание нового значения по существующему ключу **заменяет значение на новое**.

Методы словарей

```
# цикл по словарю
for key, value in f.items():
    print(key, value)
for key in f.keys():
    print(key)
for value in f.values():
    print(value)
# удаляет элемент с и возвращает его значение
print(f.pop('c'))
print(f)
# удаляет и возвращает пару (ключ, значение)
print(f.popitem())
```

Метод `.items()` - возвращает пары (ключ, значение).

Метод `.keys()` - возвращает список ключей.

Множества

Множество в python - "контейнер", содержащий неповторяющиеся элементы в случайном порядке.

Пример, демонстрирующий создание множеств и операции над ними:


```

a = set()
print('a = ', a) # set()
b = set(['a', 'b', 'c', 'c', 'a', 'e'])
print('b = ', b)
c = set('hello')
print('c = ', c)
d = {'a', 'b', 'c', 'd'}
print('d = ', d)
f = {} # А так получится словарь
print('type({}) -->', type(f)) # <class 'dict'>
# Операции с множествами
print(len(e))
print("'b' in b -->", 'b' in b)
# s == t
c1 = {'e', 'l', 'o', 'h'}
print(c == c1)
# s.issubset(t) s <= t
print(c <= c1)
# s.issuperset(t) s >= t
print(c >= {'h'})
# s.union(t, ...) s | t
print(b | d)
# s.intersection(t, ...) s & t
print(b & d)
# s.difference(t, ...) s - t
print(d - b)
# s.symmetric_difference(t) s ^ t
print(d ^ b)

```

Как видно из примера, множества имеют тот же литерал, что и словарь, но пустое множество с помощью литерала создать нельзя.

Множества удобно использовать для удаления повторяющихся элементов:

```

>>> words = ['hello', 'daddy', 'hello', 'mum']
>>> set(words)
: {'hello', 'daddy', 'mum'}

```

Подробнее о множествах [здесь](#).

Домашнее задание

1. Смотреть здесь https://github.com/GeekBrainsTutorial/Python_lessons_basic/tree/master/lesson02

Большинство заданий делятся на три категории easy, normal и hard.

- easy - простенькие задачи, на понимание основ.
- normal - если вы делаете эти задачи, то вы хорошо усвоили урок.
- hard - наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи - это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard – не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

Дополнительные материалы

Всё то, о чем сказано здесь, но подробнее:

1. [Строки](#) в python.
2. [Полный список методов строк](#).
3. [Форматирование строк](#).
4. [Кортежи](#).
5. [Словари](#).
6. [Множества](#).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Учим python качественно\(habr\)](#).
2. [Самоучитель по python](#).
3. [Книга Лутц М. "Изучаем Python" \(4-е издание\)](#).
4. [younglinux](#).