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

Последовательности в Python
Последовательность(Sequence Type) — итерируемый контейнер, к элементам которого есть эффективный доступ с использованием целочисленных индексов.

Последовательности могут быть как изменяемыми, так и неизменяемыми. Размерность и состав созданной однажды неизменяемой последовательности не может меняться, вместо этого обычно создаётся новая последовательность.

Примеры последовательностей в стандартной библиотеке Python:

1. Список (list) - изменяемая
2. Кортеж (tuple) - неизменяемая
3. Диапазон (range) - неизменяемая
4. Строка (str, unicode) - неизменяемая

Максимальная длина строки в Python

Максимальная длина строки зависит от платформы. Обычно это:

- 2**31 - 1 – для 32-битной платформы;
- 2**63 - 1 – для 64-битной платформы;

Константа `maxsize` , определенная в модуле `sys` :

```
>>> import sys

>>> sys.maxsize

2147483647
```

Создание строки в Python

Литералы строк

Литерал — способ создания объектов, в случае строк Питон предлагает несколько основных вариантов. Мы можем создать строку, заключив символы в одинарные или двойные кавычки. Python также предоставляет тройные кавычки для представления строки, но обычно используется для многострочных строк или строк документации.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)
#Using triple quotes
str3 = '''Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(str3)
```

Если внутри строки необходимо расположить двойные кавычки, и сама строка была создана с помощью двойных кавычек, можно сделать следующее:

```
>>> 'book "war and peace"'  # разный тип кавычек
'book "war and peace"'
>>> "book 'war and peace'"  # разный тип кавычек
"book 'war and peace'"
>>> "book \"war and peace\""  # экранирование кавычек одного типа
'book "war and peace"'
>>> 'book \'war and peace\''  # экранирование кавычек одного типа
"book 'war and peace'"
```

Разницы между строками с одинарными и двойными кавычками нет — это одно и то же

Какие кавычки использовать — решать вам, соглашение PEP 8 не дает рекомендаций по использованию кавычек. Просто выберите один тип кавычек и придерживайтесь его. Однако если в стоке используются те же

кавычки, что и в литерале строки, используйте разные типы кавычек — обратная косая черта в строке ухудшает читаемость кода.

PEP 8 — документ, описывающий соглашение о том, как писать код на языке Python. PEP 8 создан на основе рекомендаций создателя языка Гвидо ван Россума. Ключевая идея Гвидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. В идеале, если весь код будет написан в едином стиле, то любой сможет легко его прочесть.

https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html

Создание строки с помощью метода **str()**.

Как это работает:

```
my_num = 12345
my_str = str(my_num)
```

В данном случае мы создали новую строку путем конвертации переменной другого типа(например, **int**).

Экранирование строк

Экранированные последовательности - это служебные наборы символов, которые позволяют вставить нестандартные символы, которые сложно ввести с клавиатуры.

В таблице перечислены самые часто используемые экранированные последовательности:

\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция

\v	Вертикальная табуляция
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\x…	16-ричное значение
\o…	8-ричное значение

Теперь посмотрим, как каждая из них работает:

```
# Обычная строка
>>> str = 'Моя строка вот такая'
>>> print(str)
Моя строка вот такая
# Добавим символ переноса строки
>>> str = 'Моя строка\n вот такая'
>>> print(str)
Моя строка
 вот такая
# А теперь добавим возврат каретки
>>> str = 'Моя строка\n вот\r такая'
>>> print(str)
Моя строка
такая
# Горизонтальная табуляция(добавит отступ)
>>> str = '\tМоя строка вот такая'
>>> print(str)
    Моя строка вот такая
# Вертикальная табуляция(добавит пустую строку)
>>> str = '\vМоя строка вот такая'
>>> print(str)

Моя строка вот такая
# Добавим китайский иероглиф в строку
>>> str = 'Моя строка \u45b2 вот такая'
```

```
>>> print(str)
Моя строка 蜘 вот такая
```

"Сырые строки"

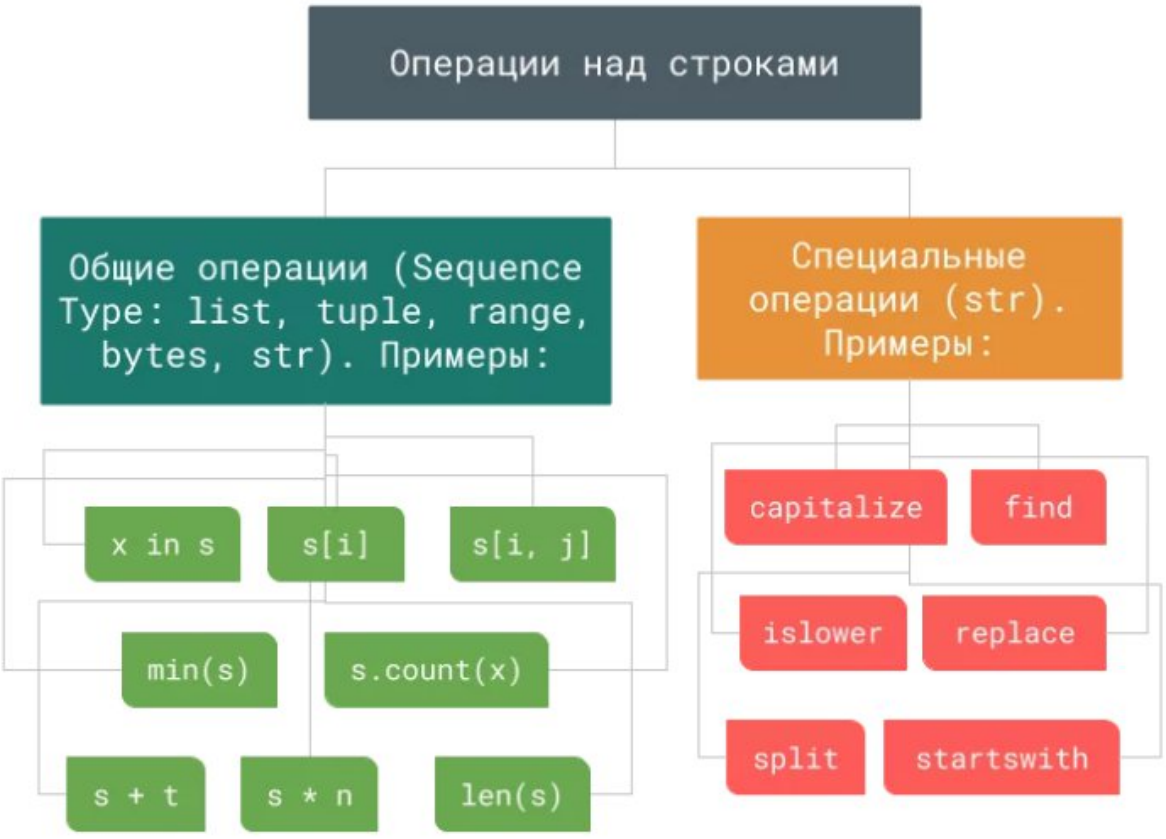
Если перед открывающей кавычкой стоит символ **'r'** (в любом регистре), то механизм экранирования отключается.

Это может быть нужно, например, в такой ситуации:

str = r'C:\new_file.txt'

Методы (функции) для работы со строками

Методов для работы со строками довольно много. Может возникнуть вопрос - а как же не запутаться в их многообразии? Ответ на него такой - необходимо структурировать и разбить методы по группам.



Работа со строками в Python. Классификация методов.

Все эти типы данных(базовые и дополнительные) поддерживают общую группу операций:

x in s	Если элемент присутствует в последовательности, то возвращает True, иначе - False
x not in s	Возвращает True, если элемент отсутствует в последовательности.
s + t	Конкатенация(сложение) двух последовательностей
s * n	Эквивалентно сложению последовательности s с собой n раз
s[i]	Возвращает i-й элемент последовательности
s[i, j]	Возвращает набор элементов последовательности с индексами из диапазона i <= k < j
min(s)	Минимальный элемент последовательности
max(s)	Максимальный элемент последовательности
len(s)	Длина последовательности
s.index(x)	Возвращает индекс подстроки x в строке s
s.count(x)	Число вхождений подстроки x в строку s

Оператор принадлежности подстроки in

Python также предоставляет оператор принадлежности, который можно использоваться для манипуляций со строками. Оператор in возвращает True, если подстрока входит в строку, и False, если нет:

```
>>> s = 'Python'
>>> s in 'I love Python.'
True
>>> s in 'I love Java.
'False
```

Есть также оператор not in, у которого обратная логика:

```
>>> 'z' not in 'abc'
True
>>> 'z' not in 'xyz'
False
```

Оператор сложения строк +

+ — оператор конкатенации строк. Он возвращает строку, состоящую из совокупности других строк.
Например:

```
>>> a = 'Вот так работает'
>>> b = ' конкатенация строк'
>>> a + b
'Вот так работает конкатенация строк'
```

Оператор умножения строк *

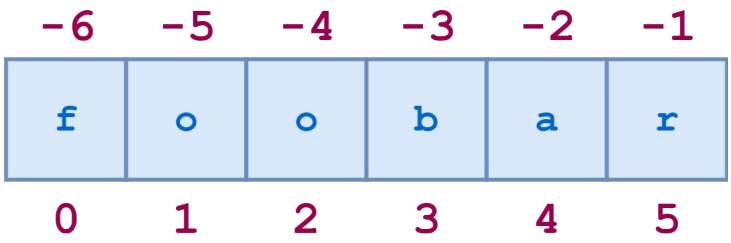
***** — оператор создает несколько копий строки. Если **str** это строка, а **n** целое число, то будет создано **n** копий строки **str**.

```
>>> str = 'Строка'
>>> 5 * str
```

```
'СтрокаСтрокаСтрокаСтрокаСтрока'
```

Индексация строк в Python

Для обращения к определенному символу строки используют индекс – порядковый номер элемента. Python поддерживает два типа индексации – **положительную**, при которой отсчет элементов начинается с **0** и **с начала** строки, и **отрицательную**, при которой отсчет начинается с **-1** и **с конца**



Вот несколько примеров отрицательного индексирования:

```
>>> s = 'foobar'

>>> s[-1]

'r'

>>> s[-2]

'a'

>>> len(s)

6
```

Срезы строк в Python

Индексы позволяют работать с **отдельными** элементами строк. Для работы с **подстроками** используют **срезы**, в которых задается нужный диапазон:

my_collection[start:stop:step] *# старт, стоп и шаг*

Особенности среза:

- Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.
- Отрицательное значение шага — перебор ведём в обратном порядке справа налево.
- Если не указан старт **[:stop:step]**— начинаем с самого края коллекции, то есть с первого элемента (включая его), если шаг положительный или с последнего (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
- Если не указан стоп **[start:: step]** — идем до самого края коллекции, то есть до последнего элемента (включая его), если шаг положительный или до первого элемента (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
- step = 1, то есть последовательный перебор слева направо указывать не обязательно — это значение шага по умолчанию. В таком случае достаточно указать **[start:stop]**
- Можно сделать даже так **[:]** — это значит взять коллекцию целиком
- **ВАЖНО**: ***При срезе, первый индекс входит в выборку, а второй нет!*** То есть от старта включительно, до стопа, где стоп не включается в результат. Математически это можно было бы записать как [start, stop) или пояснить вот таким правилом:
  [<первый включаемый> : <первый НЕ включаемый> : <шаг>]
  Поэтому, например, mylist[::-1] не идентично mylist[:0:-1], так как в первом случае мы включим все элементы, а во втором дойдем до 0 индекса, но не включим его!

Примеры срезов в виде таблицы:

Последовательность	a	b	c	d	e	f	g	**Результат слайсинга**
Индексы	0 (-7)	1 (-6)	2 (-5)	3 (-4)	4 (-3)	5 (-2)	6 (-1)	
[:] (→)	+	+	+	+	+	+	+	abcdefg
[::-1] (←)	+	+	+	+	+	+	+	gfedcba
[::2] (→)	+		+		+		+	aceg
[1::2] (→)		+		+		+		bdf
[:1]	+							a
[-1:]							+	g
[3:4]				+				d
[-3:] (→)					+	+	+	efg
[-3:1:-1] (←)			+	+	+			edc
[2:5] (→)			+	+	+			cde

Срез с двумя параметрами. Т. е. **S[a:b]** возвращает подстроку, начиная с символа с индексом **a** до символа с индексом **b**, не включая его. Если опустить второй параметр (но поставить двоеточие), то срез берется до конца строки. Пример:

```
>>> str = 'Hello'
>>> str[0:4]
'Hell'
>>> str[0:5]
'Hello'
>>> str[1:3]
'el'
>>> str[1:]
'ello'
>>> str[0:]
'Hello'
```

Срез с тремя параметрами - S[a:b:d]. Третий параметр задает шаг(как в случае с функцией **range**), то есть будут взяты символы с индексами **a, a + d, a + 2 * d** и т. д. Например, при задании значения третьего параметра, равному **2**, в срез попадет каждый второй символ:

```
>>> str = 'Hello'
>>> str[0:5:1]
'Hello'
>>> str[::1]
'Hello'
>>> str[0:5:2]
'Hlo'
>>> str[::2]
'Hlo'
```

Изменение строк

Строки — один из типов данных, которые Python считает неизменяемыми, что означает невозможность их изменять. Как вы ниже

увидите, python дает возможность ~~изменять~~ (заменять и перезаписывать) строки.

Такой синтаксис приведет к ошибке TypeError:

```
>>> s = 'python'
>>> s[3] = 't'
Traceback (most recent call last):   File "<pyshell#40>", line 1, in <module>     s[3] = 't'
TypeError: 'str' object does not support item assignment
```

На самом деле нет особой необходимости изменять строки. Обычно вы можете легко сгенерировать копию исходной строки с необходимыми изменениями. Есть минимум 2 способа сделать это в python. Вот первый:

```
>>> s = s[:3] + 't' + s[4:
]>>> s
'pytton'
```

Есть встроенный метод string.replace(x, y):

```
>>> s = 'python'
>>> s = s.replace('h', 't')
>>> s 'pytton'
```

Удаление строки

Как мы знаем, строки неизменяемы. Мы не можем удалить символы из строки. Но мы можем удалить всю строку с помощью ключевого слова del.

```
str = "JAVATPOINT"
del str[1]
```

Вывод:

TypeError: 'str' object doesn't support item deletion

Теперь мы удаляем всю строку.

```
str1 = "JAVATPOINT"

del str1

print(str1)
```

Вывод:

```
NameError: name 'str1' is not defined
```

Функции для работы со строками

Для работы со строками в Питоне предусмотрены специальные функции. Рассмотрим их:
Преобразование числового или другого типа к строке:

- `str(n)` – преобразование числового или другого типа к строке;
- `len(s)` – длина строки;
- `chr(s)` – получение символа по его коду ASCII;
- `ord(s)` – получение кода ASCII по символу;

Методы для работы со строками

Метод — это функция, применяемая к объекту, в данном случае — к строке. Метод вызывается в виде Имя_объекта.Имя_метода(параметры).
Например, S.find("e") — это применение к строке S метода find с одним параметром "e".

Кроме функций, для работы со строками есть немало методов:

- `find(s, start, end)` – возвращает индекс первого вхождения подстроки в s или -1 при отсутствии. Поиск идет в границах от `start` до `end`;
- `rfind(s, start, end)` – аналогично, но возвращает индекс последнего вхождения;
- `replace(s, new)` – меняет последовательность символов `s` на новую подстроку `new`;
- `split(x)` – разбивает строку на подстроки при помощи выбранного разделителя x;
- `join(x)` – соединяет строки в одну при помощи выбранного разделителя x;
- `strip(s)` – убирает пробелы с обеих сторон;

- `lstrip(s), rstrip(s)` – убирает пробелы только слева или справа;
- `lower()` – перевод всех символов в нижний регистр;
- `upper()` – перевод всех символов в верхний регистр;
- `capitalize()` – перевод первой буквы в верхний регистр, остальных – в нижний.

Форматирование строки

Оператор %

Строки в Python обладают встроенной операцией, к которой можно получить доступ оператором %, что дает возможность очень просто делать форматирование. Самый простой пример — когда для подстановки нужен только один аргумент, значением будет он сам:

```
>>> name = "Alex"

>>> 'Hello, %s' % name

'Hello, Alex'
```

Если же для подстановки используется несколько аргументов, то значением будет кортеж со строками:

```
>>> '%d %s, %d %s' % (6, 'bananas', 10, 'lemons')

'6 bananas, 10 lemons'
```

Как видно из предыдущего примера, зависимо от типа данных для подстановки и того, что требуется получить в итоге, пишется разный формат. Наиболее часто используются:

1. '**%d**', '**%i**', '**%u** — десятичное число;
2. '**%c**' — символ, точнее строка из одного символа или число – код символа;
3. '**%r**' — строка (литерал Python);
4. '**%s**' — строка.

Такой способ форматирования строк называет "старым" стилем, который в Python 3 был заменен на более удобные способы.

str.format()

В Python 3 появился более новый метод форматирования строк, который вскоре перенесли и в Python 2.7. Такой способ избавляет программиста от специального синтаксиса %-оператора. Делается все путем вызова `.format()` для строковой переменной. С помощью специального символа — фигурных скобок — указывается место для подстановки значения, каждая пара скобок указывает отдельное место для подстановки, значения могут быть разного типа:

```
>>> print('{}'.format(100))

100

>>> '{0}, {1}, {2}'.format('one', 'two', 'three')

'one, two, three'

>>> '{2}, {1}, {0}'.format('one', 'two', 'three')


'three, two, one'
```

f-строки (Python 3.6+)

В Python версии 3.6 был представлен новый способ форматирования строк. Эта функция официально названа литералом отформатированной строки, но обычно упоминается как f-string.

Возможности форматирования строк огромны и не будут подробно описана здесь.
Одной простой особенностью f-строк, которые вы можете начать использовать сразу, является интерполяция переменной. Вы можете указать имя переменной непосредственно в f-строковом литерале (f'string'), и python заменит имя соответствующим значением.

Например, предположим, что вы хотите отобразить результат арифметического вычисления. Это можно сделать с помощью простого print() и оператора ,, разделяющего числовые значения и строковые:

```
>>> n = 20

>>> m = 25

>>> prod = n * m

>>> print('Произведение', n, 'на', m, 'равно', prod)

Произведение 20 на 25 равно 500
```

Но это громоздко. Чтобы выполнить то же самое с помощью f-строки:

- Напишите f или F перед кавычками строки. Это укажет python, что это f-строка вместо стандартной.
- Укажите любые переменные для воспроизведения в фигурных скобках ({}).

Код с использованием f-string, приведенный ниже выглядит намного чище:

```
>>> n = 20

>>> m = 25

>>> prod = n * m

>>> print(f'Произведение {n} на {m} равно {prod}')

Произведение 20 на 25 равно 500
```

Любой из трех типов **кавычек в python** можно использовать для f-строки

Стандартная библиотека Template Strings

Еще один способ форматирования строк, который появился еще с выходом Python версии 2.4, но так и не стал популярным — использование библиотеки Template Strings. Есть поддержка передачи значения по имени, используется $-синтаксис как в языке PHP:

```
>>> from string import Template

>>> name = "Alex"

>>> age = 30
```

```
>>> s = Template('My name is $name. I'm $age.')

>>> print(s.substitute(name=name, age=age))


My name is Alex. I'm 30
```

Сравнение строк

При сравнении нескольких строк рассматриваются отдельные символы и их регистр:

- цифра условно меньше, чем любая буква из алфавита;
- алфавитная буква в верхнем регистре меньше, чем буква в нижнем регистре;
- чем раньше буква в алфавите, тем она меньше;

При этом сравниваются по очереди первые символы, затем — 2-е и так далее.

```
>>> s1 = "1a"
>>> s2 = "aa"
>>> s3 = "Aa"
>>> s4 = "ba"
>>> "1a" > "aa"  # сравнение цифры с буквой
False
>>> "aa" > "Aa"  # сравнение регистров
True
>>> "aa" > "ba"  # сравнение букв по алфавитному порядку
False
>>> "aa" < "az"  # первые буквы одинаковые, сравниваются следующие две
True
```

Далеко не всегда желательной является зависимость от регистра, в таком случае можно привести обе строки к одному и тому же регистру. Для этого используются функции `lower()` — для приведения к нижнему и `upper()` — к верхнему