Строковые операторы

Операторы `+` и `*` применяются как к численным значениям, так и могут применяться и к строкам.

Оператор сложения строк `+`

`+` — оператор конкатенации строк. Он возвращает строку, состоящую из других строк, как показано здесь:

```
>>> s = 'py'
>>> t = 'th'
>>> u = 'on'


>>> s + t
'pyth'
>>> s + t + u
'python'


>>> print('Привет, ' + 'Мир!')
Go team!!!
```

Оператор умножения строк `*`

`*` — оператор создает несколько копий строки. Если `s` это строка, а `n` целое число, любое из следующих выражений возвращает строку, состоящую из `n` объединенных копий `s`:

`s * n`
`n * s`

Вот примеры умножения строк:

```
>>> s = 'py.'


>>> s * 4
'py.py.py.py.'
>>> 4 * s

'py.py.py.py.'
```

Значение множителя `n` должно быть целым положительным числом. Оно может быть нулем или отрицательным, но этом случае результатом будет пустая строка:

```
>>> 'py' * -6

''
```

Если вы создадите строковую переменную и превратите ее в пустую строку, с помощью `'py' * -6`, кто-нибудь будет справедливо считать вас немного глупым. Но это сработает.

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
```

Встроенные функции строк в python

Python предоставляет множество функций, которые встроены в интерпретатор. Вот несколько, которые работают со строками:

Функция	**Описание**
chr()	Преобразует целое число в символ
ord()	Преобразует символ в целое число
len()	Возвращает длину строки
str()	Изменяет тип объекта на `string`

Более подробно о них ниже.

Функция `ord(c)` возвращает числовое значение для заданного символа.

На базовом уровне компьютеры хранят всю информацию в виде цифр. Для представления символьных данных используется схема перевода, которая содержит каждый символ с его репрезентативным номером.

Самая простая схема в повседневном использовании называется ASCII. Она охватывает латинские символы, с которыми мы чаще работает. Для этих символов `ord(c)` возвращает значение ASCII для символа `c`:

```
>>> ord('a')
97
>>> ord('#')
35
```

ASCII прекрасен, но есть много других языков в мире, которые часто встречаются. Полный набор символов, которые потенциально могут быть представлены в коде, намного больше обычных латинских букв, цифр и символом.

Unicode — это современный стандарт, который пытается предоставить числовой код для всех возможных символов, на всех возможных языках, на каждой возможной платформе. Python 3 поддерживает Unicode, в том числе позволяет использовать символы Unicode в строках.

Функция `ord()` также возвращает числовые значения для символов Юникода:

```
>>> ord('€')
8364
>>> ord('Σ')
```

```
8721
```

Функция `chr(n)` возвращает символьное значение для данного целого числа.

`chr()` действует обратно `ord()`. Если задано числовое значение `n`, `chr(n)` возвращает строку, представляющую символ `n`:

```
>>> chr(97)
'a'
>>> chr(35)
'#'
```

`chr()` также обрабатывает символы Юникода:

```
>>> chr(8364)
'€'
>>> chr(8721)
'∑'
```

Функция `len(s)` возвращает длину строки.

`len(s)` возвращает количество символов в строке `s`:

```
>>> s = 'Простоя строка.'
>>> len(s)
15
```

Функция `str(obj)` возвращает строковое представление объекта.

Практически любой объект в Python может быть представлен как строка. `str(obj)` возвращает строковое представление объекта `obj`:

```
>>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3 + 29)
'32'
>>> str('py')
'py'
```

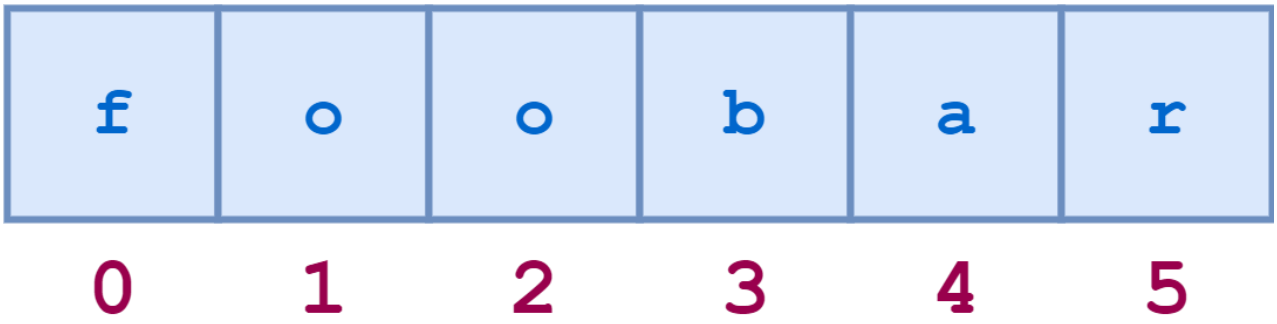
Индексация строк

Часто в языках программирования, отдельные элементы в упорядоченном наборе данных могут быть доступны с помощью числового индекса или ключа. Этот процесс называется индексация.

В Python строки являются упорядоченными последовательностями символьных данных и могут быть проиндексированы. Доступ к отдельным символам в строке можно получить, указав имя строки, за которым следует число в квадратных скобках `[]`.

Индексация строк начинается с нуля: у первого символа индекс `0`, следующего `1` и так далее. Индекс последнего символа в python — "длина строки минус один".

Например, схематическое представление индексов строки `'foobar'` выглядит следующим образом:



Отдельные символы доступны по индексу следующим образом:

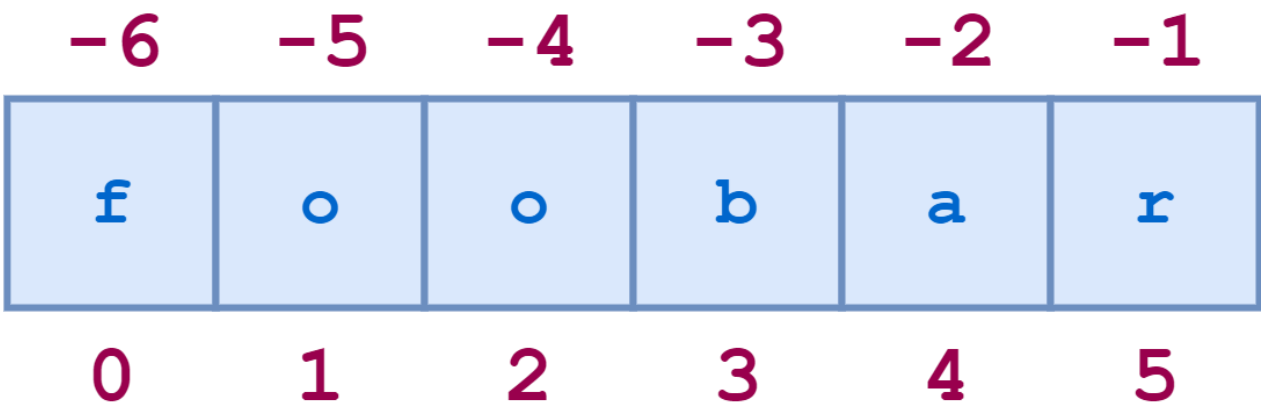
```
>>> s = 'foobar'


>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> s[5]
'r'
```

Попытка обращения по индексу большему чем `len(s) - 1`, приводит к ошибке `IndexError`:

```
>>> s[6]
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    s[6]
IndexError: string index out of range
```

Индексы строк также могут быть указаны отрицательными числами. В этом случае индексирование начинается с конца строки: `-1` относится к последнему символу, `-2` к предпоследнему и так далее. Вот такая же диаграмма, показывающая как положительные, так и отрицательные индексы строки `'foobar'`:



Вот несколько примеров отрицательного индексирования:

```
>>> s = 'foobar'
>>> s[-1]
'r'
>>> s[-2]
'a'
>>> len(s)
6
>>> s[-len(s)] # отрицательная индексация начинается с -1
'f'
```

Попытка обращения по индексу меньшему чем `-len(s)`, приводит к ошибке `IndexError`:

```
>>> s[-7]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    s[-7]
IndexError: string index out of range
```

Для любой непустой строки `s`, код `s[len(s)-1]` и `s[-1]` возвращают последний символ. Нет индекса, который применим к пустой строке.

Срезы строк

Python также допускает возможность извлечения подстроки из строки, известную как "string slice". Если `s` это строка, выражение формы `s[m:n]` возвращает часть `s`, начинающуюся с позиции `m`, и до позиции `n`, но не включая позицию:

```
>>> s = 'python'
>>> s[2:5]
'tho'
```

Примечание: индексы строк в python начинаются с нуля. Первый символ в строке имеет индекс `0`. Это относится и к срезу.

Опять же, второй индекс указывает символ, который не включен в результат. Символ `'n'` в приведенном выше примере. Это может показаться немного не интуитивным, но дает результат: выражение `s[m:n]` вернет подстроку, которая является разницей `n - m`, в данном случае `5 - 2 = 3`.

Если пропустить первый индекс, срез начинается с начала строки. Таким образом, `s[:m]` = `s[0:m]`:

```
>>> s = 'python'
```

```
>>> s[:4]
'pyth'
>>> s[0:4]
'pyth'
```

Аналогично, если опустить второй индекс `s[n:]`, срез длится от первого индекса до конца строки. Это хорошая, лаконичная альтернатива более громоздкой `s[n:len(s)]`:

```
>>> s = 'python'
>>> s[2:]
'thon'
>>> s[2:len(s)]
'thon'
```

Для любой строки `s` и любого целого `n` числа (`0 ≤ n ≤ len(s)`), `s[:n] + s[n:]`будет `s`:

```
>>> s = 'python'
>>> s[:4] + s[4:]
'python'
>>> s[:4] + s[4:] == s
True
```

Пропуск обоих индексов возвращает исходную строку. Это не копия, это ссылка на исходную строку:

```
>>> s = 'python'
>>> t = s[:]
```

```
>>> id(s)
59598496
>>> id(t)
59598496
>>> s is t
True
```

Если первый индекс в срезе больше или равен второму индексу, Python возвращает пустую строку. Это еще один не очевидный способ сгенерировать пустую строку, если вы его искали:

```
>>> s[2:2]
''
>>> s[4:2]
''
```

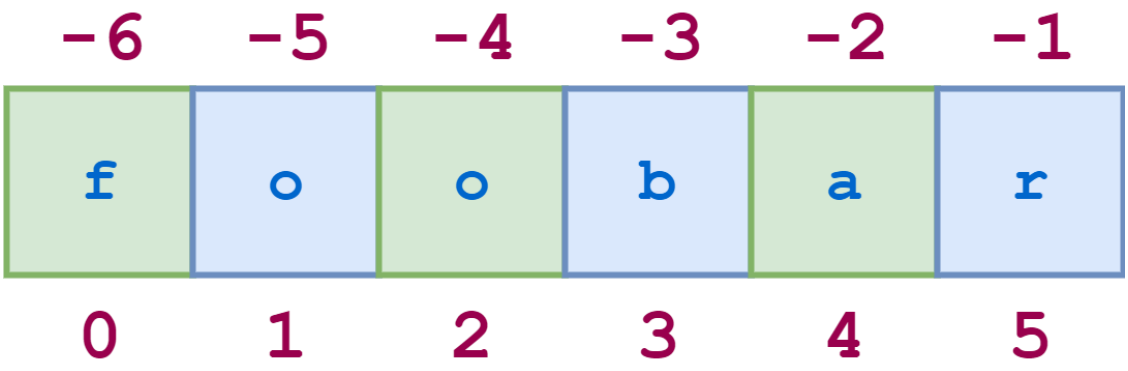
Отрицательные индексы можно использовать и со срезами. Вот пример кода Python:

```
>>> s = 'python'
>>> s[-5:-2]
'yth'
>>> s[1:4]
'yth'
>>> s[-5:-2] == s[1:4]
True
```

Шаг для среза строки

Существует еще один вариант синтаксиса среза, о котором стоит упомянуть. Добавление дополнительного `:` и третьего индекса означает шаг, который указывает, сколько символов следует пропустить после извлечения каждого символа в срезе.

Например , для строки `'python'` срез `0:6:2` начинается с первого символа и заканчивается последним символом (всей строкой), каждый второй символ пропускается. Это показано на следующей схеме:



Иллюстративный код показан здесь:

```
>>> s = 'foobar'
>>> s[0:6:2]
'foa'
>>> s[1:6:2]
'obr'
```

Как и в случае с простым срезом, первый и второй индексы могут быть пропущены:

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::5]
```

```
'11111'
>>> s[4::5]
'55555'
```

Вы также можете указать отрицательное значение шага, в этом случае Python идет с конца строки. Начальный/первый индекс должен быть больше конечного/второго индекса:

```
>>> s = 'python'
>>> s[5:0:-2]
'nhy'
```

В приведенном выше примере, `5:0:-2` означает «начать с последнего символа и делать два шага назад, но не включая первый символ.”

Когда вы идете назад, если первый и второй индексы пропущены, значения по умолчанию применяются так: первый индекс — конец строки, а второй индекс — начало. Вот пример:

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::-5]
'55555'
```

Это общая парадигма для разворота (reverse) строки:

```
>>> s = 'Если так говорит товарищ Наполеон, значит, так оно и есть.'
>>> s[::-1]
'.ьтсе и оно кат ,тичанз ,ноелопаН щиравот тировог кат илсЕ'
```

Форматирование строки

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

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:

```
>>> var = 'Гав'
>>> print(f'Собака говорит {var}!')
Собака говорит Гав!
>>> print(f"Собака говорит {var}!")
Собака говорит Гав!
>>> print(f'''Собака говорит {var}!''')
Собака говорит Гав!
```

Изменение строк

Строки — один из типов данных, которые Python считает неизменяемыми, что означает невозможность их изменять. Как вы ниже увидите, python дает возможность 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'
```

Встроенные методы строк в python

Python — это объектно-ориентированный язык. Каждый элемент данных в программе python является объектом.

Методы похожи на функции. Метод — специализированный тип вызываемой процедуры, тесно связанный с объектом. Как и функция, метод вызывается для выполнения отдельной задачи, но он вызывается только вместе с определенным объектом и знает о нем во время выполнения.

Синтаксис для вызова метода объекта выглядит следующим образом:

```
obj.foo(<args>)
```

Этот код вызывает метод `.foo()` объекта `obj`. `<args>` — аргументы, передаваемые методу (если есть).

В приведенных методах аргументы, указанные в квадратных скобках (`[]`), являются необязательными.

Изменение регистра строки

Методы этой группы выполняют преобразование регистра строки.

`string.capitalize()` приводит первую букву в верхний регистр, остальные в нижний.

`s.capitalize()` возвращает копию `s` с первым символом, преобразованным в верхний регистр, и остальными символами, преобразованными в нижний регистр:

```
>>> s = 'everyTHing yoU Can IMaGine is rEAl'
>>> s.capitalize()
'Everything you can imagine is real'
```

Не алфавитные символы не изменяются:

```
>>> s = 'follow us @PYTHON'
>>> s.capitalize()
'Follow us @python'
```

`string.lower()` преобразует все буквенные символы в строчные.

`s.lower()` возвращает копию `s` со всеми буквенными символами, преобразованными в нижний регистр:

```
>>> 'everyTHing yoU Can IMaGine is rEAl'.lower()
'everything you can imagine is real'
```

`string.swapcase()` меняет регистр буквенных символов на противоположный.

`s.swapcase()` возвращает копию `s` с заглавными буквенными символами, преобразованными в строчные и наоборот:

```
>>> 'everyTHing yoU Can IMaGine is rEAl'.swapcase()
'EVERYthING YOu cAN imAgINE IS ReaL'
```

`string.title()` преобразует первые буквы всех слов в заглавные

`s.title()` возвращает копию, `s` в которой первая буква каждого слова преобразуется в верхний регистр, а остальные буквы — в нижний регистр:

```
>>> 'the sun also rises'.title()
'The Sun Also Rises'
```

Этот метод использует довольно простой алгоритм. Он не пытается различить важные и неважные слова и не обрабатывает апострофы, имена или аббревиатуры:

```
>>> 'follow us @PYTHON'.title()
'Follow Us @Python'
```

`string.upper()` преобразует все буквенные символы в заглавные.

`s.upper()` возвращает копию `s` со всеми буквенными символами в верхнем регистре:

```
>>> 'follow us @PYTHON'.upper()
'FOLLOW US @PYTHON'
```

Найти и заменить подстроку в строке

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

Каждый метод в этой группе поддерживает необязательные аргументы `<start>` и `<end>` аргументы. Они задают диапазон поиска: действие метода ограничено частью целевой строки, начинающейся в позиции символа `<start>` и продолжающейся вплоть до позиции символа `<end>`, но не включая его. Если `<start>` указано, а `<end>` нет, метод применяется к части строки от `<start>` конца.

`string.count(<sub>[, <start>[, <end>]])` подсчитывает количество вхождений подстроки в строку.

`s.count(<sub>)` возвращает количество точных вхождений подстроки `<sub>` в `s`:

```
>>> 'foo goo moo'.count('oo')
3
```

Количество вхождений изменится, если указать `<start>` и `<end>`:

```
>>> 'foo goo moo'.count('oo', 0, 8)
2
```

`string.endswith(<suffix>[, <start>[, <end>]])` определяет, заканчивается ли строка заданной подстрокой.

`s.endswith(<suffix>)` возвращает, `True` если `s` заканчивается указанным `<suffix>` и `False` если нет:

```
>>> 'python'.endswith('on')
True
>>> 'python'.endswith('or')
False
```

Сравнение ограничено подстрокой, между `<start>` и `<end>`, если они указаны:

```
>>> 'python'.endswith('yt', 0, 4)
True
>>> 'python'.endswith('yt', 2, 4)
False
```

`string.find(<sub>[, <start>[, <end>]])` ищет в строке заданную подстроку.

`s.find(<sub>)` возвращает первый индекс в `s` который соответствует началу строки `<sub>`:

```
>>> 'Follow Us @Python'.find('Us')
7
```

Этот метод возвращает, `-1` если указанная подстрока не найдена:

```
>>> 'Follow Us @Python'.find('you')
-1
```

Поиск в строке ограничивается подстрокой, между `<start>` и `<end>`, если они указаны:

```
>>> 'Follow Us @Python'.find('Us', 4)
7
>>> 'Follow Us @Python'.find('Us', 4, 7)
-1
```

`string.index(<sub>[, <start>[, <end>]])` ищет в строке заданную подстроку.

Этот метод идентичен `.find()`, за исключением того, что он вызывает исключение `ValueError`, если `<sub>` не найден:

```
>>> 'Follow Us @Python'.index('you')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Follow Us @Python'.index('you')
ValueError: substring not found
```

`string.rfind(<sub>[, <start>[, <end>]])` ищет в строке заданную подстроку, начиная с конца.

`s.rfind(<sub>)` возвращает индекс последнего вхождения подстроки `<sub>` в `s`, который соответствует началу `<sub>`:

```
>>> 'Follow Us @Python'.rfind('o')
15
```

Как и в `.find()`, если подстрока не найдена, возвращается `-1`:

```
>>> 'Follow Us @Python'.rfind('a')
-1
```

Поиск в строке ограничивается подстрокой, между `<start>` и `<end>`, если они указаны:

```
>>> 'Follow Us @Python'.rfind('Us', 0, 14)
7
>>> 'Follow Us @Python'.rfind('Us', 9, 14)
-1
```

`string.rindex(<sub>[, <start>[, <end>]])` ищет в строке заданную подстроку, начиная с конца.

Этот метод идентичен `.rfind()`, за исключением того, что он вызывает исключение `ValueError`, если `<sub>` не найден:

```
>>> 'Follow Us @Python'.rindex('you')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Follow Us @Python'.rindex('you')
ValueError: substring not found
```

`string.startswith(<prefix>[, <start>[, <end>]])` определяет, начинается ли строка с заданной подстроки.

`s.startswith(<suffix>)` возвращает, `True` если `s` начинается с указанного `<suffix>` и `False` если нет:

```
>>> 'Follow Us @Python'.startswith('Fol')
True
>>> 'Follow Us @Python'.startswith('Go')
False
```

Сравнение ограничено подстрокой, между `<start>` и `<end>`, если они указаны:

```
>>> 'Follow Us @Python'.startswith('Us', 7)
True
>>> 'Follow Us @Python'.startswith('Us', 8, 16)
False
```

Классификация строк

Методы в этой группе классифицируют строку на основе символов, которые она содержит.

`string.isalnum()` определяет, состоит ли строка из букв и цифр.

`s.isalnum()` возвращает `True`, если строка `s` не пустая, а все ее символы буквенно-цифровые (либо буква, либо цифра). В другом случае `False` :

```
>>> 'abc123'.isalnum()
True
>>> 'abc$123'.isalnum()
False
>>> ''.isalnum()
False
```

`string.isalpha()` определяет, состоит ли строка только из букв.

`s.isalpha()` возвращает `True`, если строка `s` не пустая, а все ее символы буквенные. В другом случае `False`:

```
>>> 'ABCabc'.isalpha()
```

```
True
>>> 'abc123'.isalpha()
False
```

`string.isdigit()` определяет, состоит ли строка из цифр (проверка на число).

`s.digit()` возвращает `True` когда строка `s` не пустая и все ее символы являются цифрами, а в `False` если нет:

```
>>> '123'.isdigit()
True
>>> '123abc'.isdigit()
False
```

`string.isidentifier()` определяет, является ли строка допустимым идентификатором Python.

`s.isidentifier()` возвращает `True`, если `s` валидный идентификатор (название переменной, функции, класса и т.д.) python, а в `False` если нет:

```
>>> 'foo32'.isidentifier()
True
>>> '32foo'.isidentifier()
False
>>> 'foo$32'.isidentifier()
False
```

Важно: `.isidentifier()` вернет `True` для строки, которая соответствует зарезервированному ключевому слову python, даже если его нельзя использовать:

```
>>> 'and'.isidentifier()

True
```

Вы можете проверить, является ли строка ключевым словом Python, используя функцию `iskeyword()`, которая находится в модуле `keyword`. Один из возможных способов сделать это:

```
>>> from keyword import iskeyword

>>> iskeyword('and')

True
```

Если вы действительно хотите убедиться, что строку можно использовать как идентификатор python, вы должны проверить, что `.isidentifier()` = `True` и `iskeyword()` = `False`.

`string.islower()` определяет, являются ли буквенные символы строки строчными.

`s.islower()` возвращает `True`, если строка `s` не пустая, и все содержащиеся в нем буквенные символы строчные, а `False` если нет. Не алфавитные символы игнорируются:

```
>>> 'abc'.islower()

True

>>> 'abc1$d'.islower()

True

>>> 'Abc1$D'.islower()

False
```

`string.isprintable()` определяет, состоит ли строка только из печатаемых символов.

`s.isprintable()` возвращает, `True` если строка `s` пустая или все буквенные символы которые она содержит можно вывести на экран. Возвращает, `False` если `s` содержит хотя бы один специальный символ. Не алфавитные символы игнорируются:

```
>>> 'a\tb'.isprintable() # \t - символ табуляции
False
>>> 'a b'.isprintable()
True
>>> ''.isprintable()
True
>>> 'a\nb'.isprintable() # \n - символ перевода строки
False
```

Важно: Это единственный `.is****()` метод, который возвращает `True`, если `s` пустая строка. Все остальные возвращаются `False`.

`string.isspace()` определяет, состоит ли строка только из пробельных символов.

`s.isspace()` возвращает `True`, если `s` не пустая строка, и все символы являются пробельными, а `False`, если нет.

Наиболее часто встречающиеся пробельные символы — это пробел `' '`, табуляция `'\t'` и новая строка `'\n'`:

```
>>> ' \t  \n '.isspace()
True
>>> ' a '.isspace()
False
```

Тем не менее есть несколько символов ASCII, которые считаются пробелами. И если учитывать символы Юникода, их еще больше:

```
>>> '\f\u2005\r'.isspace()
True
```

`'\f'` и `'\r'` являются escape-последовательностями для символов ASCII; `'\u2005'` это escape-последовательность для Unicode.

`string.istitle()` определяет, начинаются ли слова строки с заглавной буквы.

`s.istitle()` возвращает `True` когда `s` не пустая строка и первый алфавитный символ каждого слова в верхнем регистре, а все остальные буквенные символы в каждом слове строчные. Возвращает `False`, если нет:

```
>>> 'This Is A Title'.istitle()
True
>>> 'This is a title'.istitle()
False
>>> 'Give Me The #$#@ Ball!'.istitle()
True
```

`string.isupper()` определяет, являются ли буквенные символы строки заглавными.

`s.isupper()` возвращает `True`, если строка `s` не пустая, и все содержащиеся в ней буквенные символы являются заглавными, и в `False`, если нет. Не алфавитные символы игнорируются:

```
>>> 'ABC'.isupper()
```

```
True
>>> 'ABC1$D'.isupper()
True
>>> 'Abc1$D'.isupper()
False
```

Выравнивание строк, отступы

Методы в этой группе влияют на вывод строки.

`string.center(<width>[, <fill>])` выравнивает строку по центру.

`s.center(<width>)` возвращает строку, состоящую из `s` выровненной по ширине `<width>`. По умолчанию отступ состоит из пробела ASCII:

```
>>> 'py'.center(10)
'    py    '
```

Если указан необязательный аргумент `<fill>`, он используется как символ заполнения:

```
>>> 'py'.center(10, '-')
'----py----'
```

Если `s` больше или равна `<width>`, строка возвращается без изменений:

```
>>> 'python'.center(2)
'python'
```

`string.expandtabs(tabsize=8)` заменяет табуляции на пробелы

`s.expandtabs()` заменяет каждый символ табуляции (`'\t'`) пробелами. По умолчанию табуляция заменяются на 8 пробелов:

```
>>> 'a\tb\tc'.expandtabs()
'a       b       c'
>>> 'aaa\tbbb\tc'.expandtabs()
'aaa     bbb     c'
```

`tabsize` необязательный параметр, задающий количество пробелов:

```
>>> 'a\tb\tc'.expandtabs(4)
'a   b   c'
>>> 'aaa\tbbb\tc'.expandtabs(tabsize=4)
'aaa bbb c'
```

`string.ljust(<width>[, <fill>])` выравнивание по левому краю строки в поле.

`s.ljust(<width>)` возвращает строку `s`, выравненную по левому краю в поле шириной `<width>`. По умолчанию отступ состоит из пробела ASCII:

```
>>> 'python'.ljust(10)
'python    '
```

Если указан аргумент `<fill>` , он используется как символ заполнения:

```
>>> 'python'.ljust(10, '-')
'python----'
```

Если `s` больше или равна `<width>`, строка возвращается без изменений:

```
>>> 'python'.ljust(2)
'python'
```

`string.lstrip([<chars>])` обрезает пробельные символы слева

`s.lstrip()`возвращает копию `s` в которой все пробельные символы с левого края удалены:

```
>>> '   foo bar baz   '.lstrip()
'foo bar baz   '
>>> '\t\nfoo\t\nbar\t\nbaz'.lstrip()
'foo\t\nbar\t\nbaz'
```

Необязательный аргумент `<chars>`, определяет набор символов, которые будут удалены:

```
>>> 'https://www.pythonru.com'.lstrip('/:pths')
'www.pythonru.com'
```

`string.replace(<old>, <new>[, <count>])` заменяет вхождения подстроки в строке.

`s.replace(<old>, <new>)` возвращает копию `s` где все вхождения подстроки `<old>`, заменены на `<new>`:

```
>>> 'I hate python! I hate python! I hate python!'.replace('hate', 'love')
'I love python! I love python! I love python!'
```

Если указан необязательный аргумент `<count>`, выполняется количество `<count>` замен:

```
>>> 'I hate python! I hate python! I hate python!'.replace('hate', 'love', 2)
'I love python! I love python! I hate python!'
```

`string.rjust(<width>[, <fill>])` выравнивание по правому краю строки в поле.

`s.rjust(<width>)` возвращает строку `s`, выравненную по правому краю в поле шириной `<width>`. По умолчанию отступ состоит из пробела ASCII:

```
>>> 'python'.rjust(10)
'    python'
```

Если указан аргумент `<fill>` , он используется как символ заполнения:

```
>>> 'python'.rjust(10, '-')
'----python'
```

Если `s` больше или равна `<width>`, строка возвращается без изменений:

```
>>> 'python'.rjust(2)
'python'
```

`string.rstrip([<chars>])` обрезает пробельные символы справа

`s.rstrip()` возвращает копию `s` без пробельных символов, удаленных с правого края:

```
>>> '   foo bar baz   '.rstrip()
'   foo bar baz'
>>> 'foo\t\nbar\t\nbaz\t\n'.rstrip()
```

```
'foo\t\nbar\t\nbaz'
```

Необязательный аргумент `<chars>`, определяет набор символов, которые будут удалены:

```
>>> 'foo.$$$;'.rstrip(';$.')
'foo'
```

`string.strip([<chars>])` удаляет символы с левого и правого края строки.

`s.strip()` эквивалентно последовательному вызову `s.lstrip()`и `s.rstrip()`. Без аргумента `<chars>` метод удаляет пробелы в начале и в конце:

```
>>> s = '   foo bar baz\t\t\t'
>>> s = s.lstrip()
>>> s = s.rstrip()
>>> s
'foo bar baz'
```

Как в `.lstrip()` и `.rstrip()`, необязательный аргумент `<chars>` определяет набор символов, которые будут удалены:

```
>>> 'www.pythonru.com'.strip('w.moc')
'pythonru'
```

Важно: Когда возвращаемое значение метода является другой строкой, как это часто бывает, методы можно вызывать последовательно:

```
>>> '   foo bar baz\t\t\t'.lstrip().rstrip()
```

```
'foo bar baz'

>>> '   foo bar baz\t\t\t'.strip()

'foo bar baz'


>>> 'www.pythonru.com'.lstrip('w.').rstrip('.moc')

'pythonru'

>>> 'www.pythonru.com'.strip('w.moc')

'pythonru'
```

`string.zfill(<width>)` дополняет строку нулями слева.

`s.zfill(<width>)` возвращает копию `s` дополненную `'0'` слева для достижения длины строки указанной в `<width>`:

```
>>> '42'.zfill(5)

'00042'
```

Если `s` содержит знак перед цифрами, он остается слева строки:

```
>>> '+42'.zfill(8)

'+0000042'

>>> '-42'.zfill(8)

'-0000042'
```

Если `s` больше или равна `<width>`, строка возвращается без изменений:

```
>>> '-42'.zfill(3)

'-42'
```

`.zfill()` наиболее полезен для строковых представлений чисел, но python с удовольствием заполнит строку нулями, даже если в ней нет чисел:

```
>>> 'foo'.zfill(6)
'000foo'
```

Методы преобразование строки в список

Методы в этой группе преобразовывают строку в другой тип данных и наоборот. Эти методы возвращают или принимают итерируемые объекты — термин Python для последовательного набора объектов.

Многие из этих методов возвращают либо список, либо кортеж. Это два похожих типа данных, которые являются прототипами примеров итераций в python. Список заключен в квадратные скобки (`[]`), а кортеж заключен в простые (`()`).

Теперь давайте посмотрим на последнюю группу строковых методов.

`string.join(<iterable>)` объединяет список в строку.

`s.join(<iterable>)` возвращает строку, которая является результатом конкатенации объекта `<iterable>` с разделителем `s`.

Обратите внимание, что `.join()` вызывается строка-разделитель `s` . `<iterable>` должна быть последовательностью строковых объектов.

Примеры кода помогут вникнуть. В первом примере разделителем `s` является строка `', '`, а `<iterable>` список строк:

```
>>> ', '.join(['foo', 'bar', 'baz', 'qux'])
'foo, bar, baz, qux'
```

В результате получается одна строка, состоящая из списка объектов, разделенных запятыми.

В следующем примере `<iterable>` указывается как одно строковое значение. Когда строковое значение используется в качестве итерируемого, оно интерпретируется как список отдельных символов строки:

```
>>> list('corge')
['c', 'o', 'r', 'g', 'e']


>>> ':'.join('corge')
'c:o:r:g:e'
```

Таким образом, результатом `':'.join('corge')` является строка, состоящая из каждого символа в `'corge'`, разделенного символом `':'`.

Этот пример завершается с ошибкой `TypeError`, потому что один из объектов в `<iterable>` не является строкой:

```
>>> '---'.join(['foo', 23, 'bar'])
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '---'.join(['foo', 23, 'bar'])
TypeError: sequence item 1: expected str instance, int found
```

Это можно исправить так:

```
>>> '---'.join(['foo', str(23), 'bar'])
'foo---23---bar'
```

Как вы скоро увидите, многие объекты в Python можно итерировать, и `.join()` особенно полезен для создания из них строк.

`string.partition(<sep>)` делит строку на основе разделителя.

`s.partition(<sep>)` отделяет от `s` подстроку длиной от начала до первого вхождения `<sep>`. Возвращаемое значение представляет собой кортеж из трех частей:

- Часть `s` до `<sep>`
- Разделитель `<sep>`
- Часть `s` после `<sep>`

Вот пара примеров `.partition()`в работе:

```
>>> 'foo.bar'.partition('.')
('foo', '.', 'bar')
>>> 'foo@@bar@@baz'.partition('@@')
('foo', '@@', 'bar@@baz')
```

Если `<sep>` не найден в `s`, возвращаемый кортеж содержит `s` и две пустые строки:

```
>>> 'foo.bar'.partition('@@')
('foo.bar', '', '')
```

`s.rpartition(<sep>)` делит строку на основе разделителя, начиная с конца.

`s.rpartition(<sep>)` работает как `s.partition(<sep>)`, за исключением того, что `s` делится при последнем вхождении `<sep>` вместо первого:

```
>>> 'foo@@bar@@baz'.partition('@@')
```

```
('foo', '@@', 'bar@@baz')


>>> 'foo@@bar@@baz'.rpartition('@@')
('foo@@bar', '@@', 'baz')
```

`string.rsplit(sep=None, maxsplit=-1)` делит строку на список из подстрок.

Без аргументов `s.rsplit()` делит `s` на подстроки, разделенные любой последовательностью пробелов, и возвращает список:

```
>>> 'foo bar baz qux'.rsplit()
['foo', 'bar', 'baz', 'qux']
>>> 'foo\n\tbar   baz\r\fqux'.rsplit()
['foo', 'bar', 'baz', 'qux']
```

Если `<sep>` указан, он используется в качестве разделителя:

```
>>> 'foo.bar.baz.qux'.rsplit(sep='.')
['foo', 'bar', 'baz', 'qux']
```

Если `<sep>` = `None`, строка разделяется пробелами, как если бы `<sep>` не был указан вообще.

Когда `<sep>` явно указан в качестве разделителя `s`, последовательные повторы разделителя будут возвращены как пустые строки:

```
>>> 'foo...bar'.rsplit(sep='.')
['foo', '', '', 'bar']
```

Это не работает, когда `<sep>` не указан. В этом случае последовательные пробельные символы объединяются в один разделитель, и результирующий список никогда не будет содержать пустых строк:

```
>>> 'foo\t\t\tbar'.rsplit()
['foo', 'bar']
```

Если указан необязательный параметр `<maxsplit>`, выполняется максимальное количество разделений, начиная с правого края `s`:

```
>>> 'www.pythonru.com'.rsplit(sep='.', maxsplit=1)
['www.pythonru', 'com']
```

Значение по умолчанию для `<maxsplit>` — `-1`. Это значит, что все возможные разделения должны быть выполнены:

```
>>> 'www.pythonru.com'.rsplit(sep='.', maxsplit=-1)
['www', 'pythonru', 'com']
>>> 'www.pythonru.com'.rsplit(sep='.')
['www', 'pythonru', 'com']
```

`string.split(sep=None, maxsplit=-1)` делит строку на список из подстрок.

`s.split()` ведет себя как `s.rsplit()`, за исключением того, что при указании `<maxsplit>`, деление начинается с левого края `s`:

```
>>> 'www.pythonru.com'.split('.', maxsplit=1)
['www', 'pythonru.com']
>>> 'www.pythonru.com'.rsplit('.', maxsplit=1)
```

```
['www.pythonru', 'com']
```

Если `<maxsplit>` не указано, между `.rsplit()` и `.split()` в python разницы нет.

`string.splitlines([<keepends>])` делит текст на список строк.

`s.splitlines()` делит `s` на строки и возвращает их в списке. Любой из следующих символов или последовательностей символов считается границей строки:

Разделитель	**Значение**
\n	Новая строка
\r	Возврат каретки
\r\n	Возврат каретки + перевод строки
\v или же \x0b	Таблицы строк
\f или же \x0c	Подача формы
\x1c	Разделитель файлов
\x1d	Разделитель групп
\x1e	Разделитель записей
\x85	Следующая строка

Разделитель	**Значение**
\u2028	Новая строка (Unicode)
\u2029	Новый абзац (Unicode)

Вот пример использования нескольких различных разделителей строк:

```
>>> 'foo\nbar\r\nbaz\fqux\u2028quux'.splitlines()
['foo', 'bar', 'baz', 'qux', 'quux']
```

Если в строке присутствуют последовательные символы границы строки, они появятся в списке результатов, как пустые строки:

```
>>> 'foo\f\f\fbar'.splitlines()
['foo', '', '', 'bar']
```

Если необязательный аргумент `<keepends>` указан и его булевое значение `True`, то символы границы строк сохраняются в списке подстрок:

```
>>> 'foo\nbar\nbaz\nqux'.splitlines(True)
['foo\n', 'bar\n', 'baz\n', 'qux']
>>\> 'foo\nbar\nbaz\nqux'.splitlines(8)
['foo\n', 'bar\n', 'baz\n', 'qux']
```