

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 3

Строки, списки

Содержание

1. Строки.....	4
Кодировка ASCII, Unicode, UTF-8, Byte-code	4
Строка — неизменяемая последовательность символов.....	14
Методы строк	16
Особенности работы со строками	21
Срез строки	30
Экранированные последовательности.....	35
«Сырые» строки	39
Форматированный вывод.....	43
Модуль string	51
Регулярные выражения, модуль re	56
Списки	73
Понятие классического массива	73
Понятие коллекции объектов.....	74
Ссылочный тип данных list.....	76
Создание списков	77
Генераторы списков.....	80
Работа со списками	83
Методы списков	87
Оператор принадлежности in.....	93
Особенности списков, ссылки и клонирование	94
Поиск элемента	96
Матрицы	97

1. Строки

Кодировка ASCII, Unicode, UTF-8, Byte-code

Большое количество информации, обрабатываемой программами, находится в текстовом виде. Текст воспринимается человеком легко, но понятия строки и символа для него довольно абстрактны. Для программиста же важно конкретно и четко понимать, как такие данные хранятся и обрабатываются компьютером.

Процессор компьютера работает только с числами, причем представленными в двоичном формате (с битами). Вспомним, что бит — это минимальная единица информации в памяти компьютера, которая может принимать только одно из двух значений: ноль или единицу. Соответственно, все данные (числовые, текстовые, видео, аудио-информация, изображения), хранимые и обрабатываемые компьютером, это последовательность нулей и единиц.

Процессор работает с двоичным представлением данных, но может преобразовывать их в другие системы счисления, например, в шестнадцатеричную (которая легче и удобнее воспринимается человеком)

В шестнадцатеричной системе счисления каждый байт представляется двумя цифрами (например, число 0 — 0x00 в шестнадцатеричном виде, префикс «0x» — признак шестнадцатеричной формы).

Текстовые данные состоят из символов, которые уже являются элементарными объектами. Символов в языке

конечное множество (алфавит языка). Количество символов в алфавите языка — мощность алфавита.

Для того, чтобы узнать, сколько информации можно представить некоторым алфавитом, используется формула:

$$N = 2^b,$$

где

N — мощность алфавита;

b — количество бит для представления символа.

Если для одного символа выделяется один байт (8 бит), то максимальное количество символов алфавита — это $2^8 = 256$, которое является достаточным для представления всех необходимых символов языка.

Таким образом, каждый символ текста представляет из себя восьмиразрядный двоичный код.

Процесс кодирования заключается в том, что каждому символу алфавита ставится в соответствие уникальный код (от 0 до 255 в десятичной системе или его двоичное представление от 00000000 до 11111111). Компьютер различает (распознает) символы по их коду.

Для того, чтобы понимать, какому символу соответствует какой код используются так называемые таблицы кодировки, в которых каждому символу алфавита (алфавитно-цифровому символу) поставлен в соответствие набор числовых значений (код)

Первой таблицей кодировок стала ASCII (англ. *American Standard Code for Information Interchange* — Американский кодовый стандарт для обмена информацией). Первая версия ASCII использовала только семь бит для

представления символа, соответственно, мощность алфавита была 127 символов языка.

Первые тридцать два кода в таблице называются управляющими (используются для форматированного вывода текста и других служебных целей). Коды буквенно-цифровых символов находятся в диапазоне с 32 по 126 (код 127 не занят никаким символом).

Кроме символов латинского алфавита, таблица содержала цифры, арифметические знаки, знаки препинания, символы пробела, скобки и т.п.

Например, код латинского символа «А» — 65, а «В» — 66. Так как символ «В» идет в алфавите после символа «А», то его численный код больше.

Коды строчных символов отличаются от кодов заглавных, например, код «а» — это 97.

Если символ является цифрой, то это не значит, что его код совпадает с его значением. Например, код цифры «0» — это 48.

Однако, кроме английского языка в мире есть и другие, алфавиты которых также нужно представлять с помощью таблиц кодировок. Тогда 127 символов явно недостаточно, чтобы можно было работать и с другими языками. Для решения этой проблемы кодировка ASCII развивалась и параллельно появлялись другие стандарты. Каждый символ стал уже кодироваться восьмибитным кодом и, соответственно, мощность алфавита уже составляла 256 символов.

Первая половина таблицы кодировки ASCII оставалась неизменной (см. рис. 1).

Нижняя часть кодировочной ASCII – таблицы.							
0	16 ►	32 пробел	48 0	64 @	80 P	96 `	112 p
1 ☺	17 ◀	33 !	49 1	65 A	81 Q	97 a	113 q
2 ☹	18 ↑	34 «	50 2	66 B	82 R	98 b	114 r
3 ♥	19 !!	35 #	51 3	67 C	83 S	99 c	115 s
4 ♦	20 ¶	36 \$	52 4	68 D	84 T	100 d	116 t
5 ♣	21 §	37 %	53 5	69 E	85 U	101 e	117 u
6 ♠	22 —	38 &	54 6	70 F	86 V	102 f	118 v
7 •	23 ↓	39 '	55 7	71 G	87 W	103 g	119 w
8 ■	24 ↑	40 (56 8	72 H	88 X	104 h	120 x
9 ○	25 ↓	41)	57 9	73 I	89 Y	105 i	121 y
10 ■	26 →	42 *	58 :	74 J	90 Z	106 j	122 z
11 ♂	27 ←	43 +	59 ;	75 K	91 [107 k	123 {
12 ♀	28 ⊥	44 ,	60 <	76 L	92 \	108 l	124
13 ♪	29 ↔	45 -	61 =	77 M	93]	109 m	125 }
14 🎵	30 ▲	46 .	62 >	78 N	94 ^	110 n	126 ~
15 ☀	31 ▼	47 /	63 ?	79 O	95 _	111 o	127

Рисунок 1

Данная часть ASCII таблицы — это общепринятый мировой стандарт (текст, использующий только эти символы, будет правильно отображаться на любом компьютере, т.е. текст, состоящий только из латинских букв и цифр будет правильно прочитан).

Вторая (верхняя) часть таблицы (128 кодов, со 128 по 255) была задействована для символов других языков и псевдографики. Однако, языков много и для кодирования

всех символов всех языков 128 кодов не хватит. В связи с этим при использовании однобайтовой кодировки для каждого языка существует своя верхняя часть таблицы ASCII.

Проблемы были не в том, что для каждого языка приходилось разрабатывать свой стандарт, а в том, что этот стандарт оказывался несовместимый с другими (например, в случаях, когда в пределах одного документа используется несколько языков, то установка одной кодировки на весь документ может привести к неверной интерпретации одного или нескольких таких фрагментов) Также часто для одного языка могло быть разработано несколько разных стандартов. Например, для русского языка, были созданы стандарты CP866 (Code Page 866), KOI8-R, KOI8-U, ISO-8859-5. И в этом хаотическом наборе стандартов было достаточно трудно разбираться.

Таким образом, возникла необходимость перехода к стандартам, в которых для кодирования символа будет использоваться более одного байта. В 1991 г. был разработан стандарт Юникод (англ. Unicode), с применением которого можно было в одном документе использовать разные языки, знаки математических формул, т.д.

Каждый символ в стандарте Unicode кодируется с помощью 31 бита (4 байта за вычетом одного бита). Таким образом, получаем $2^{31} = 2\,147\,483\,684$ (т.е. более двух миллиардов), что вполне достаточно для создания уникального кода для каждого символа каждого языка.

Unicode является именно стандартом, а не кодировкой, т.е. это таблица символов, состоящая из 1114112 позиций, большинство из которых не заполнены. Это пространство разделено на 17 блоков, по 65 536 символов в каждом.

Каждый блок включает определенную группу символов, например, в нулевом (базовом) блоке содержатся наиболее употребляемые символы всех современных алфавитов. В Unicode первые 128 кодов совпадают с таблицей ASCII.

Записываются символы в шестнадцатеричном виде с приставкой «U+». Например, базовый блок включает в себя символы от U+0000 до U+FFFF (от 0 до 65535).

Отображение кодов выглядит так:

"Hello World"	
U+0048	: LATIN CAPITAL LETTER H
U+0065	: LATIN SMALL LETTER E
U+006C	: LATIN SMALL LETTER L
U+006C	: LATIN SMALL LETTER L
U+006F	: LATIN SMALL LETTER O
U+0020	: SPACE [SP]
U+0057	: LATIN CAPITAL LETTER W
U+006F	: LATIN SMALL LETTER O
U+0072	: LATIN SMALL LETTER R
U+006C	: LATIN SMALL LETTER L
U+0064	: LATIN SMALL LETTER D

Рисунок 2

U+ указывает на то, что это стандарт Unicode, а номер — число, в которое переведен двоичный код для данного символа. В Юникоде используется шестнадцатеричная система.

Одним из основных принципов в философии Unicode является чёткое разграничение между символами, их представлением в памяти компьютера и их внешним отображением на устройстве вывода.

Например, юникод-символ U+041F — это заглавная кириллическая буква П. Существует несколько возможностей представления данного символа в памяти компьютера (в зависимости от того, какое количество бит выделяется под его код), ровно как и огромное количество способов отображения его на экране монитора. Но при этом символ «П» понимается всегда только, как символ «П»

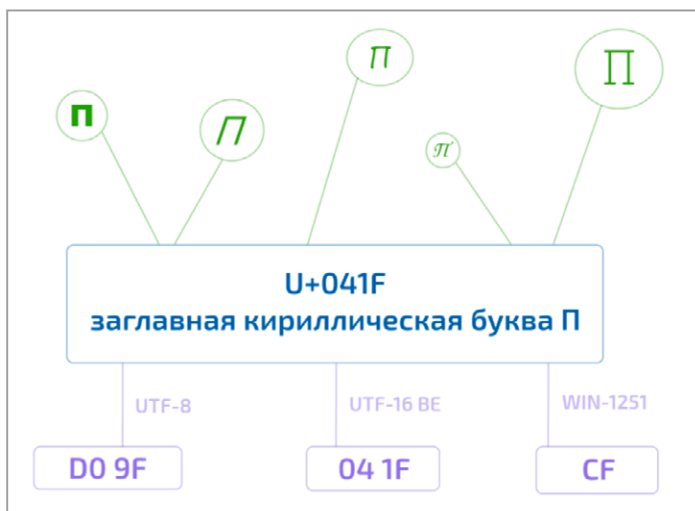


Рисунок 3

Если до Unicode каждому символу соответствовал код длиной одним байт, то в Unicode символ может быть закодирован разным количеством байтов. Для решения этих вопросов были созданы Unicode — кодировки (форматы преобразования Unicode), например, UTF-8.

UTF (*Unicode Transform Format*) — это способ представления Unicode. Кодировки UTF определены стандартом Unicode и позволяют закодировать любой нужный нам кодовый пункт Unicode.

Существует несколько разных видов UTF-стандартов, которые отличаются количеством байтов, используемых для кодирования одного кодового пункта. В UTF-8 используется один байт на кодовый пункт, в UTF-16 — 2 байта, в UTF-32 — 4 байта.

Как уже было отмечено, UTF-8 использует для кода символа 1 байт (как и ASCII), т.е. если программист при разработке программы использовал кодировку ASCII, а пользователи — UTF-8, то в отображении символов не будет никакой разницы.

Однако, есть символы Юникод, для кодирования которых требуется больше байт. Понятно, что это зависит от языка. Символ английского алфавита представляется 1 байтом, а вот иврит, арабские символы — уже 2 байтами. Для кодирования символов многих языков (например, китайского или японского) необходимо 3 байта.

Таким образом, особенностью UTF-8 является то, что она позволяет кодировать символы таким способом, что самые распространённые символы занимают 1-2 байта, а для кодирования редко встречающихся символов 3-4 байта.

Для того, чтобы указать, что символ занимает более одного байта, используется специальная битовая комбинация (знак продолжения), указывающая что код данного символа продолжается в следующих байтах.

В Python строка — это последовательность символов, т.е. последовательность кодов, например, с использованием юникод-символов.

Как было уже рассмотрено выше, любая последовательность символов сохраняется и обрабатывается

компьютером как последовательность байт (байт-строка, Byte-code).

Эта последовательность образуется в результате применения кодирования, способ которого зависит от кодировки.

Для преобразования байт-строки в обычную (воспринимаемую человеком) строку необходимо выполнять декодирования с помощью специальных методов (`.decode()`). Но чтобы процесс декодирования был выполнен правильно нам необходимо знать кодировку (которая была использована для получения этой последовательности байтов), т.к. поскольку другая кодировка может отображать одни и те же байты в виде других символов строки.

Кодировку можно условно представлять как некоторый ключ шифрования, который помогает:

- «зашифровать» (выполнить кодирование) строку в байты: обычно для этого используется метод `encode`;
- «расшифровать» байты (декодировать) в строку: например, с помощью метода `decode`.

Т.е. для выполнения указанных преобразований должна использоваться одна и та же кодировка.

Например:

```
userMessage = 'Hello'

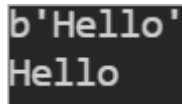
userMessageEnc=userMessage.encode('utf-8')
print(userMessageEnc)

userMessageDec=userMessageEnc.decode('utf-8')
print(userMessageDec)
```

Мы создали строку `userMessage`, которую вначале перевели в последовательность байт (`userMessageEnc`),

используя метод `encode` и кодировку `utf-8`. А затем декодировали обратно в строку `userMessageDec`, используя метод `decode` и такую же кодировку (`utf-8`).

Так как наша строка состоит из символов латиницы (входит в нижнюю часть ASCII таблицы), то в результате первой команды `print(userMessageEnc)` мы видим ASCII-символы, а не коды этих символов в Unicode (таковы особенности работы функции `print()`)



```
b'Hello'
Hello
```

Рисунок 4

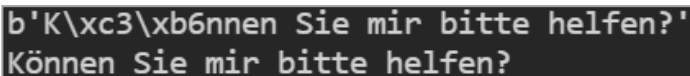
Префикс «`b`» перед первой строкой говорит о том, что фактически это байт-строка.

В случае, если наша строка будет содержать символы, отличные от латиницы (например, некоторые символы немецкого алфавита), то мы увидим коды символов в Unicode:

```
userMessage = 'Können Sie mir bitte helfen?'

userMessageEnc=userMessage.encode('utf-8')
print(userMessageEnc)

userMessageDec=userMessageEnc.decode('utf-8')
print(userMessageDec)
```



```
b'K\\xc3\\xb6nnen Sie mir bitte helfen?'
Können Sie mir bitte helfen?
```

Рисунок 5

Строка — неизменяемая последовательность символов

Во многих языках программирования имеются такие структуры данных, которые позволяют хранить набор объектов, доступных по индексу (номеру вхождения). Например, статический массив в C++ имеет фиксированный размер и содержит группу объектов одного типа.

Последовательности в Python позволяют хранить набор объектов как одного типа данных, так и разных. Причем некоторые из этих наборов (например, списки) могут также увеличиваться или уменьшаться в размерах.

Одним из самых распространенных видов последовательностей является строка — последовательность символов.

Примеры строк:

```
'Python'  
'\n'  
""  
  
"%s is number %d"  
""""hey there""""
```

Как вам уже известно, в Python существуют изменяемые и неизменяемые типы данных. Строка (**str**) является неизменяемым (**immutable**) типом данных.

Как было рассмотрено ранее, при создании переменной, вначале будет создан объект, в состав которого входит уникальный идентификатор, тип и значение. Только после этого переменная может ссылаться на уже созданный объект.

Неизменяемость типа данных означает, что созданный объект больше не изменится. Например, если мы объявим переменную `myStr = "hello"`, то будет создан объект со

значением «hello», типа `str` и идентификатором, который можно узнать с помощью функции `id()`.

```
myStr="hello"  
print(id(myStr))  
print(type(myStr))  
print(myStr)
```

```
1529165530352  
<class 'str'>  
hello
```

Рисунок 6

На первый взгляд это может показаться ограничением или недостатком, но на самом деле это не так. В последствии на примерах работы со строками мы увидим, почему это не является ограничением.

В данном примере мы использовали литералы строк.

Строковый литерал — это некоторая последовательность символов, заключенная в одинарные или двойные кавычки.

```
myStr1="hello"  
myStr2='hello'
```

Все пробелы и знаки табуляции сохранятся в строке так, как есть.

```
myStr3="What's your name?"
```

Причина наличия двух вариантов строковых литералов состоит в возможности добавлять внутрь строки

символы кавычек или символ апострофа (как в примере выше) без использования экранирования (которое мы рассмотрим далее).

Также можно указывать «многострочные» строковые литералы, используя использованием тройных кавычек (""" или ''').

В пределах тройных кавычек в свою очередь можно использовать двойные или тройные кавычки.

Например:

```
myStr4='''This is a multi-line string (text).  
        This is the first line of text.  
        This is the second line of text.  
        This is the third line of text with 'quotes'  
        '''
```

Таким образом, одним из достоинств строковых литералов является возможность записи многострочных текстов, в том числе с кавычками, апострофами внутри.

Методы строк

Для работы со строками в Python предусмотрено большое число встроенных функций и методов строк и прежде чем переходить к изучению этих разнообразных методов, необходимо рассмотреть особенности работы со строками, а именно индексацию строк.

Так как строка представляет собой упорядоченный набор символов (является последовательностью), то у каждого элемента строки (символа строки) есть свой уникальный индекс (порядковый номер). Нумерация индексов начинается с нуля. Индекс второго элемента

строки — 1 и так далее. Индекс последнего символа вычисляется как длина строки минус один.

Распределение индексов элементов строки «foobar» происходит так:

f	o	o	b	a	r
0	1	2	3	4	5

Рисунок 7

Для доступа к отдельным символам строки нужно указать имя строки, за которым следует индекс нужного символа в квадратных скобках [].

Мы можем обратиться к нужному нам символу строки следующим образом:

```
myStr="foobar"
print(myStr[0]) #'f'
print(myStr[1]) #'o'
print(len(myStr)-1) #'r'
```

В Python возможен также и доступ по отрицательному индексу, при этом отсчет идет от конца строки, т.е. последний элемент имеет индекс, равный -1, предпоследний — индекс, равный -2 и т.д.

```
myStr="foobar"
print(myStr[-1]) #r
print(myStr[-2]) #a
```

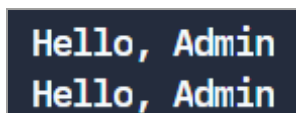
Теперь перейдем к изучению базовых операций при работе со строками.

Конкатенация (объединение) строк

Если необходимо объединить две (или более) строк в одну (новую, т.к. помним, что строки неизменяемы), то это можно выполнить с помощью символа «+», расположенного между объединяемыми объектами строк.

```
str1 = "Hello,"  
str2 = "Admin"  
  
print(str1 + str2)  
str3 = str1 + str2  
print(str3)
```

Результат:



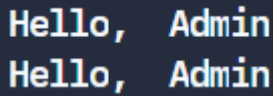
```
Hello, Admin  
Hello, Admin
```

Рисунок 8

Как уже было отмечено, в результате конкатенации строк `str1` и `str2` будет создан новый объект (который можно вывести в консоль или сохранить в отдельную переменную — `str3`).

Рассмотрим пример конкатенации трех строк:

```
str1 = "Hello"  
str2 = ", "  
str3 = " Admin"  
  
print(str1+str2+str3)  
str4=str1+str2+str3  
print(str4)
```



```
Hello, Admin  
Hello, Admin
```

Рисунок 9

Конкатенацию также называют оператором «сложения» строк.

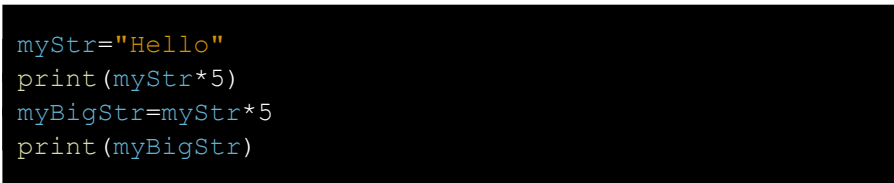
Аналогичную возможность предоставляет метод строк `.join()`, который будет рассмотрен дальше.

Дублирование строки (оператор «умножения» строк)

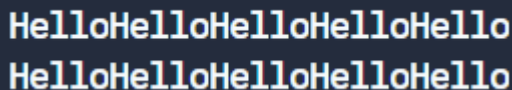
Если нам необходимо повторить строку заданное число раз (продублировать ее), то для этого воспользуемся символом «`*`» (оператор умножения), в качестве одного операнда указав строку, а второй — это количество повторений.

Как и в случае конкатенации строк, результат — это новый объект.

Например:



```
myStr="Hello"  
print(myStr*5)  
myBigStr=myStr*5  
print(myBigStr)
```



```
HelloHelloHelloHelloHello  
HelloHelloHelloHelloHello
```

Рисунок 10

Определение длины строки

Для определения длины строки (количества символов, входящих в ее состав) предусмотрена встроенная функция `len()`, аргументом которой является сама строка.

```
myStr="Hello"  
print(len(myStr)) # 5
```

Данная функция является универсальной (т.е. используется со всеми) итерируемыми типами: строками, списками, кортежами и словарями, возвращая количество элементов такой последовательности.

Теперь рассмотрим методы объекта `str` (строки). `str` (сокращение от «string») — это название типа данных (объекта) в Python, представляющего из себя последовательность символов (строку).

Вначале вспомним общий синтаксис для вызова метода некоторого объекта:

```
objName.methodName(args)
```

Данный код вызывает метод `.methodName()` объекта `objName.`, а `args` — это список аргументов (перечисляются через запятую), которые передаются методу (если такие аргументы есть).

Строки в Python являются объектами (`str`), следовательно, данный синтаксис будет таким:

```
str.methodName(args)
```

Встроенные методы объекта `str`, которые есть в Python для работы со строками удобно разделить на группы в зависимости от типа задачи, которые они реализуют.

Особенности работы со строками

Методы для изменения регистра строки

- `str.capitalize()` переводит первый символ строки `str` в верхний регистр, остальные — в нижний, возвращаемый результат — преобразованная копия оригинальной строки `str` (при этом оригинальная строка в результате работы метода не изменится, как и в случае использования рассмотренных ниже методов преобразования регистра).
- `str.lower()` переводит все буквенные символы оригинальной строки `str` в нижний регистр, возвращаемый результат — преобразованная копия строки `str`.
- `str.upper()` преобразует все буквенные символы строки `str` в верхний регистр, возвращаемый результат — преобразованная копия строки `str`.
- `str.title()` преобразует первые буквы каждого слова в строке `str` в верхний регистр (а остальные буквы слов переводит в нижний регистр), возвращаемый результат — преобразованная копия строки `str`.
- `str.swapcase()` преобразует буквенные символы строки `str`, меняя их регистр на противоположный, возвращаемый результат — преобразованная копия строки `str`.

Рассмотрим перечисленные методы на примере:

```
myStr="python was created in the late 1980's  
by Guido van Rossum."  
print(myStr.capitalize()) # Python was created in  
the late 1980's by guido van rossum.  
print(myStr.lower()) #python was created in the late  
1980's by guido van rossum.
```

```
print(myStr.upper()) #PYTHON WAS CREATED IN THE LATE
                      1980'S BY GUIDO VAN ROSSUM.

print(myStr.title()) #Python Was Created In The Late
                     1980'S By Guido Van Rossum.

print(myStr.swapcase()) #PYTHON WAS CREATED IN
                        THE LATE 1980'S BY gUIDO VAN rOSSUM.
```

Методы поиска подстроки в строке

Эти методы предоставляют различные способы поиска в целевой строке указанной подстроки (фрагмента).

Каждый метод в этой группе, кроме обязательного аргумента подстроки (фрагмента оригинальной строки, который ищется) включает два необязательных аргумента (`startIndex` и `endIndex`) для указания диапазона поиска.

При использовании этих параметров поиск проходит не во всей строке, а в ее части, начиная с индекса `startIndex` до индекса `endIndex-1` включительно. Если аргумент `startIndex` указан, а `endIndex` — нет, то поиск в строке проходит от индекса `startIndex` до конца строки.

- `str.count(pattern [, startIndex [, endIndex]])` — определяет количество вхождений фрагмента `pattern` в строку `str` (или в ее часть при задании диапазона поиска (с индекса `startIndex` ... по индекс `endIndex` ...)).

```
myStr = "Python was created in the late 1980's
        by Guido van Rossum. Python- cool!"

print(myStr.count('Python')) #2
print(myStr.count('Python', 20, 65)) #1
print(myStr.count('Python', 10)) #1
```

- `str.find(pattern [, startIndex [, endIndex]])` — используется для поиска в строке `str` нужного фрагмента `pattern`, возвращаемый результат — индекс начала первого вхождения фрагмента `pattern` в строку `str` или `-1` в случае, если фрагмент `pattern` не входит в состав `str`.

Можно также ограничить диапазон поиска с помощью параметров `startIndex` и `endIndex`.

```
myStr="Python was created in the late 1980's by Guido
van Rossum. Python- cool!"

print(myStr.find('a'))    #8
print(myStr.find('a',2,5)) # -1
```

- `str.index(pattern [, startIndex [, endIndex]])` — работа метода аналогична методу `.find()`, отличие — в вызове исключения `ValueError` в случае, когда фрагмент `pattern` не найден (не входит в состав `str`).

```
print(myStr.index('a'))    #8
print(myStr.index('a',2,5)) #ValueError - substring
                             not found
```

- `str.rfind(pattern [, startIndex [, endIndex]])` — используется для поиска в строке `str` нужного фрагмента `pattern`, начиная с конца строки `str`, возвращаемый результат — индекс начала последнего вхождения фрагмента `pattern` в строку `str` или `-1` в случае, если фрагмент `pattern` не входит в состав `str`. Для ограничения диапазона поиска можно использовать параметры `startIndex` и `endIndex`.

- `str.rindex(pattern [, startIndex [, endIndex]])` работа метода аналогична методу `.rfind()`, отличие — в вызове исключения `ValueError` в случае, когда фрагмент `pattern` не найден (не входит в состав `str`).

```
print(myStr.rfind('a'))    #48
print(myStr.rfind('a',2,20)) #14
print(myStr.rindex('a'))   #48
print(myStr.rindex('a',2,6)) #ValueError -
                             substring not found
```

Методы проверки начала (окончания) строк

Методы данной группы возвращают `True` в случае, когда оригинальная строка удовлетворяет условию, `False` — иначе. Для ограничения диапазона проверки можно использовать параметры `startIndex` и `endIndex`.

- `str.endswith(pattern [, startIndex [, endIndex]])` — определяет, заканчивается ли строка `str` указанным фрагментом `pattern`.
- `str.startswith(pattern [, startIndex [, endIndex]])` — определяет, начинается ли строка `str` с указанного фрагмента `pattern`.

```
myStr="Python was created in the late 1980's by Guido
van Rossum. Python- cool!"

print(myStr.startswith('P'))    #True
print(myStr.startswith('p'))    #False

print(myStr.startswith('w',7))  #True
```



```
print(myStr.endswith('!'))    #True
print(myStr.endswith('n',2,6)) #True
```

Методы проверки строк

Иногда нужно проверить, состоит ли строка из определенных символов, например, цифр. Для решения такого рода задач предусмотрены следующие методы (все они, как результат, возвращают значение **True** — если строка содержит нужные символы, **False** — иначе).

- **str.isalnum()** — проверяет, состоит ли строка **str** только из буквенных и цифровых символов.
- **str.isalpha()** — проверяет, состоит ли строка **str** только из буквенных символов.
- **str.isdigit()** — проверяет, состоит ли строка **str** только из цифровых символов (используется для проверки, является ли строка **str** числом).
- **str.islower()** проверяет, находятся ли все буквенные символы строки **str** в нижнем регистре (символы строки **str**, которые не являются буквой алфавита — игнорируются данной проверкой).
- **str.isspace()** проверяет, что в состав строки **str** входят только пробельные символы, к которым относятся символы пробела ' ', табуляции '\t' и перехода на новую строку '\n'.
- **str.istitle()** проверяет, начинается ли каждое слово строки **str** с символа в верхнем регистре.
- **str.isupper()** определяет, находятся ли все буквенные символы строки **str** в верхнем регистре.

- `str.islower()` определяет, находятся ли все буквенные символы строки `str` в нижнем регистре.

Рассмотрим приведенные методы на примерах:

```
myStr="Python2021"
print(myStr.isalnum()) #True
print(myStr.isalpha()) #False

myAge="16"
print(myAge.isdigit()) #True

myStr1="it was created in the late 1980's"
print(myStr1.islower()) #True

myStr2=" \t \n \t\t"
print(myStr2.isspace()) #True

myName1= "Guido van Rossum"
print(myName1.istitle()) #False

myName2= "GUIDO VAN ROSSUM"
print(myName2.isupper()) #True
```

Методы форматирования строк

Методы данной группы изменяют вывод (форматируют) строки.

- `str.center(width [, fillchar])` дополняет (расширяет) строку `str` до указанной длины `width`, возвращаемый результат — расширенная копия строки `str`. Если параметр `fillchar` указан, то он будет использован, как символ заполнения, иначе — отступы заполняются пробелами

В случае, если параметр `width` меньше или равен длине строки `str`, то строка не изменяется.

```
myStr="Python2021"

print(myStr.center(30))
print(myStr.center(30, '*'))
print(myStr.center(5))
```

Результат:

```
Python2021
*****Python2021*****
Python2021
```

Рисунок 11

- `str.expandtabs(tabsize = 8)` возвращает копию строки `str`, в которой каждый символ табуляции (`'\t'`) заменен на пробел, количество которых задается через параметр `tabsize`.

```
myStr="Python2021\n\t\tPython- cool!"
print(myStr.expandtabs(tabsize=4))
```

```
Python2021
Python- cool!
```

Рисунок 12

Если параметр `tabsize` не задан, то каждый символ табуляции заменяется на 8 пробелов.

```
myStr="Python2021\n\t\tPython- cool!"
print(myStr.expandtabs())
```

Python2021

Python- cool!

Рисунок 13

- `str.ljust(width [, fillchar])` возвращает выровненную по левому краю копию строки `str` указанной ширины `width`.

Если параметр `fillchar` задан, то он используется для заполнения недостающего количества символов, иначе используется символ пробела.

В случае, если параметр `width` меньше или равен длине строки `str`, то строка не изменяется.

```
myStr="Python- cool!"
print(myStr.ljust(20))
print(myStr.ljust(20, '@'))
print(myStr.ljust(5))
```

```
Python- cool!
Python- cool!@@@@@@@@
Python- cool!
```

Рисунок 14

- Для аналогичного выравнивания строки по правому краю в поле указанной ширины используется метод `str.rjust(width [, fillchar])`.
- `str.lstrip([characters])` возвращает копию строки `str`, удаляя начальные символы (слева), указанные в качестве аргумента `characters`. Если параметр `characters` не указан, то удаляются символы пробелов.

- Для аналогичного удаления завершающих символов (или пробелов) в строке справа предусмотрен метод `str.rstrip([characters])`.
- Если необходимо удалить пробелы (или указанные символы) и с левой, и с правой стороны строки, то используется метод `str.strip([characters])`.

```
myStr="          Python- cool!          "
print(myStr.lstrip())
print(myStr.rstrip())
print(myStr.strip())

myStr="@;          Python- cool!          @"
print(myStr.lstrip('@;'))
print(myStr.rstrip('@'))
print(myStr.strip('@'))
```

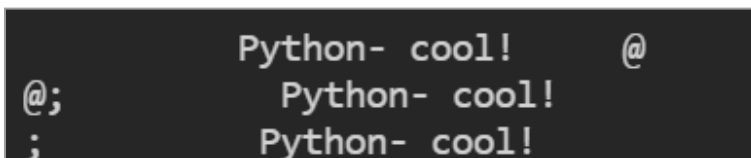


Рисунок 15

Также для работы со строками предусмотрен интересный метод `str.zfill(width)`, который дополняет строку слева символами «0» ширины `width`.

Если строка `str` содержит какой-либо символ перед цифрами, то он остается в левой части строки и заполнение нулями происходит после него.

```
myStr="123"
print(myStr.zfill(5))
```

```
myStr="+123"
print(myStr.zfill(5))
```

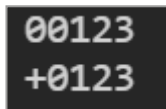


Рисунок 16

Срез строки

Часто в различных задачах по обработке текстовой информации нам необходимо извлекать определенную часть строки. Например, если строка содержит имя и фамилию студента «*John Smith*», а нам для дальнейшей работы нужна только фамилия студента «*Smith*», т.е. фрагмент строки

В Python для извлечения фрагмента строки можно воспользоваться одним из следующих способов:

- индексированием — для получения одного символа строки;
- срезом строки — для получения как отдельного символа, так и фрагмента строки;
- функциями стандартной библиотеки Python.

Особенности индексирования строк мы уже рассмотрели ранее. Срез строки — способ извлечения фрагмента строки (подстроки) за одной действие. Варианты формирования срезов можно разделить на две категории:

- базовые срезы с указанием границ среза (обоих или одной из);
- расширенные срезы с указанием шага индексирования.

Рассмотрим синтаксис базовых срезов:

- `str[a:b]` — обе границы (`a` — левая и `b` — правая) среза заданы явно.

В этом случае из строки `str` извлекается подстрока, начиная с позиции `a` до позиции `b` (но не включая `b`).

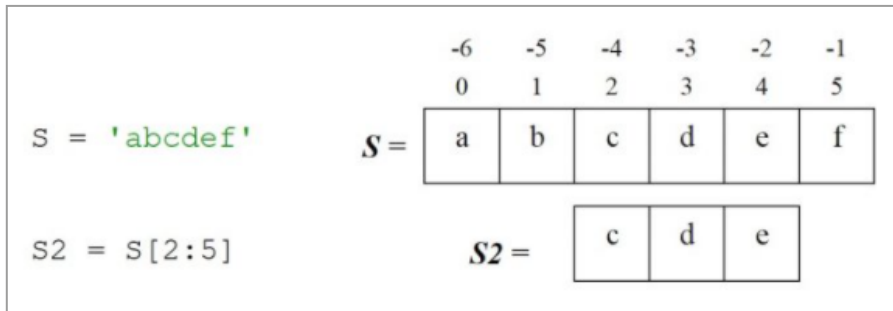


Рисунок 17

На рисунке 17 приведен состав строки с указанием индексов.

Первый срез (с использованием положительных индексов) `S2[2:5]` извлекает из строки `S` фрагмент, начиная с символа строки `S` с индексом 2 («с») и до символа с индексом 5, но не включая его. Последний символ фрагмента будет «е», т.е. по индексу 4 (5-1).

Возможно также использование отрицательных индексов. Вспомним, что при индексации строк мы можем также использовать отрицательные индексы, которые уменьшаются справа налево от -1 до -длина строки: последний элемент строки имеет индекс -1, предпоследний — на один меньше (-2) и так далее.

Также возможно сочетание отрицательных и положительных индексов в одном срезе.

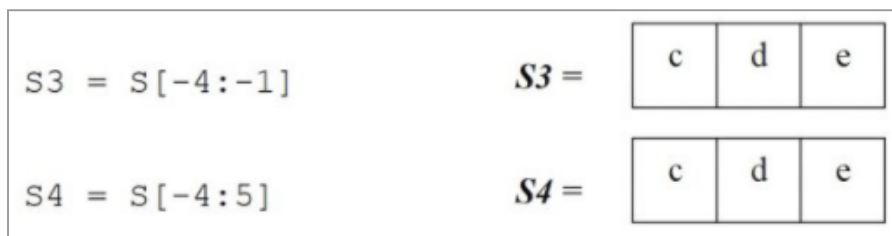


Рисунок 18

Срез **S3** извлекает из строки **S** фрагмент, начиная с символа строки **S** с индексом **-4** («**c**») и до символа с индексом **-1**, но не включая его. Последний символ фрагмента будет «**e**», т.е по индексу **-2** (**-1-1**).

- **str[:b]** — указана только правая граница среза.

В этом случае из строки **str** извлекается подстрока, начиная с начала строки и до позиции **b** (но не включая **b**).

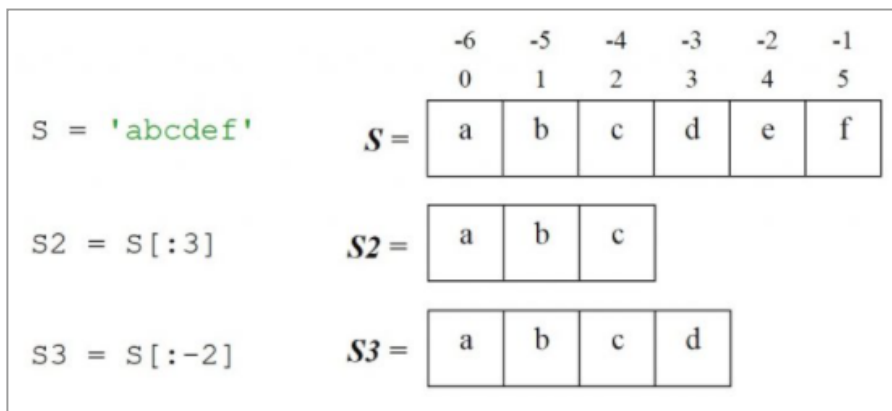


Рисунок 19

- **str[a:]** — указана только левая граница среза.

В этом случае из строки **str** извлекается подстрока, начиная с позиции **a** и конца строки.

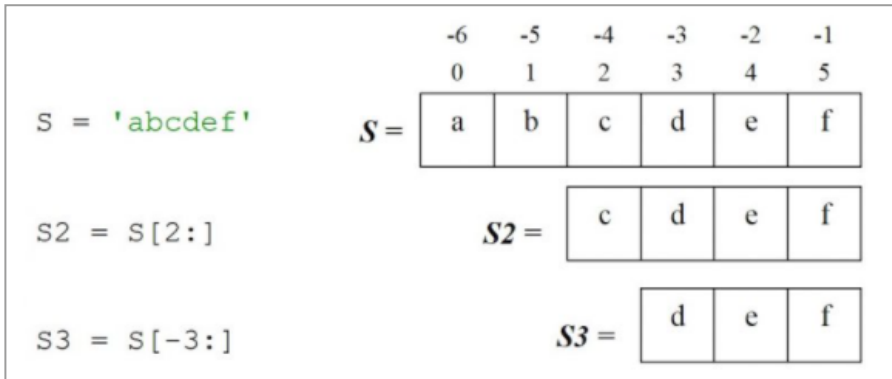


Рисунок 20

- `str[:]` — не указана границы среза. В этом случае извлеченная подстрока будет копией строки `str`, начиная от начала и до конца.

Рассмотрим еще примеры таких срезов:

```
myStr="Python-cool!"
print(myStr[1:3]) #yt
print(myStr[-5:-2]) #coo
print(myStr[-5:11]) #cool

print(myStr[:6]) #Python
print(myStr[:-1]) #Python-cool
print(myStr[6:]) #-cool!
print(myStr[-5:]) #cool!
```

В рассмотренных способах формирования срезов в срез попадает каждый символ из оригинальной строки от левой до правой границы (но не включая ее), т.е. шаг среза равен единице.

Расширенные срезы предоставляют возможность задать величину шага.

Такой вариант срезов имеет следующий синтаксис:

`str[a : b : k]`,

где

`a` и `b` — левая и правая границы среза;

`k` — шаг по индексу.

При формировании такого среза величина шага `k` добавляется к индексу каждого элемента, который извлекается из строки `str`.

При этом шаг `k` может быть, как положительным, так и отрицательным:

- если `k ≥ 0`, то срез формируется слева направо (от `a` до `b`);
- если `k < 0`, то строка срез формируется справа налево (от `b` до `a`).

Рассмотрим возможные варианты формирования таких срезов.

- `str[a : b : k]` — извлекаются все элементы строки `str`, начиная с индекса `a`, завершая индексом `b - 1` включительно, со смещением (шагом) `k`.
- `str[a : : k]` — извлекаются все элементы строки `str`, начиная с индекса `a` и до конца строки со смещением (шагом) `k`.
- `str[: b : k]` — извлекаются все элементы строки `str`, начиная с начала строки и завершая индексом `b - 1` включительно, со смещением (шагом) `k`.
- `str[: : k]` — извлекаются все элементы строки `str`, начиная с начала строки и до конца со смещением (шагом) `k`.

Рассмотрим примеры таких срезов:

```

myStr="1234567890"

print(myStr[2:8:2]) #357
print(myStr[8:2:-2]) #975
print(myStr[::-1]) #0987654321

print(myStr[5::2]) #680
print(myStr[-1::-2]) #08642
print(myStr[:len(myStr):3]) #1470

```

Экранированные последовательности

Предположим, что нам нужно отобразить (согласно некоторому макету) следующий текстовый фрагмент:

```

Python books:
    'Python Programming: An Introduction to Computer Science'
    'The Python Guide for Beginners'

```

Рисунок 21

Если бы мы набирали его в каком-то текстовом редакторе, то после первой строки мы были нажали клавишу «**Enter**» после первой и второй строки текста, чтобы перейти на новую, а в начале второй и третьей использовали бы клавишу «**Tab**».

Таким образом при создании такого текстового фрагмента в виде строки нам понадобятся символы-коды, которые соответствуют таким действиям, как переход на новую строку и табуляция. То есть механизму-обработчику символьной последовательности нужно особым образом указать, что в данном месте не следует отображать символ, что это вообще не печатаемый символ, а указание выполнить определенные действия.

Для реализации такого подхода в языках программирования существуют экранированные последовательности.

Экранированные последовательности представляют собой набор текстовых символов, начинающихся (и распознающимися механизмом — обработчиком) с символа «\» (*backslash*), «\» является признаком того, что начинается экранированная последовательность, следующие за ним символы (и сам символ «\») не печатаются, а воспринимаются обработчиком, как код действия.

Отдельные символы такой последовательности не имеют смысла для обработчика, поэтому и называется «последовательность», т.к. распознается именно набор символов в указанном порядке после «\».

Экранированные последовательности также еще называют управляющими последовательностями или *Escape*-последовательностями (англ. *Escape Sequence*).

Общий синтаксис для представления экранированных последовательностей следующий:

```
"\символ (символы)"
```

Для того, чтобы понять, как именно экранированные последовательности влияют на отображение строк, рассмотрим наиболее распространенные из них.

- `\n` — переход на начало новой строки (*newline*). С помощью данной комбинации мы можем переносить следующий за ним фрагмент на новую строку

Например:

```
myStr="There is no shortage of material to learn  
Python.\nThe following books might serve
```

```

as a starting point, in the order specified:
\n'Python Programming: An Introduction to
Computer Science' by John M. Zelle, Ph.D.
\n'The Python Guide for Beginners' by Renan
Moura.\n'Effective Python' by Brett Slatkin"
print(myStr)

```

Результат:

```

There is no shortage of material to learn Python.
The following books might serve as a starting point, in the order specified:
'Python Programming: An Introduction to Computer Science' by John M. Zelle, Ph.D.
'The Python Guide for Beginners' by Renan Moura.
'Effective Python' by Brett Slatkin

```

Рисунок 22

- `\t` — горизонтальная табуляция (вывод последующих символов начнется с отступом по горизонтали)

Например:

```

print("Python books:")
print("\t'Python Programming: An Introduction
to Computer Science'")
print("\t'The Python Guide for Beginners'")

```

Результат:

```

Python books:
    'Python Programming: An Introduction to Computer Science'
    'The Python Guide for Beginners'

```

Рисунок 23

- `\x` — признак шестнадцатеричного код символа, например, `\x2C` — это представление шестнадцатеричного кода символа запятой:

Например:

```
myStr="\x2C"
print(myStr) # ,
```

- `\ooo` — используется для представления восьмеричного кода символов (`ooo` — условное обозначение трех знакомест для цифр кода), например, `\053` — это представление шестнадцатеричного код символа плюс:

Например:

```
myStr="\053"
print(myStr) # +
```

Символ «`\`» используется (занят) экранированными последовательностями, однако есть ситуации, когда нам нужно его отобразить в строке именно, как символ «`\`».

Также есть и другие символы, используемые синтаксисом Python для других целей, например, символ двойных кавычек или символ апострофа.

Для отображения (печати) таких символов используется подход, называемый экранированием. При этом непосредственно перед специальным символом, который необходимо отобразить, вставляется символ «`\`»:

- `\\` — для отображения символа «`\`»;
- `\'` — для отображения символа апострофа;
- `\"` — для отображения символа двойных кавычек.

Например:

```
print('\ Python Programming: An Introduction to
      Computer Science\')
```

```
print("\Python Programming: An Introduction to
      Computer Science")
print("15\\2")
```

Результат:

```
'Python Programming: An Introduction to Computer Science'
"Python Programming: An Introduction to Computer Science"
15\2
```

Рисунок 24

«Сырые» строки

Мы уже знаем, в Python escape-последовательности всегда начинаются с символа «\» ([backslash](#)). Это удобно для форматирования, но что, если мы хотим отобразить этот символ в строке? Для этого, как было рассмотрено ранее, используется механизм экранирования, при котором перед специальным символом, который необходимо отобразить, вставляется символ «\».

Например:

```
myStr="Backslash symbol: \\"
print(myStr) # Backslash symbol: \
```

Теперь предположим, что внутри строки нам необходимо отобразить два таких символа «[\n](#)» и «[\t](#)». Набор символов «[\n](#)» и «[\t](#)» — это escape-последовательности, которые реализуют переход на новую строку и табуляцию, соответственно, эти символы не отображаются. Опять применяем механизм экранирования для их отображения:

```
myStr="In Python strings, the backslash '\\' is
      a special character. It is used in representing
      certain whitespace characters: '\\t' is a tab,
      '\\n' is a newline"

print(myStr)
```

Нетрудно заметить, что данная строка тяжело воспринимается, т.к. нужно помнить о каждой позиции строке (например, «\\»), которая будет печататься не так («\»), как она выглядит в коде.

Ситуация усугубляется, если в строке есть несколько подряд идущих символов «\», приходится точно высчитывать, какой из «\» используется для экранирования, а какой будет фактически отображаться при выводе.

Для решения этой проблемы в Python предусмотрены «Сырые» строки (*raw strings*). **Raw strings** «рассматривают» обратную косую черту (\) как буквальный символ. Это полезно в случаях, когда нам необходимо работать со строкой, содержащую обратную косую черту (\), и мы не хотим, чтобы она рассматривалась как escape-последовательность.

«Сырые» строки Python имеют префикс «r» или «R». Просто добавьте к строке префикс «R» или «r», и она будет рассматриваться как «**raw strings**». Рассмотрим на примерах:

1)

```
normalStr = "This is a \nnormal string"
print(normalStr)

rawStr = r"This is a \n raw string"
print(rawStr)
```


Результат:

```
This is a
normal string
This is a \n raw string
```

Рисунок 25

2)

```
normalText='''Python Arithmetic Operators:\n
    Arithmetic operators are used to perform
    mathematical operations like addition,
    subtraction, multiplication and division.\n
    \tThere are 7 arithmetic operators in Python:
    \t\tAddition +\n
    \t\tSubtraction -\n
    \t\tDivision /\n
    \t\tModulus %\n
    \t\tExponentiation **\n
    \t\tFloor division //\n'''

rawText=r'''Python Arithmetic Operators:\n
    Arithmetic operators are used to perform
    mathematical operations like addition,
    subtraction, multiplication and division.\n
    \tThere are 7 arithmetic operators in Python:
    \t\tAddition +\n
    \t\tSubtraction -\n
    \t\tDivision /\n
    \t\tModulus %\n
    \t\tExponentiation **\n
    \t\tFloor division //\n
    '''

print(normalText)
print(rawText)
```

Результат:

```

Python Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations like addition,
subtraction, multiplication and division.

    There are 7 arithmetic operators in Python:
        Addition +

        Subtraction -

        Division /

        Modulus %

        Exponentiation **

        Floor division //

Python Arithmetic Operators:\n
Arithmetic operators are used to perform mathematical operations like addition,
subtraction, multiplication and division.\n
\tThere are 7 arithmetic operators in Python:
\t\tAddition +\n
\t\tSubtraction -\n
\t\tDivision /\n
\t\tModulus %\n
\t\tExponentiation **\n
\t\tFloor division //\n

```

Рисунок 26

Однако не каждая строка (последовательность символов в кавычках) является корректной «сырой» строкой.

«Сырая» строка, содержащая только один символ «\» недопустима. Точно так же «сырые» строки с нечетным числом символов «\».

Например:

```

errRawStr1 = r'\ '
errRawStr2 = r'123\'
errRawStr3 = r'abc\\\'

```

В этом случае интерпретатор выдаст следующую ошибку:

```
errRawStr1 = r'\'  
            ^  
SyntaxError: unterminated string literal (detected at line 1)
```

Рисунок 27

Форматированный вывод

Мы уже знаем, что для вывода (совмещения) в одной строке некоторых текстовых фрагментов (строковых литералов) и строковых переменных (нескорых данных, которые образуются и изменяются в ходе работы программы) можно использовать либо конкатенацию, либо (если нужен только вывод) функцию `print()` с несколькими аргументами.

Например:

```
userLogin=input("Your login: ")  
print("Welcome,", userLogin, ". Let's start game!")  
  
strMsg="Dear, "+userLogin+". Game over!"  
print(strMsg)
```

Результат:

```
Welcome, Admin . Let's start game!  
Dear, Admin. Game over!
```

Рисунок 28

Помимо не очень удобного способа подстановки в случае использования функции `print()` мы видим лишний

символ пробела после логина пользователя (это связано с особенностями работы функции `print()`, которая при задании ей нескольких аргументов разделяет их значения при выводе пробелом). Конечно, такое поведение по умолчанию функции `print()` мы можем изменить, задав ей дополнительный атрибут `sep=""`, но это еще больше снижает читабельность и понимание кода.

```
print("Welcome,", userLogin,  
      ". Let's start game!", sep="")
```

Более того, использование атрибута `sep=""` удалило пробел между «Welcome!» и логином пользователя.

Более удобным для такого рода форматирования строк является использование метода `format()`.

В случае, когда в строковый литерал нужно подставить значение только одной переменной, то можно использовать такой синтаксис:

```
str.format(value)
```

- `str` — строка-шаблон с заполнителем `{}` (место в шаблоне, куда следует поместить значение, переданное аргументом метода)
- `value` — аргумент-значение, которое будет подставляться в указанное место строки-шаблона.

Например:

```
userLogin=input("Your login: ")  
strMsg="Welcome, {}. Let's start game!".format(userLogin)  
print(strMsg)
```

Результат:

```
Your login: Admin
Welcome, Admin. Let's start game!
```

Рисунок 29

Если же переменных, значения которых нужно вставить в шаблон, несколько, то количество заполнителей должно соответствовать количеству аргументов метода `format()`.

При этом заполнители можно идентифицировать с помощью именованных индексов `{indexName}`, нумерованных индексов `{index}` или оставить пустыми `{}`, как в примере выше.

Рассмотрим все эти случаи на примере.

```
strMsg = "My name is {}, I'm {}".format("Student",25)
print(strMsg) # My name is Student, I'm 25
```

В случае использования пустых заполнителей `{}` в них будут подставлены аргументы метода `format()` четко в том порядке, как они перечислены в методе.

То есть, если поменять местами литералы «Student» и 25, то строка потеряет всякий смысл:

```
strMsg = "My name is {}, I'm {}".format(25,"Student")
print(strMsg) # My name is 25, I'm Student
```

Для предотвращения таких ситуаций рекомендуется идентифицировать заполнители с помощью именованных

индексов `{indexName}` или нумерованных индексов `{index}`.

Например, используем нумерованные индексы:

```
strMsg1 = "My name is {0}, I'm {1}".  
format("Student",25)  
print(strMsg1) # My name is Student, I'm 25  
  
strMsg2 = "I'm {1}. My name is {0}".format("Student",25)  
print(strMsg2) # I'm 25. My name is Student
```

Или именованные индексы, которые обеспечивают еще более четкое позиционирование даже при изменении порядка аргументов:

```
strMsg3 = "My name is {name}, I'm {age}".  
format(name="Student",age=25)  
print(strMsg3) # My name is Student, I'm 25  
  
strMsg4 = "My name is {name}, I'm {age}".  
format(age=25,name="Student")  
print(strMsg4) # My name is Student, I'm 25
```

Также метод `format()` обеспечивает нам дополнительные возможности для вывода вещественных чисел, позволяя четко задать количество цифр после запятой и общее количество знаков под числом.

Рассмотрим на примере:

```
strMsg = "Your salary is {0:9.2f}".format(200.846)  
  
print(strMsg) # Your salary is      200.85
```

Заполнитель `{0}`, который получает значение из первого аргумента метода (т.к. индекс `0`) заменяется на значение `200.846`, которое предварительно подвергается форматированию по шаблону `9.2f`:

- `f` — это спецификатор, который указывает на тип аргумента (`float`, вещественный)
- количество цифр после запятой сокращается до двух (если у оригинального числа цифр после запятой больше, то происходит округление, в нашем примере до `200.85`);
- число в шаблоне `9.2f` до точки (цифра `9` здесь) определяет общее количество знакомест (позиций), отводимое под вывод всего числа;
- если чисто знакомест больше длины числа (в нашем случае длина числа вместе с символом «.» после округления `6`), то свободные позиции заполняются пробелом и число как бы отодвигается вправо (в нашем примере было вставлено `3` пробела перед числом);
- если число знакомест меньше длины числа, то эта часть формата игнорируется (данная ситуация показан в примере ниже)

```
strMsg = "Your salary is {0:3.2f}".format(200.846)
print(strMsg) # Your salary is 200.85
```

Помимо вещественных чисел также предусмотрены возможности форматированного вывода десятичных чисел, значений в двоичном, восьмеричном, шестнадцатеричном формате и проч., для чего предусмотрены соответствующие спецификаторы.

Наиболее распространенные типы спецификаторов приведены в следующей таблице.

Таблица 1

Специ- фикатор	Описание
:b	Двоичный форма
:c	Преобразует значение в соответствующий символ Юникода
:d	Десятичный формат
:o	Восьмеричный формат
:x	Шестнадцатеричный формат, нижний регистр

Спецификаторы помещаются при этом внутри заполнителей {}, как в примерах выше.

Рассмотрим их действие на примерах.

```

userNumber = int(input("Your number? ")) #255
myStrB = "The binary representation of a number {n}
         is {n:b}".format(n=userNumber)
print(myStrB) #The binary representation of a number
              255 is 11111111

myStrO="The octal representation of a number {n}
        is {n:o}".format(n=userNumber)
print(myStrO) #The octal representation of a number
              255 is 377

myStrH="The Hex representation of a number {n}
        is {n:x}".format(n=userNumber)
print(myStrH) #The Hex representation of a number
              255 is ff

```

Дополнительные возможности форматирования предусмотрены для чисел со знаками:

Таблица 2

Тип форматирования	Описание
<code>:-</code>	Отображает знак минус только для отрицательных значений
<code>:+</code>	Отображает знак плюс только для положительных значений
<code>:символ пробела</code>	Вставляет дополнительный пробел перед положительными числами (и знак минус перед отрицательными числами)

Примеры:

```
myStr1="The number1 range from {0:-} to {1:-}".
format(-10,10)
print(myStr1) #The number1 range from -10 to 10

myStr2="The number2 range from {0:+} to {1:+}".
format(-20,50)
print(myStr2) #The number2 range from -20 to +50

myStr3="The number3 range from {0: } to {1:
}".format(-30,30)
print(myStr3) #The number3 range from -30 to 30
```

Теперь рассмотрим, какие возможности метод `format()` предоставляет нам для выравнивания чисел и строк.

Таблица 3

Тип форматирования	Описание
<code>:<</code>	Выравнивает по левому краю (в пределах доступного пространства)
<code>:></code>	Выравнивает по правому краю (в пределах доступного пространства)
<code>:^</code>	Выравнивает по центру (в пределах доступного пространства)

Например:

```
#установим доступное пространство для значения
до 10 символов.
myStr1 = "You have {:<10} points."
print(myStr1.format(12)) #You have 12           points.

myStr2 = "You have {:>10} points."
print(myStr2.format(12)) #You have           12 points.

myStr3 = "You have {:^10} points."
print(myStr3.format(12)) #You have      12      points.
```

При выравнивании чисел после типа форматирования можно указывать спецификатор

```
myStr1 = "Number is {:<10.2f}!"
print(myStr1.format(34.8256)) #Number is 34.83      !

myStr2 = "Number is {:>10.2f}!"
print(myStr2.format(34.8256)) #Number is           34.83!

myStr3 = "Number is {:^10.2f}!"
print(myStr3.format(34.8256)) #Number is    34.83    !
```

Рассмотренные типы форматирования также полезны при выравнивании строк:

```
myStr1 = "Your login is {:<20}!"
print(myStr1.format("Admin")) #Your login is Admin   !

myStr2 = "Your password is {:>20}!"
print(myStr2.format("12345")) #Your password is    12345!

myStr3 = "Your secret word is {:^15}!"
print(myStr3.format("IT")) #Your secret word is     IT   !
```

Модуль string

В предыдущих разделах мы познакомились со многими полезными методами объекта строки (`str`). Большая часть из них ранее была реализована в отдельном модуле `string`. Однако в связи с высокой частотой их использования они были перенесены в методы объекта `str`, чтобы не нужно было выполнять предварительный импорт модуля.

Сейчас модуль `string` содержит набор констант, утилит и классов для работы с объектами `str` (например, возможности создания пользовательских шаблонов для настройки собственного форматирования строк).

Для использования возможностей модуля `string` вначале необходимо выполнить его импорт таким образом:

```
import string
```

Рассмотрим вначале перечень констант, предоставляемых модулем:

Таблица 4

Имя константы	Описание	Значение
<code>string.ascii_letters</code>	буквенные символы латинского алфавита (в верхнем и в нижнем регистре)	<code>abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ</code>
<code>string.ascii_lowercase</code>	буквенные символы латинского алфавита (в нижнем регистре)	<code>abcdefghijklmnopqrstuvwxyz</code>
<code>string.ascii_uppercase</code>	буквенные символы латинского алфавита (в верхнем регистре)	<code>ABCDEFGHIJKLMNOPQRSTUVWXYZ</code>

Имя константы	Описание	Значение
<code>string.digits</code>	символы цифр десятичной системы исчисления	0123456789
<code>string.hexdigits</code>	символы цифр шестнадцатеричной системы исчисления	0123456789 abcdefABCDEF
<code>string.octdigits</code>	символы цифр восьмеричной системы исчисления	01234567
<code>string.punctuation</code>	символы пунктуации	!"#\$%&'()*+,-./ :;<=>?@[\]^_`{ }~
<code>string.whitespace</code>	символы, которые считаются пробельными: переход на новую строку, табуляция и т.д.	
<code>string.printable</code>	печатные символы (объединение множеств <code>digits</code> , <code>ascii_letters</code> , <code>punctuation</code> и <code>whitespace</code>)	

Каждая из рассмотренных констант является строкой, т.е., например, `string.digits` это строка «0123456789».

Рассмотрим пример практического применения данных констант.

Предположим, что нам необходимо сгенерировать случайным образом логин пользователя заданной длины (например, 6 символов), который должен состоять только из символов латинского алфавита в нижнем регистре, и пароль (8 символов), который должен состоять из символов латинского алфавита и десятичных цифр.

Для этого мы воспользуемся методом `sample()` модуля `random`, который извлекает случайные элементы из

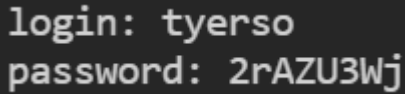
последовательности в заданном количестве. Для объединения полученного набора символов можно использовать метод строки `join()`.

```
import string
import random

userLogin = "".join(random.sample((string.ascii_lowercase), 6))
userPass = "".join(random.sample((string.ascii_letters +
                                   string.digits), 8))

print("login:", userLogin)
print("password:", userPass)
```

Результат:



```
login: tyerso
password: 2rAZU3Wj
```

Рисунок 30

В состав модуля `string` входит единственная функция `capwords()`:

```
capwords(strObj, sep = None)
```

где

- `strObj` — это строка, которая подвергается обработке;
- `sep` — необязательный аргумент, задающий символ-разделитель в строке-результате.

Эта функция вначале разделяет строку `strObj` на слова, затем первый символ каждого слова переводит в верхний

регистр после соединяет преобразованные слова в новую строку, используя указанный разделитель.

Если аргумент `sep` не указан (его значение по умолчанию `None`), то начальные и конечные пробелы (если они есть) будут удалены из строки `strObj`, а слова новой строки будут соединены символом одного пробела.

```
import string
myStr = " We have BEEN happy\n to welcome\n\n back
        OLD friends, \n\n\nand to make new ones  "

#We Have Been Happy To Welcome Back Old Friends,
And To Make New Ones
print(string.capwords(myStr))
```

Также модуль `string` предоставляет нам два класса: `Formatter` и `Template`.

Функциональность первого (класса `Formatter`) практически совпадает с возможностями, которые предоставляет рассмотренный ранее метод строк `format()`, т.е. данный класс является альтернативным решением для создания и настройки форматирования строк так, как это необходимо разработчику.

```
from string import Formatter
formatter = Formatter()

print(formatter.format('{userLog}',
                      userLog = 'Admin')) #Admin
print(formatter.format('{} {userLog}', 'Welcome, ',
                      userLog = 'Admin')) #Welcome, Admin
print('{} {userLog}'.format('Welcome, ',
                             userLog = 'Admin')) #Welcome, Admin
```

Как видно из примера, первый аргумент метода `format()` класса `Formatter` — это строка-шаблон с заполнителями, а следующие аргументы — набор значений (или переменные), которые будут подставлены в заполнители.

В последней строке нашего кода приведен пример использования метода `format()` для строки-шаблона, который обеспечил такой же результат.

Второй класс `Template` полезен при создании строк-шаблонов с последующей подстановкой в них необходимых данных.

После его импорта из модуля `string` шаблон создается конструктором класса, которому в качестве аргумента передается строка с именованными заполнителями в нужных позициях.

```
from string import Template

t = Template('$userName has the rights to $userRights
             in the $appName')
```

В нашем примере в строке три позиции с заполнителями: `$userName`, `$userRights`, `$appName`. Далее необходимо применять шаблон, используя метод `.substitute()`, которому передаются данные для вставки в заполнители:

```
resStr = t.substitute(userName='Admin',
                      userRights = 'edit',
                      appName='SuperApp.')

print(resStr) #Admin has the rights to edit in
              the SuperApp.
```

Регулярные выражения, модуль re

Одной из наиболее распространенных задач при работе с текстовыми данными является задачи поиска и проверки на соответствие некоторым условиям.

Ранее мы уже рассмотрели набор методов объекта строка, с помощью которых можно реализовать поиск (методы `.find()`, `.index()` и их вариации). Однако их возможности ограничены тем фактом, что ключом поиска является четко определенный паттерн (подстрока). Также, если хотя бы один из символов в паттерне находится не в том регистре по сравнению с искомой строкой, то результат поиска будет неудачным. А что, если паттерн заранее невозможно определить? И есть только набор правил, которому строка должна соответствовать. Например, логин пользователя должен не просто состоять только из латинских символов и цифр, а содержать хотя бы одну букву в верхнем регистре.

Для решения подобных задач можно на основе имеющихся правил (ограничений) составить шаблон и выполнять его поиск в строке, которую необходимо проверить на соответствие.

Именно для создания и использования таких шаблонов во многих языках программирования (в том числе и в Python) предусмотрены регулярные выражения.

Фактически, регулярные выражения («*regular expressions*», «*regex*», «*regexp*», «*RE*») — это специальная последовательность символов (строка), которая задает шаблон.

Такой шаблон может применяться для поиска, замены, упорядочивания, извлечения данных из строки (или текстового файла).

В состав шаблона, который мы создаем с помощью регулярного выражения, входят как обычные символы, так и специальные символы (или их последовательности).

Предположим, что нам нужно определить входит ли слово «*cat*» в состав фразы «*My Supercat*». В этом случае шаблон поиска будет просто «*cat*».

А теперь во фразе «*My1 S123upercat*» найдем слова, в состав которых входят две цифры подряд. Шаблон будет «`\d\d`», который обнаружит слово «*S123upercat*». Во этом примере для представления правила «одна любая цифра» используется последовательность «`\d`».

Работа с регулярными выражениями в Python происходит средствами модуля `re` (который первоначально необходимо импортировать).

Однако, вначале нам необходимо познакомиться с основами синтаксиса RE.

Например, если нам нужно определить, входит ли фрагмент «*student*» в логин пользователя, то мы будем использовать символы только из слова «*student*» и в нужном порядке. А что если правила для логина такие: начинается со слова «*student*» и дальше идут три любые цифры? В этом случае нам понадобится что-то, представляющее ситуацию «любая цифра» и возможность задавать нужное количество таких экземпляров ситуации (три цифры в нашем примере). Для решения именно таких проблем в синтаксисе регулярных выражений существуют метасимволы.

В состав RE входят не только отдельные специальные символы (метасимволы), но и их последовательности. К метасимволам относятся символы точки «`.`», символ «`^`»,

знак доллара «\$», символ «*», плюс «+», вопросительный знак «?», открывающаяся, закрывающаяся фигурные «{ }», квадратные «[]», круглые «()» скобки, обратный слэш «\», вертикальная черта «|».

Метасимволы в РЕ являются некоторыми управляющими конструкциями. Рассмотрим назначение каждого из них.

Таблица 5

Метасимвол	Назначение
.	Задание (представление) одного произвольного символа (кроме символа новой строки)
^	Признак начала последовательности
\$	Признак окончания последовательности
*	Обозначает любое количество повторений одного символа (0 или более), предшествующего символу «*»
+	Обозначает любое количество повторений одного символа (1 или более), предшествующего символу «+»
?	Обозначает ноль или одно повторений одного символа, предшествующего символу «?»
{n}	Обозначает заданное число (n) повторений одного символа, предшествующего символу «{»
[]	Используется для задания любого символа, из перечисленных внутри []
\	Используется для экранирования метасимволов
	Соответствует логическому ИЛИ (значение до или после символа « »)
()	Для создания группы символов (выражение внутри () рассматривается как один элемент)

Например, есть требование, что логин пользователя с правами администратора должен содержать слово «*admin*» (т.е логины «*admin12*», «*admin12a*» «*54admin*», «*adminadmin*» — допустимы, а логины «*user_adm23*», «*adm60*» — нет)

Регулярное выражение, описывающее это требование, будет таким:

```
'(admin)+ '
```

Так как нам нужно проверить наличие целой группы символов и в указанном порядке (слово «*admin*»), то мы использовали круглые скобки «(*admin*)», а для указания того, что слово «*admin*» должно встретиться хотя бы один раз — знак «+» после группы.

Рассмотрим следующий пример: пароль пользователя должен состоять только из символов латиницы в нижнем регистре (длина пароля — произвольная, т.е. пароль из одного символа — также подходит).

Ранее нами были изучены возможности и функциональность модуля [string](#), в котором есть полезная для данной задачи константа — [string.ascii_lowercase](#), представляющая собой строку из символов латинского алфавита в нижнем регистре.

Так как в состав пароля может входить любой из этих символов, то нам понадобится метасимвол `[]`. Шаблон соберем с помощью конкатенации строк таким образом:

```
'['+string.ascii_lowercase+'] '+'+'
```

Теперь рассмотрим шаблоны (которые входят в состав RE) для представления одного символа.

Таблица 6

Шаблон	Назначение
<code>\d</code>	Соответствует одной десятичной цифре
<code>\D</code>	Соответствует одному любому символу, кроме десятичной цифры
<code>\s</code>	Соответствует одному (любому) пробельному символу
<code>\S</code>	Соответствует одному (любому) символу, который не относится к пробельным
<code>\w</code>	Соответствует любому буквенно-цифровому символу или символу нижнего подчеркивания («_»)
<code>\W</code>	Соответствует любому не буквенно-цифровому символу и не символу нижнего подчеркивания («_»)
<code>[..]</code>	Соответствует любому одному из символов, перечисленных в скобках, можно также указывать диапазоны символов (например, <code>[0-5]</code> — любая цифра от 0 до 5). Внимание! метасимволы внутри <code>[]</code> теряют свое специальное значение и обозначают просто символ. Например, точка внутри <code>[]</code> будет обозначать именно точку, а не любой символ
<code>[^..]</code>	Соответствует любому одному символу, кроме перечисленных в скобках или кроме тех, что попадают в указанный диапазон
<code>\b</code>	Соответствует началу или концу слова (т.е. слева от <code>\b</code> пусто или не буквенный символ, справа буква и наоборот для конца слова). В отличие от предыдущих шаблонов соответствует позиции, а не символу
<code>\B</code>	Соответствует «внутреннему» (неграничному) символу слова (т.е. слева и справа от <code>\B</code> буквенные символы, или слева и справа от <code>\B</code> не буквенные символы)

Рассмотрим примеры шаблонов с метасимволами и результаты их применения к строкам

Таблица 7

Пример шаблона	Описание шаблона	Примеры строк, соответствующих шаблону
<code>course.topic.part.a</code>	Подходят все строки, содержащие фрагменты <code>course</code> , <code>topic</code> , <code>part</code> , <code>a</code> в указанном порядке. Между фрагментами должен находиться один произвольный символ, кроме символа новой строки	<code>course1topic2part7a</code> <code>courseAtopic1partDa</code> <code>mycourse1topic5partCapital</code>
<code>course\d\d</code>	Подходят все строки, содержащие фрагмент <code>course</code> , сразу после которого должны идти две десятичные цифры	<code>course25</code> <code>mycourse01</code> <code>mycourse01part23</code>
<code>part\D001</code>	Подходят все строки, содержащие фрагмент <code>part</code> , сразу после которого должен быть один любой символ, кроме десятичной цифры, после которого идет фрагмент <code>001</code>	<code>part-001</code> <code>part@001</code>
<code>user\s24</code>	Подходят все строки, содержащие фрагмент <code>user</code> , сразу после которого должен быть один любой пробельный символ, после которого идет фрагмент <code>24</code>	<code>user 24</code> <code>user</code> <code>24</code>
<code>user-\S007</code>	Подходят все строки, содержащие фрагмент <code>user-</code> , сразу после которого должен быть один любой не пробельный символ, после которого идет фрагмент <code>007</code>	<code>user-A007</code> <code>my user-1007</code>

Пример шаблона	Описание шаблона	Примеры строк, соответствующих шаблону
<code>\w\w\555</code>	Подходят все строки, содержащие фрагмент из двух любых буквенно-цифровых символов (или символа нижнего подчеркивания), сразу после которого идет фрагмент 555	aa555 12555 a_555
<code>Python\W</code>	Подходят все строки, содержащие фрагмент Python, сразу после которого идет один символ, не относящийся к буквенно-цифровым символам (и не символ нижнего подчеркивания)	Python! Python?
<code>[0-5] [0-7A-Ca-c]</code>	Подходят все строки, содержащие фрагмент из двух символов, первый из которых это цифра в диапазоне от 0 до 5, а второй — это либо цифра от 0 до 7, либо символ латинского алфавита от «A» до «C», или от «a» до «c»	1a 2B 4c
<code>([^B-Db-d])</code>	Подходят все строки, содержащие фрагмент из одного символа, заключенного в круглые скобки. Это должен быть символ латинского алфавита, кроме символов в диапазонах от «B» до «D» и от «b» до «d»	(1) (a) (A) (e)

При создании шаблона нам часто нужно не только сформировать конструкцию, которая описывает правило соответствия, но и указать количество символов, которые должны быть в анализируемой строке, чтобы соответствие было зафиксировано.

В регулярных выражениях для этого предназначены квантификаторы. Один из квантификаторов был рассмотрен выше — это число в фигурных скобках `{n}`. Квантификатор указывается после символа (или набора в `[...]`), определяя, сколько таких экземпляров нам нужно.

Таблица 8

Квантификатор	Задаваемое количество повторений
<code>{n}</code>	ровно n
<code>{m,n}</code>	от m до n (включительно)
<code>{m,}</code>	от m и больше (не меньше, чем m)
<code>{,n}</code>	до n (не больше, чем n)
<code>?</code>	ноль или одно
<code>*</code>	ноль или больше нуля (от нуля включая его)
<code>+</code>	одно или больше одного (как минимум одно)

Рассмотрим примеры использования квантификаторов в шаблонах

Таблица 9

Пример шаблона	Описание шаблона	Примеры строк, соответствующих шаблону
<code>student\d{5}</code>	Подходят все строки, содержащие фрагмент <code>student</code> , сразу после которого идет пять десятичных цифр	<code>student12345</code> <code>student80567</code>

Пример шаблона	Описание шаблона	Примеры строк, соответствующих шаблону
<code>student\d{3,5}</code>	Подходят все строки, содержащие фрагмент <code>student</code> , сразу после которого идут от трех до пяти десятичных цифр	<code>student12345</code> <code>student0000</code> <code>student805</code>
<code>user\d{3,}</code>	Подходят все строки, содержащие фрагмент <code>user</code> , сразу после которого идут десятичные цифры в количестве не менее трех	<code>user111</code> <code>user1111111111</code>
<code>user\d{,2}</code>	Подходят все строки, содержащие фрагмент <code>user</code> , сразу после которого идут десятичные цифры в количестве не более двух	<code>user1</code> <code>user99</code>
<code>come?</code>	Подходят все строки, содержащие фрагмент <code>come</code> , причем последний символ «е» может как присутствовать, так и нет	<code>come</code> <code>coming</code> <code>com.</code>
<code>user\d*</code>	Подходят все строки, содержащие фрагмент <code>user</code> , после которого может идти одна десятичная цифра (но ее наличие необязательно)	<code>user</code> <code>user1</code> <code>user2345</code>
<code>user\d+</code>	Подходят все строки, содержащие фрагмент <code>user</code> , после которого должна идти как минимум одна десятичная цифра	<code>user1</code> <code>user99</code> <code>user9999999999999999</code>

Как было сказано выше, работа с регулярными выражениями в Python предусматривает использование возможностей модуля `re`.

Наиболее распространенными и востребованными функциями модуля `re` для обнаружения совпадений являются:

- `re.search(pattern, strObj)` — ищет в строке `strObj` первое совпадение с шаблоном `pattern`
- `re.findall(pattern, strObj)` — ищет в строке `strObj` все (непересекающиеся) совпадения с шаблоном `pattern` (результат — список совпавших с шаблоном строк)
- `re.match(pattern, strObj)` — ищет в начале строки `strObj` совпадение с шаблоном `pattern`

Помимо функций поиска совпадений модуль `re` предоставляет нам еще такие полезные функции, как `re.sub()` для замены найденных совпадений на новый фрагмент и `re.split()` для разбиения строки по фрагментам, которые совпадают с шаблоном.

Некоторые из перечисленных функций возвращают объект `Match` (в случае обнаружения совпадения с шаблоном в анализируемой строке). Рассмотрим его особенности на примере.

Предположим, что в строке необходимо найти первое слово, состоящее только из буквенно-цифровых символов длиной четыре. Для это нам понадобится такой шаблон: `'\w{4}'`

```
import re

userStr="abcd abc efgh"
match = re.search(r'\w{4}', userStr)
```

Шаблон подаем как «сырую» строку для учета (при необходимости) всех символов (в том числе и непечатаемых)

В случае, если совпадение с шаблоном не найдено, то функция `search()` вернет `None`.

Иначе для дальнейшей работы с результатами поиска мы можем воспользоваться одним из методов объекта `Match`:

- `group()` — возвращает подстроку, которая совпала со всем шаблоном или с выражением в группе шаблона (будет рассмотрено далее).

Если нужно получить всю совпавшую с шаблоном подстроку, то метод вызывается без аргументов или с аргументом `0`:

```
print(match.group()) # abcd
print(match.group(0)) # abcd
```

Теперь попробуем найти первое вхождение в строку последовательности (фрагмента) длиной три и состоящего только из цифр.

Шаблон: `'\d{3}'`

```
import re

userStr="abcd abc 123 efgh 456"
match = re.search(r'\d{3}', userStr)
print(match.group()) # 123
```

А сейчас рассмотрим, как работать с группами в шаблонах.

Допустим, что нам необходимо проанализировать строку на наличие в ней номеров мобильных телефонов двух операторов (известен код оператора)

```
import re

userStr="My cell phone numbers: Vodafone +38(095)1234567;
        Cellcom +38(067)9875612";

match1 = re.search(r'Vodafone \+38\ (095\
                    (\d\d\d\d\d\d\d\d); Cellcom \+38\ (067\
                    (\d\d\d\d\d\d\d\d)', userStr)
```

Как видно в примере символы «+» , открывающих-ся «(» и закрывающихся «)» круглых скобок в номерах телефонов необходимо экранировать, т.к. они имеют специальное значение в регулярных выражениях. Так далее с помощью `()` мы создали две группы (для каждого из номера телефона), чтобы далее можно было работать с каждым из извлеченных номеров в отдельности.

```
print(match1.group()) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612

print(match1.group(0)) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612
```

Для получения подстрок, совпавших с определенной группой, нужно вызвать метод `group()` с ее номером в качестве аргумента:

```
print(match1.group(1)) # 1234567
print(match1.group(2)) # 9875612
```

В случае, если в какую-то группу ничего не попало, то будет пустая строка.

Если указать номер группы больше, чем общее количество групп, то интерпретатор выдаст ошибку «**no such group**».

Метод `group()` также может принимать несколько аргументов (можно задавать несколько номеров групп). В этом случае результат — кортеж, состоящий из подстрок, соответствующих совпадениям с группами.

```
print(match1.group(1,2)) # ('1234567', '9875612')
```

Также у объекта `Match` есть два полезных метода для получения индексов начала и конца совпадающего с шаблоном фрагмента: `start()` и `end()`. Данные методы по аналогии с методом `group()` могут работать без аргументов (для получения индексов фрагмента с полным совпадением) либо с номерами групп.

Так для нашего примера:

```
print(match1.start(), match1.end()) #23 73
print(match1.start(0), match1.end(0)) #23 73
print(match1.start(1), match1.end(1)) #40 47
print(match1.start(2), match1.end(2)) #66 73
```

У рассмотренных методов `start()` и `end()` есть аналог — метод `span()`, который возвращает кортеж, состоящий из индекса начала и индекса конца совпадающего с шаблоном фрагмента:

```
print(match1.span()) # (23, 73)
print(match1.span(0)) # (23, 73)
print(match1.span(1)) # (40, 47)
print(match1.span(2)) # (66, 73)
```

Одновременно с объектом `Match` мы рассмотрели пример работы с функцией `re.search(pattern, strObj)`, при вызове которой, если поиск в строке `strObj` первого совпадения с шаблоном `pattern` удачен, то результат — объект `Match`, иначе — `None`.

Данная функция полезна, если надо найти только одно совпадение в строке. Если же нам необходимо найти все совпадения, то можно использовать функцию `re.findall(pattern, strObj)`.

Рассмотрим на примере. Предположим, что нам нужно вывести все даты (в формате дд.мм.гггг) из текста с расписанием соревнований.

```
import re

userStr="2021-2022 Competition Calendar:30.11.2021 —
        2021 Grand Prix Series; 14.01.2022 —
        Grand Premio D'Italia"

match2=re.findall(r'\d{2}\.\d{2}\.\d{4}', userStr)

print(match2) # ['30.11.2021', '14.01.2022']
```

В зависимости от того, есть ли в шаблоне группы, результат, возвращаемый функцией `findall()` может быть:

- список строк, которые соответствуют шаблону (если в шаблоне нет групп, как в нашем примере);
- список строк, которые соответствуют фрагменту шаблона в группе (если в шаблоне только одна группа);
- список кортежей, состоящих из строк, которые соответствуют фрагментам шаблона в группах (если групп в шаблоне несколько).

В ситуации, когда нам необходимо проверить, что начало строки соответствует заданному шаблон, подойдет функция `re.match(pattern, strObj)`, которая аналогично функции `search()` возвращает объект `Match` при совпадении и `None` — иначе.

```
import re

userStr1="My cell phone numbers: Vodafone
        +38(095)1234567; Cellcom +38(067)9875612";
userStr2="Vodafone +38(095)1234567; Cellcom
        +38(067)9875612 — my cell phone numbers";

match3 = re.match(r'Vodafone \+38\.(095\.)
                  (\d\d\d\d\d\d\d\d); Cellcom \+38\.(067\.)
                  (\d\d\d\d\d\d\d\d)', userStr1)
match4 = re.match(r'Vodafone \+38\.(095\.)
                  (\d\d\d\d\d\d\d\d); Cellcom \+38\.(067\.)
                  (\d\d\d\d\d\d\d\d)', userStr2)

print(match3) #None
print(match4.group()) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612
```

Например, если применить ее для обнаружения рассмотренного выше паттерна «`Vodafone \+38\.(095\.)\d\d\d\d\d\d\d\d); Cellcom \+38\.(067\.)\d\d\d\d\d\d\d\d`» для проверки строки `userStr1` «`My cell phone numbers: Vodafone +38(095)1234567; Cellcom +38(067)9875612`», то результат будет `None`, т.к. строка не начинается с данного шаблона, а содержит его в середине, а результат проверки `userStr2` — успешен, получен объект `Match`.

Иногда в задачах обработки текстовой информации мы сталкиваемся с необходимостью замены отдельных

фрагментов контента (удовлетворяющих некоторому условию) на новую информацию.

Модуль `re` предоставляет нам функцию `re.sub(pattern, strRepl, strObj)`, которая заменяет найденные в строке `strObj` совпадения с шаблоном `pattern` на новый фрагмент `strRepl`.

Допустим, что нам необходимо заменить в нашем расписании соревнований символы минуса и точки с запятой на символ пробела.

```
import re

userStr="2021-2022 Competition Calendar:
        30.11.2021 — 2021 Grand Prix Series;
        14.12.2021 — Grand Pemio D'Italia"
newStr=re.sub(r'[-;]', '/',userStr)
print(newStr) #2021/2022 Competition
               Calendar:30.11.2021 /
               2021 Grand Prix Series/ 14.12.2021 /
               Grand Pemio D'Italia
```

Ранее при изучении методов объекта строка мы уже познакомились с методом `.split()`, который разбивает строку на части.

В модуль `re` включена функция `split`, которая работает аналогично, но позволяет выполнять разбиение на основании более сложных условий. Возвращаемый результат — набор (список) строк.

Например, строку «30.11.2021 — 2021 Grand Prix Series, 14.12.2021 — Grand Pemio D'Italia; 27.12.2021 — Cup of Austria by IceChallenge» надо разбить на отдельные строки (под каждое соревнование). Как мы видим, информация

о соревнованиях разделяется в одном случае символом запятой, а в другом — точкой с запятой.

```
import re

userStr="30.11.2021 — 2021 Grand Prix Series,  
        14.12.2021 — Grand Pemio D'Italia;  
        27.12.2021 — Cup of Austria by IceChallenge"

strList=re.split(r'[;,]+', userStr)

print(strList) #['30.11.2021 — 2021 Grand Prix Series',  
               #' 14.12.2021 — Grand Pemio D'Italia',  
               #' 27.12.2021 — Cup of Austria by IceChallenge']
```


СПИСКИ

Понятие классического массива

Очень часто для хранения информации о некотором объекте предметной области нам недостаточно одного значения.

Базовые типы данных (целочисленные, вещественные и строки) могут в один момент времени хранить только одно значение, соответственно, они не подходят в данной ситуации, т.к. при присваивании переменной нового значения предыдущее «затирается» и для его хранения нужна еще одна переменная.

А если возможных значений такой переменной не два, а намного больше? Например, нам нужно хранить оценки студентов группы, в группе 30 человек. Тогда нам нужно создать 30 отдельных переменных для хранения оценки каждого студента группы? А если групп несколько?

Здесь мы приходим к необходимости создания такой переменной (и такого типа данных), которая может хранить несколько значений. При этом логично перенумеровать их по порядку следования и получать к ним доступ далее по этому порядковому номеру.

Именно таким образом и организована классическая структура данных под названием массив, которая относится к составным типам данным.

В некоторых языках программирования (например, в C++) массив — это упорядоченный набор элементов

одинакового типа, которые последовательно расположены в памяти.

Адрес	n	$n+k$	$n+2k$	$n+k(q-1)$		
Значение	$a[0]$	$a[1]$	$a[2]$...	$a[q-1]$	
Индекс	0	1	2		$q-1$	

Рисунок 31

Представленный на этом рисунке массив a содержит q элементов (одного типа данных) с индексами (порядковыми номерами элементов в массиве) от 0 до $q-1$. Каждый элемент занимает в памяти компьютера одинаковое число (k) байт, при этом элементы размещены в памяти один за другим (непрерывная область памяти).

В Python классического массива, как отдельного типа данных, нет. Вместо этого нам предоставляется возможность использования различных коллекций, обладающих более разнообразной функциональностью по сравнению с классическими массивами.

Понятие коллекции объектов

Представим, что нам необходимо хранить и обрабатывать жанры-категории фильмов: комедия, приключения, вестерн, т.е. с одной переменной `category` будет связан целый набор значений: «Drama», «Comedy», «Fantasy», «Western» (в этом примере все значения одного типа данных — строки). Также возможны ситуации, когда нужно, чтобы у переменной было несколько значений разных

типов, например, список предметов может содержать, как названия предметов, так и их целочисленные идентификаторы («*Algorithms*», 2345, 7009, «*Networks*», «*Databases*»).

Именно для таких ситуаций нам необходимы коллекции. В Python под коллекцией понимается объект, который может хранить несколько (набор) значений. При этом эти значения могут быть как одного, так и разных типов данных (в том числе и другими объектами).

В зависимости от типа коллекции:

- могут быть изменяемыми или неизменяемым;
- обладать определенными методами;
- быть упорядоченными (индексированными) или неупорядоченными;
- состоять из набора только уникальных элементов или в них возможны повторения значений.

Рассмотрим перечисленные выше характеристики коллекций детальнее.

Изменяемость — над коллекцией допустимы операции добавления новых и удаления существующих значений.

Упорядоченность — каждый элемент коллекции характеризуется не только своим значением, но и индексом (порядковым номером элемента в коллекции). С понятиями индекса, индексацией и срезами мы уже познакомились ранее при работе со строками.

Уникальность — коллекция состоит только из уникальных (неповторяющихся) элементов.

Как было сказано выше у каждой коллекции есть набор собственных методов для работы с ее элементами или коллекцией в целом. Однако также есть набор операций,

которые допустимы и полезны при работе с любой коллекцией независимо от ее типа.

1. Определение длины коллекции (количества ее элементов) с помощью встроенной функции `len()`;
2. Проверка принадлежности некоторого элемента к коллекции с помощью оператора принадлежности `in`;
3. Обход коллекции с помощью цикла `for in`;
4. Вывод всех элементов коллекции с помощью встроенной функции `print()`.

Ссылочный тип данных `list`

Первый тип коллекций в Python, с которой мы познакомимся — это список. Практически в любой задаче независимо от предметной области мы сталкиваемся с понятием списка: список студентов, список ингредиентов рецепта, список книг, список фильмов и т.д.

В Python список (`list`) — это изменяемая, упорядоченная (т.е. индексируемая) коллекция значений любых (в том числе и разных) типов данных. При этом в одном списке значения могут повторяться (т.е. список не обладает свойством уникальности).

Список (`list`) является объектом, т.е. после создания переменной типа `list` она будет хранить адрес объекта класса список. По этому адресу в памяти резервируется область, в которой хранятся адреса (ссылки) на элементы списка в памяти (рис. 32).

Например, после создания переменной — списка `myList`, содержащей три значения (целое число `50`, строку «*student*», вещественное число `100.25`), в памяти по адресу `adr1` было

выделено место для хранения трех адресов (ссылок): **adr2**, **adr3**, **adr4**. По адресу **adr2** в памяти располагается значение первого элемента списка — целое число **50** и т.д. Таким образом, каждый элемент списка хранит адрес другого объекта (является ссылкой на другой объект).

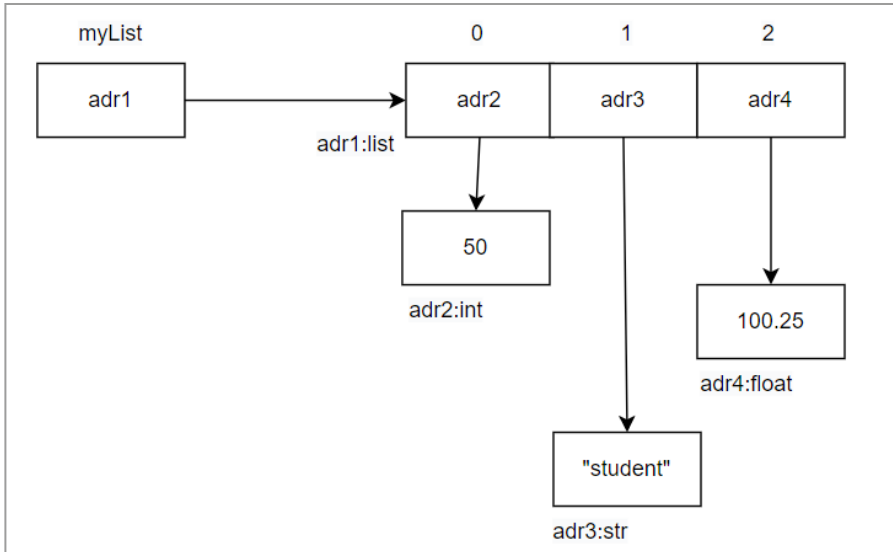


Рисунок 32

Создание списков

Для того, чтобы использовать в своих задачах функциональность списков, нам необходимо предварительно создать его (определить).

Определить списки в Python можно двумя способами.

1. Можно создать переменную и инициализировать ее набором элементов, заключенных в квадратные скобки:

```
category = ["Drama", "Comedy", "Fantasy"]
```

2. А можно воспользоваться встроенной функцией `list`, передав ей значения элементов в виде набора:

```
courses = list(("Math", "Algorithms", "Databases"))
```

Примечание: обращаем внимание на двойные открывающиеся и закрывающиеся круглые скобки! Функция `list` может вызываться или без аргументов (для создания пустого) или с одним. Поэтому для передачи нескольких значений мы заключаем набор в собственные круглые скобки.

В обоих случаях результат будет одинаков — будет создан объект-список.

Воспользуемся функцией `print()` для вывода элементов списка:

```
print(category) #['Drama', 'Comedy', 'Fantasy']  
print(courses) #['Math', 'Algorithms', 'Databases']
```

Передавая в качестве аргумента функции `print()` имя списка мы выводим все его значения (вместе с символами `[]`, знаками апострофа, т.к. элементы — это строки, и запятыми в виде разделителей элементов списка).

```
['Drama', 'Comedy', 'Fantasy']  
['Math', 'Algorithms', 'Databases']
```

Рисунок 33

Если нам необходимо создать пустой список (без элементов), например, когда перечень значений еще неизвестен, то это также можно реализовать двумя способами:

```
studentScores=[]
students=list()

print(studentScores) #[]
print(students) #[]
```

Попытка вывести пустой список приведет к отображению пустых квадратных скобок:



Рисунок 34

Теперь предположим, что в нашем списке должны быть элементы разных типов:

```
myCourses= ["Algorithms", 2345, 7009, "Networks",
            "Databases"]
print(myCourses) #['Algorithms', 2345, 7009,
                  'Networks', 'Databases']
```

Также элементом списка может быть другой список:

```
nestedList=[1,2.5,[45, "Example"]]
print(nestedList) #[1, 2.5, [45, 'Example']]
```

Элементы списка могут повторяться:

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
```

Также, если передать функции `list()` строку, то результатом ее работы будет список, состоящий из символов строки:

```
mySymbols=list("abcdef")
print(mySymbols) #['a', 'b', 'c', 'd', 'e', 'f']
```

Генераторы списков

Помимо рассмотренных выше подходов к созданию списков существует дополнительный способ — с помощью генераторов списков.

List comprehensions (списковые включения, генераторы списков) — это способ создания списка на основе генерируемых (по заданном правилу) значений.

Рассмотрим общий синтаксис такого генератора:

```
newList = [expression for item in sequence]
```

expression — выражение (вычисление), которое выполняется над каждым элементом **item** в последовательности **sequence**. Результат работы генератора — новый список **newList**.

Рассмотрим работу генератора на примерах. Допустим, что нам необходимо сгенерировать список из квадратов чисел в диапазоне от 0 до 5:

```
list1=[i*i for i in range(6)]
print(list1) #[0, 1, 4, 9, 16, 25]
```

В этом примере:

- **i*i** — это выражение, используемое для получения квадрата числа;
- **i** — каждый элемент из последовательности 0, 1, 2, 3, 4, 5 (**sequence**), которая была получена в результате работы функции **range(6)**.

В качестве последовательности может быть использована и строка.

Предположим, что у нас есть строка «*example*», из элементов которой нужно создать список, присоединяя к ним символ *.

```
list2=[i+"*" for i in "example"]
print(list2) #['e*', 'x*', 'a*', 'm*', 'p*', 'l*', 'e*']
```

Или, например, нужно создать список из продублированных 5 раз символов строки:

```
list3=[i*5 for i in "abctdf"]
print(list3) #['aaaaa', 'bbbbb', 'ccccc', 'ddddd',
              'ttttt', 'ffffff']
```

В генераторы списков можно добавлять условие, на основании которого выбираются элементы для обработки выражением. В этом случае синтаксис, следующий:

```
newList = [expression for item in sequence if condition]
```

expression — выражение (вычисление), которое выполняется над таким элементом **item** в последовательности **sequence**, для которого **condition == True**.

Условие похоже на некоторый фильтр, который отбирает для обработки и помещения в новый список только те элементы, для которых это условие выполняется.

Допустим, что нам необходимо сгенерировать список из квадратов четных чисел в диапазоне от 1 до 10:

```
list4=[i*i for i in range(1,11) if i%2==0]
print(list4) #[4, 16, 36, 64, 100]
```

Или выбрать из уже имеющегося списка всех клиентов, кроме *Bob* и *Joe*:

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']  
  
list5=[i for i in customers if i!='Bob' and i!='Joe']  
  
print(list5) #['Anna', 'Nick']
```

В Python также предусмотрена возможность создания генератора с несколькими циклами.

```
list6 = [x*y for x in range(1, 4) for y in range(1, 4)]  
  
print(list6) #[1, 2, 3, 2, 4, 6, 3, 6, 9]
```

В данном примере таблица умножения на три представляется в виде списка. Здесь цикл `for x in range(1, 4)` является вложенным в цикл `for y in range(1, 4)`, а действие, создающее элемент нового списка, `x*y` — располагается во внутреннем цикле, т.е. повторяется 9 раз.

Генераторы списков также полезны, если необходимо создать вложенный список. Например, чтобы представить ту же таблицу умножения на три в виде матрицы:

```
list7 = [[x*y for x in range(1, 4)] for y in range(1, 4)]  
  
print(list7) #[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Цикл, наполняющий элементами, каждую строку нового списка помещен вместе с выражением в собственные квадратные скобки.

Работа со списками

Ранее нами уже было отмечена такая особенность списков, как упорядоченность или индексированность, т.е. к отдельным элементам списка мы можем получать доступ по его индексу.

Аналогично строкам и другим упорядоченным коллекциям, индексация элементов списка начинается с нуля и возможно обращение к элементам с помощью отрицательных индексов.

Допустим, что у нас есть список:

```
myList ["user", 12, 200.34, False, True]
```

Таблица 10

0	1	2	3	4
"user"	12	200.34	False	True
-5	-4	-3	-2	-1

Первый элемент списка со значением “user” мы можем получить по индексу 0 или -5 (как и в строках, отрицательный индекс последнего элемента списка -1, а индекс первого — отрицательное значение длины списка):

```
myList[0], myList[-5].
```

Рассмотрим на примерах:

```
myList = ["user", 12, 200.34, False, True]
print(myList [1]) #12
print(myList [-1]) #True
print(myList [len(L)-1]) #True
print(myList [-2]) #False
```

Попытка обращения к элементу по несуществующему индексу (например, `myList[5]` или `myList[-7]`) вызовет ошибку: *IndexError: list index out of range*.

Также нам доступна возможность использовать срезы (способ получения непрерывного фрагмента списка за одно действие). Синтаксис срезов списка аналогичен рассмотренным ранее срезам строк. Мы можем формировать как базовые срезы с указанием границ среза (обоих или одной из), так и расширенные срезы с указанием шага индексирования.

Рассмотрим работу с различными вариантами срезов списка на примерах.

```
myCourses= ["Algorithms", 2345, 7009, "Networks",
            "Databases"]
print(myCourses[1:3]) #[2345, 7009]
print(myCourses[-4:-2]) #[2345, 7009]
print(myCourses[1:-1]) #[2345, 7009, 'Networks']
print(myCourses[:-1]) #['Algorithms', 2345, 7009,
                        'Networks']
print(myCourses[3:]) #['Networks', 'Databases']
print(myCourses[::2]) #['Algorithms', 7009,
                        'Databases']
print(myCourses[::-1]) #['Databases', 'Networks',
                        7009, 2345, 'Algorithms']
print(myCourses[-4::-1]) #[2345, 'Algorithms']
```

В отличие от строк списки — это изменяемая коллекция, т.е. с использованием индекса мы можем изменять значение элемента:

```
category =["Drama", "Comedy", "Fantasy"]
print(category) #['Drama', 'Comedy', 'Fantasy']
```

```
category[-1]="Action"
print(category) #['Drama', 'Comedy', 'Action']
```

Возможно также и изменение значений целых фрагментов списка, используя срезы. Например, необходимо заменить логины через одного пользователя (т.е. у пользователей «admin», «teacher», «user») на логины «newUser1», «newUser2» и т.д.:

```
userLogs=["admin","student","teacher","HR","user"]
print(userLogs) #['admin', 'student', 'teacher',
                 'HR', 'user']

userLogs[::2]=["newUser1","newUser2","newUser3"]
print(userLogs) #['newUser1', 'student', 'newUser2',
                 'HR', 'newUser3']
```

Примечание! *Количество новых значений должно быть равно длине среза (в нашем примере в результате среза мы получим 3 элемента, соответственно список новых значений должен содержать тоже 3 элемента).*

Для работы со списками можно использовать следующие встроенные Python-функции:

- `len(listObj)` — возвращает длину списка `listObj` (количество элементов в списке);
- `max(listObj)` — возвращает максимальный элемент в списке `listObj`;
- `min(listObj)` — возвращает минимальный элемент в списке `listObj`;

- `sum(listObj)` — возвращает сумму значений в списке `listObj`;
- `sorted(listObj)` — возвращает копию списка `listObj`, в котором элементы упорядочены по возрастанию (в случае числовых значений) или по алфавиту. Не изменяет оригинальный список `listObj`.

Функции `max(listObj)`, `min(listObj)`, `sum(listObj)` применяются для списков, содержащих числовые значения.

Рассмотрим применение перечисленных функций на примерах:

```
userLogs = ["admin", "student", "teacher", "hr", "user"]
print(len(userLogs)) #5
print(sorted(userLogs)) #['admin', 'hr', 'student',
                        'teacher', 'user']

prices = [100, 250.45, 1200, 20.78]
print(sum(prices)) #1571.23
print(max(prices)) #1200
print(min(prices)) #20.78
print(sorted(prices)) #[20.78, 100, 250.45, 1200]
```

Операции конкатенации (символ «+») и дублирования (символ «*») со списками работают также, как и со строками:

```
category1 = ["Drama", "Comedy"]
category2 = ["Action", "Fantasy"]

print(category1+category2) #['Drama', 'Comedy',
                          'Action', 'Fantasy']
print(category1*2) #['Drama', 'Comedy', 'Drama',
                  'Comedy']
```

Методы списков

Так как список — это коллекция, то обработку его элементов можно выполнять в циклах, используя конструкцию `for ... in`, как и со строками.

Например, выведем значение каждого элемента списка:

```
category = ["Drama", "Comedy", "Mystery", "Romance"]  
  
for film in category:  
    print(film)
```

Здесь на каждом шаге цикла в переменную `film` попадает элемент списка `category`.

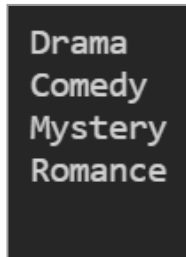


Рисунок 35

Также можно организовать цикл с использованием функции `range()`, используя длину цикла:

```
category = ["Drama", "Comedy", "Mystery", "Romance"]  
  
for i in range(len(category)):  
    print(category[i])
```

В этом цикле значения переменной `i` пройдут в диапазоне от `0` до `len(category)-1`, т.е.: `0, 1, 2, 3`.

Результат будет аналогичен предыдущему:

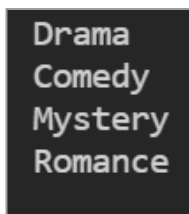


Рисунок 36

Однако, большая часть операций, проводимых над элементами списков осуществляется с помощью методов объекта список. Рассмотрим наиболее популярные и полезные из них.

Вначале изучим методы для расширения списка, т.е., как выполнять добавление, вставку элементов и т.д.

Метод `listObj.append(item)` позволяет добавить еще один элемент (аргумент метода, `item`) в конец списка `listObj`.

```
category1=["Drama", "Comedy"]
print(category1) #['Drama', 'Comedy']
category1.append("Fantasy")
print(category1) #['Drama', 'Comedy', 'Fantasy']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Comedy', 'Fantasy']
```

Рисунок 37

Метод `listObj.append(iterableItem)` похож на метод `append()` в том, что он позволяет нам добавлять в список

`listObj`, однако его отличие в том, что мы можем добавить сразу несколько элементов, в том числе и из другого списка.

```
category1=["Drama", "Comedy"]
category2=['Action', 'Fantasy']
print(category1) #['Drama', 'Comedy']
category1.extend(category2)
print(category1) #['Drama', 'Comedy', 'Action',
                  'Fantasy']
category1.extend(["Mystery", "Romance"])
print(category1) #['Drama', 'Comedy', 'Action',
                  'Fantasy', 'Mystery', 'Romance']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Comedy', 'Action', 'Fantasy']
['Drama', 'Comedy', 'Action', 'Fantasy', 'Mystery', 'Romance']
```

Рисунок 38

Метод `listObj.insert (itemIndex, item)` вставляет указанный элемент `item` в список `listObj` по указанному индексу `itemIndex`.

```
category1=["Drama", "Comedy"]
print(category1) #['Drama', 'Comedy']
category1.insert(1,"Fantasy")
print(category1) #['Drama', 'Fantasy', 'Comedy']
category1.insert(2,"Action")
print(category1) #['Drama', 'Fantasy', 'Action',
                  'Comedy']
category1.insert(-1,"Romance")
print(category1) #['Drama', 'Fantasy', 'Action',
                  'Romance', 'Comedy']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Fantasy', 'Comedy']
['Drama', 'Fantasy', 'Action', 'Comedy']
['Drama', 'Fantasy', 'Action', 'Romance', 'Comedy']
```

Рисунок 39

Если в качестве `itemIndex` указать `0`, то элемент будет вставлен в начало списка. Если указанный `itemIndex` находится за пределами списка, то элемент будет добавлен в конец списка.

Теперь рассмотрим методы для удаления элементов списка.

- Метод `listObj.remove(item)` удаляет первое вхождение указанного значения (`item`) в списке.
- Метод `listObj.pop(itemIndex)` удаляет элемент по указанному (`itemIndex`) значению индекса. При использовании данного метода без аргументов будет удален последний элемент в списке `listObj`.

```
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]
print(category) #['Drama', 'Comedy', 'Mystery',
                 'Romance', 'Comedy']

category.remove("Comedy")
print(category) #['Drama', 'Mystery', 'Romance',
                 'Comedy']

category.pop(2)
print(category) #['Drama', 'Mystery', 'Comedy']

category.pop()
print(category) #['Drama', 'Mystery']
```

Результат:

```
['Drama', 'Comedy', 'Mystery', 'Romance', 'Comedy']
['Drama', 'Mystery', 'Romance', 'Comedy']
['Drama', 'Mystery', 'Comedy']
['Drama', 'Mystery']
```

Рисунок 40

Метод `listObj.clear()` удаляет все элементы из списка `listObj`.

```
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]
print(category) #['Drama', 'Comedy', 'Mystery',
                 'Romance', 'Comedy']
category.clear()
print(category) #[]
```

Результат:

```
['Drama', 'Comedy', 'Mystery', 'Romance', 'Comedy']
[]
```

Рисунок 41

Если необходимо определить позицию элемента в списке `listObj` по его значению (`item`), то используется метод `listObj.index(item)`.

```
category =["Drama", "Comedy", "Mystery", "Romance",
           "Comedy"]
print(category.index("Mystery")) #2
print(category.index("Comedy")) #1
```

Примечание: метод `index()` возвращает только первое вхождение элемента.

Если необходимо определить, сколько раз определенное значение `item` встречается в списке `listObj`, то используем метод `listObj.count(item)`.

```
category = ["Drama", "Comedy",
            "Mystery", "Romance", "Comedy"]
print(category.count("Comedy")) #2
```

Ранее нами уже была рассмотрена встроенная функция `sorted()`, которая возвращала отсортированную копию списка.

Также есть метод списка `listObj.sort(reverse=False)` с аналогичной функциональностью, который по умолчанию сортирует список по возрастанию (т.к у параметра `reverse` значение по умолчанию `False`). Если необходимо изменить направление сортировки (сортировать по убыванию), то следует установить `reverse=True`.

Метод `listObj.reverse()` меняет порядок сортировки элементов на обратный.

```
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]

category.sort()
print(category) #['Comedy', 'Comedy', 'Drama',
                'Mystery', 'Romance']

category.sort(reverse=True)
print(category) #['Romance', 'Mystery', 'Drama',
                'Comedy', 'Comedy']

prices=[100, 250.45, 1200, 20.78]
prices.sort()
```

```
print(prices) #[20.78, 100, 250.45, 1200]
prices.sort(reverse=True)
print(prices) #[1200, 250.45, 100, 20.78]
prices.reverse()
print(prices) #[20.78, 100, 250.45, 1200]
```

Результат:

```
['Comedy', 'Comedy', 'Drama', 'Mystery', 'Romance']
['Romance', 'Mystery', 'Drama', 'Comedy', 'Comedy']
[20.78, 100, 250.45, 1200]
[1200, 250.45, 100, 20.78]
[20.78, 100, 250.45, 1200]
```

Рисунок 42

Оператор принадлежности `in`

Достаточно часто при работе со списками встречается задача о проверке принадлежности определенного элемента некоторому списку. Например, входит ли в состав клиентов, которые хранятся в списке `customers = ['Bob', 'Anna', 'Joe', 'Bob', 'Nick']`, клиент с именем `'Bob'`?

Для таких ситуаций в Python есть возможность проверить наличие элемента в списке с помощью оператора `in`.

Рассмотрим на примере.

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
print('Bob' in customers) #True
```

Одно и тоже значение может находиться в списке более одного раза (как клиент `'Bob'` в нашем примере). До тех пор, пока оно будет в списке хотя бы в одном экземпляре, оператор `in` будет возвращать значение `True`.

Данный оператор удобен для формирования условий:

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
if ('Bob' in customers):
    print("Bob is our customer")
else:
    print("Sorry")
```

Особенности списков, ссылки и клонирование

Одной из наиболее важных особенностей списков является создание псевдонимов при операции присваивания списка другой переменной.

Псевдонимы — это переменные, которые имеют разные имена, но содержат одинаковые адреса памяти.

Данная особенность важна и ее необходимо учитывать, т.к. можно случайно, работая с одной переменной, испортить значения, хранящиеся в другой. Рассмотрим эти моменты детальнее на следующем примере.

```
list1=[1,2,3,4,5]
print(list1)  #[1, 2, 3, 4, 5]
list2=list1
print(list2)  #[1, 2, 3, 4, 5]
list2[1]="Hello"
print(list2)  #[1, 'Hello', 3, 4, 5]
print(list1)  #[1, 'Hello', 3, 4, 5]
```

Вначале у нас есть список `list1=[1, 2, 3, 4, 5]`. Далее мы создаем новый список `list2` и присваиваем ему список `list1`. После этой операции переменная `list2` содержит тот же адрес в памяти, что и переменная `list1`, т.е. фактически ссылается на тот же самый список.

Поэтому, когда мы, используя переменную `list2`, изменили второй элемент списка на слово «Hello», то при выводе списка как через переменную `list2`, так и через переменную `list1`, мы видим обновлённый список.

Если необходимо проверить, ссылаются ли две разные переменные на один и тот же список, то можно использовать следующий подход с помощью оператора `is`, который проверяет, являются ли две переменные одним и тем же объектом:

```
list1=[1,2,3,4,5]
list2=list1
list3=[6,7,8]

print(list2 is list1) #True
print(list3 is list1) #False
```

Если же нам необходимо скопировать элементы из существующего списка в новый (т.е. создать новый объект с такими же значениями), то можно использовать один из следующих способов:

- использовать функцию `copy()`;
- использовать функцию-конструктор списка `list()`;
- использовать срез всего списка `[:]`.

```
list1=[1,2,3,4,5]
print(list1) #[1, 2, 3, 4, 5]

list2=list1.copy()
list2[1]="Hello"
print(list2) #[1, 'Hello', 3, 4, 5]
print(list1) #[1, 2, 3, 4, 5]

list3=list(list1)
```

```
list3[2]="Hello"
print(list3) #[1, 2, 'Hello', 4, 5]
print(list1) #[1, 2, 3, 4, 5]

list4=list1[:]
list4[3]="Hello"
print(list4) #[1, 2, 3, 'Hello', 5]
print(list1) #[1, 2, 3, 4, 5]
```

В нашем примере мы создали копии списка `list1` тремя перечисленными способами, т.е. `list2`, `list3`, `list4` это новые объекты, которые имеют свои собственные значения, не связанные со значениями списка `list1`. Поэтому после выполнения изменений в этих трех новых списках содержимое списка `list1` осталось неизменным.

Точно также, если мы внесем изменения в список `list1`, то это не повлияет на содержимое списков `list2`, `list3`, `list4`:

```
list1[-1]=0
print(list1) #[1, 2, 3, 4, 0]
print(list2) #[1, 'Hello', 3, 4, 5]
print(list3) #[1, 2, 'Hello', 4, 5]
print(list4) #[1, 2, 3, 'Hello', 5]
```

Поиск элемента

Ранее нами был рассмотрен оператор `in`, который можно использовать для того, чтобы узнать, есть ли нужный нам элемент в списке. Результат такой проверки `True` или `False`. А что, если нам нужно не просто проверить наличие элемента, а узнать, где именно он находится (его позицию в списке)?

Именно для таких ситуаций и предназначен метод `listObj.index(item)` который возвращает позицию элемента `item` при его первом появлении в списке `ListObj`.

```
customers=['Bob','Anna','Joe','Nick']
print(customers.index('Joe')) #2

category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]
print(category.index('Comedy')) #1
```

Если же нам необходимо выбрать все элементы, равные указанному, то поиск можно организовать вручную, перебирая список.

```
customers=['Bob','Anna','Joe','Bob','Nick']

for i in range(len(customers)):
    if customers[i]=='Bob':
        print(i)

#0
#3
```

Матрицы

Как мы уже знаем, элементами списка могут быть любые типы данных, в том числе и другие списки, т.е. внутри списка могут находиться списки, которые называются вложенными. Подобные структуры называют матрицы.

Матрицы полезны для хранения данных, которые традиционно представляются в виде таблиц. Например, таблица с успеваемостью группы студентов (первый

столбец — студент, а далее — его оценки по предметам, каждая строка соответствует одному студенту):

Таблица 11

Bob	11	8	10	12	12
Jane	12	11	11	11	12
Kate	7	8	9	9	10

Рассмотрим особенности работы с матрицами подробнее. Например, для хранения данных такой таблицы

Таблица 12

111	112	113
221	222	223

нам понадобится такой список:

```
myTbl=[ [111,112,113], [221,222,223] ]
```

Таким образом, каждый элемент такого списка имеет два индекса: порядковый номер вложенного списка, в который он входит (номер «строки»), и его порядковый номер внутри этого списка (номер «столбца»). Как нам уже известно, индексация в Python начинается с нуля.

Таблица 13

111	112	113
[0][0]	[0][1]	[0][2]
221	222	223
[1][0]	[1][1]	[1][2]

```
myTbl=[ [111,112,113], [221,222,223] ]
print(myTbl[1][1]) #222
print(myTbl[0]) # [111, 112, 113]
```

Для перебора всех элементов таких списков можно использовать вложенные циклы:

```
for i in range(2):
    for j in range(3):
        print(myTbl[i][j])
#111
#112
#113
#221
#222
#223
```

ИЛИ

```
for i in range(len(myTbl)):
    for j in range(len(myTbl[i])):
        print(myTbl[i][j])
```

Для создания матриц также можно использовать генераторы списков:

```
myTbl2 = [[j for j in range(2)] for i in range(3)]
print(myTbl2) #[[0, 1], [0, 1], [0, 1]]
```

Здесь цикл по столбцам `for j in range(2)` является вложенным в цикл по строкам `for i in range(3)`, а действие, создающее новый элемент списка `j` — располагается во внутреннем цикле, т.е. повторяется 6 раз. Вложенный цикл помещается в собственные квадратные скобки.

Рассмотрим примеры работы с матрицами.

Предположим, что у нас есть данные об успеваемости 3 студентов в группе по 4 предметами. Для каждого студента нужно найти максимальную оценку.

Таблица 14

Bob	11	8	10	12
Jane	12	11	11	11
Kate	7	8	9	9

Как мы видим, каждому студенту соответствует одна строка таблицы, а данные о его оценках находятся в столбцах (начиная с первого и до конца, нулевой столбец хранит имя студента). Создадим такую матрицу и выведем ее построчно (каждый студент с новой строки):

```
studScores = [['Bob', 11, 8, 10, 12],
               ['Jane', 12, 11, 11, 11], ['Kate', 7, 8, 9, 9]]
for student in studScores:
    print(student)
```

```
['Bob', 11, 8, 10, 12]
['Jane', 12, 11, 11, 11]
['Kate', 7, 8, 9, 9]
```

Рисунок 43

Теперь организуем аналогичный цикл по строкам (студентам) и в каждой строке, начиная с первого столбца (с помощью среза), найдем максимальную оценку, используя встроенную функцию `max()`:

```
for student in studScores:
    print(student[0], max(student[1:]))
```

```
Bob 12
Jane 12
Kate 9
```

Рисунок 44

Примечание: функция `max()` находит первый максимальный элемент.

Рассмотрим еще один пример. Допустим, что у нас есть данные, к какой категории относится определенный фильм.

```
films=[['Catch Me If You Can', 'Biography', 'Crime', 'Drama'],
        ['Mrs. Doubtfire', 'Comedy', 'Drama', 'Family']]
```

Пользователь вводит название категории, которая его интересует, и программа выводит названия фильмов, относящихся к этой категории:

```
userCategory=input("Input film category: ")

for film in films:
    if userCategory in film[1:]:
        print(film[0])
```

Так как в нашем списке только один фильм относится к категории **Crime**, то результат запроса такой:

```
Input film category: Crime
Catch Me If You Can
```

Рисунок 45

А на запрос категории **Drama** у нас уже есть два фильма:

```
Input film category: Drama
Catch Me If You Can
Mrs. Doubtfire
```

Рисунок 46



Урок 3

Строки, списки

© STEP IT Academy, www.itstep.org

© Анна Егошина

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.