Python: работа со строками

Введение

Строки в языке Python являются типом данных, специально предназначенным для обработки текстовой информации. Строка может содержать произвольно длинный текст (ограниченный имеющейся памятью).

Форматирование строк в Python

Существует четыре разных подхода к форматированию строк:

1. "Старый стиль" форматирования строк (оператор %)
2. "Новый стиль" форматирования строк (str.format)
3. Интерполяция строк / f-Strings (Python 3.6+)
4. Шаблоны (Standard Library)

"Старый стиль" форматирования строк (оператор %)

"Старый стиль" форматирования строк (оператор %)

Во многом аналогичен функции printf в C.

Подробнее можно прочитать в документации

https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

```
name = "Eric"
"Hello, %s." % name
```

Результат: Hello, Eric.

Поскольку оператор % принимает только один аргумент, для случая с несколькими переменными нужно обернуть правую часть в кортеж, например, так:

```
name = "Eric"
age = 74
"Hello, %s. You are %s." % (name, age)
```

Hello, Eric. You are 74.

Если переменных много, что код быстро становиться плохо читаемым:

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
"Hello, %s %s. You are %s. You are a %s. You were a member of %s." %
(first_name, last_name, age, profession, affiliation)
```

```
Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.
```

Можно использовать спецификатор формата %x для преобразования значения int в строку и представления его в виде шестнадцатеричного числа:

```
errno = 50159747054
name = 'Bob'

'Hey %s, there is a 0x%x error!' %
(name, errno)
```

Результат:

```
Hey Bob, there is a 0xbadc0ffee error!
```

Также можно производить подстановку переменных по имени:

```
'Hey %(name)s, there is a 0x%(errno)x error!' % {"name": name, "errno": errno }
```

```
Hey Bob, there is a 0xbadc0ffee error!
```

К сожалению, этот вид форматирования не очень хорош, потому что он многословен и приводит к ошибкам, таким как неправильное отображение кортежей или словарей.

Официальная документация Python 3 не рекомендует форматирование «старого стиля» и говорит о нем не слишком любезно:

“The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the str.format() interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.” (Source)

https://docs.python.org/3/library/stdtypes.html?highlight=sprintf#printf-style-string-formatting

"Новый стиль" (str.format)

"Новый стиль" (str.format)

str.format () - это улучшение % форматирования. Он использует обычный синтаксис вызова функции и может быть расширен с помощью переопределения метода __format__() для объекта, преобразуемого в строку.

https://www.python.org/dev/peps/pep-3101/#controlling-formatting-on-a-per-type-basis

Был представлен в Python 2.6

https://docs.python.org/3/library/stdtypes.html#str.format

Поля замены отмечены фигурными скобками:

```
"Hello, {}. You are {}.".format(name, age)
```

Hello, Bob. You are 74.

Можно ссылаться на переменные в любом порядке, используя их индекс:

```
"Hello, {1}. You are {0}.".format(age, name)
```

```
Hello, Bob. You are 74.
```

Но если вы вставите имена переменных, вы получите дополнительную возможность передавать объекты, а затем ссылаться на параметры и методы между фигурными скобками:

```
person = {'name': 'Eric', 'age': 74}

"Hello, {name}. You are {age}.".format(name=person['name'], age=person['age'])
```

```
Hello, Eric. You are 74.
```

Также часто удобно использовать ** , для вывода значений из словаря:

```
"Hello, {name}. You are {age}.".format(**person)
```

```
Hello, Eric. You are 74.
```

Недостатки: Хотя код, использующий str.format (), гораздо легче читается, в сравнении с %-форматированием, однако str.format () все еще может быть слишком громоздким, когда строка длинная и параметров много.

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print(("Hello, {first_name} {last_name}. You are {age}. " +
      "You are a {profession}. You were a member of {affiliation}.") \
      .format(first_name=first_name, last_name=last_name, age=age, \
      profession=profession, affiliation=affiliation))
```

```
Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.
```

f-Strings / Интерполяция строк

f-Strings:

новый и улучшенный способ форматирования строк в Python

Поддержка добавлена начиная с Python 3.6. Вы можете прочитать все об этом в PEP 498. https://www.python.org/dev/peps/pep-0498/

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

```
name = "Eric"
age = 74

f"Hello, {name}. You are {age}."
```

Hello, Eric. You are 74.

Произвольные выражения:

Поскольку f-строки вычисляются во время выполнения, в них можно поместить все допустимые выражения Python.

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

```
print( f"{2 * 37}" )
74
```

```
def to_lowercase(input):
    return input.lower()

name = "Eric Idle"
print(f"{to_lowercase(name)} is funny.")

eric idle is funny.
```

Можно вызвать метод напрямую:

```
print( f"{name.lower()} is funny." )
```

```
eric idle is funny.
```

Другой пример:

```
print(f'Если поделить торт на трех человек, каждый из них получит {1/3*100:.2f} %')
```

```
Если поделить торт на трех человек, каждый из них получит 33.33 %
```

Пример с классом:

```
class Comedian:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __str__(self):
        return f"{self.first_name} {self.last_name} is {self.age}."

    def __repr__(self):
        return f"{self.first_name} {self.last_name} is {self.age}. Surprise!"

new_comedian = Comedian("Eric", "Idle", "74")
print(f"{new_comedian}")
```

Eric Idle is 74.

```
print(f"{new_comedian!r}")
```

Eric Idle is 74. Surprise!

Многострочные f-строки

```
name = "Eric"
profession = "comedian"
affiliation = "Monty Python"
message = (
    f"Hi {name}. "
    f"You are a {profession}. "
    f"You were in {affiliation}."
    )
print(message)
```

```
Hi Eric. You are a comedian. You were in Monty Python.
```

Важно, чтобы f было перед каждой строкой многострочной строки. Следующий код не будет работать:

```
message = (
    f"Hi {name}. "
    "You are a {profession}. "
    "You were in {affiliation}."
    )
print(message)
```

Hi Eric. You are a {profession}. You were in {affiliation}.

При использовании """ строк:

```
message = f"""
    Hi {name}.
    You are a {profession}.
    You were in {affiliation}.
"""

print(message)
```

```
Hi Eric.
You are a comedian.
You were in Monty Python.
```

Скорость

Символ f в f-string также может означать «быстро» (fast).

f-строки быстрее, чем %-форматирование и str.format().

f-строки - это выражения, вычисляемые во время выполнения.

```
import timeit

timeit.timeit("""name = "Eric" age = 74 '%s is %s.'
% (name, age)""", number = 10000)

0.004173202378340368

timeit.timeit("""name = "Eric" age = 74 '{} is
{}.'.format(name, age)""", number = 10000)

0.0048879855108212 3

timeit.timeit("""name = "Eric" age = 74 f'{name}
is {age}.'""", number = 10000)

0.0025852118251977196
```

Может быть потенциально небезопасно, если строка получается от пользователя.

```
# Это наш супер секретный ключ:
SECRET = 'this-is-a-secret'

class Error:
    def __init__(self):
        pass

# Злонамеренный пользователь может ввести строку, которая
# может прочитать данные из глобального пространства имен:
user_input = '{error.__init__.__globals__[SECRET]}'

# Что позволит получить секретную информацию,
# такую как секретный ключ:
err = Error()
print(user_input.format(error=err))
```

```
this-is-a-secret
```

Шаблонные строки (Стандартная библиотека Template Strings)

Шаблоны (Standard Library)

Еще один инструмент для форматирования строк в Python: шаблоны. Был добавлен в Python 2.4.

Это более простой и менее мощный механизм, но в некоторых случаях это может быть именно то, что нужно.

```
from string import Template

t = Template('Hey, $name!')

s = t.substitute(name=name)

print(s)
```

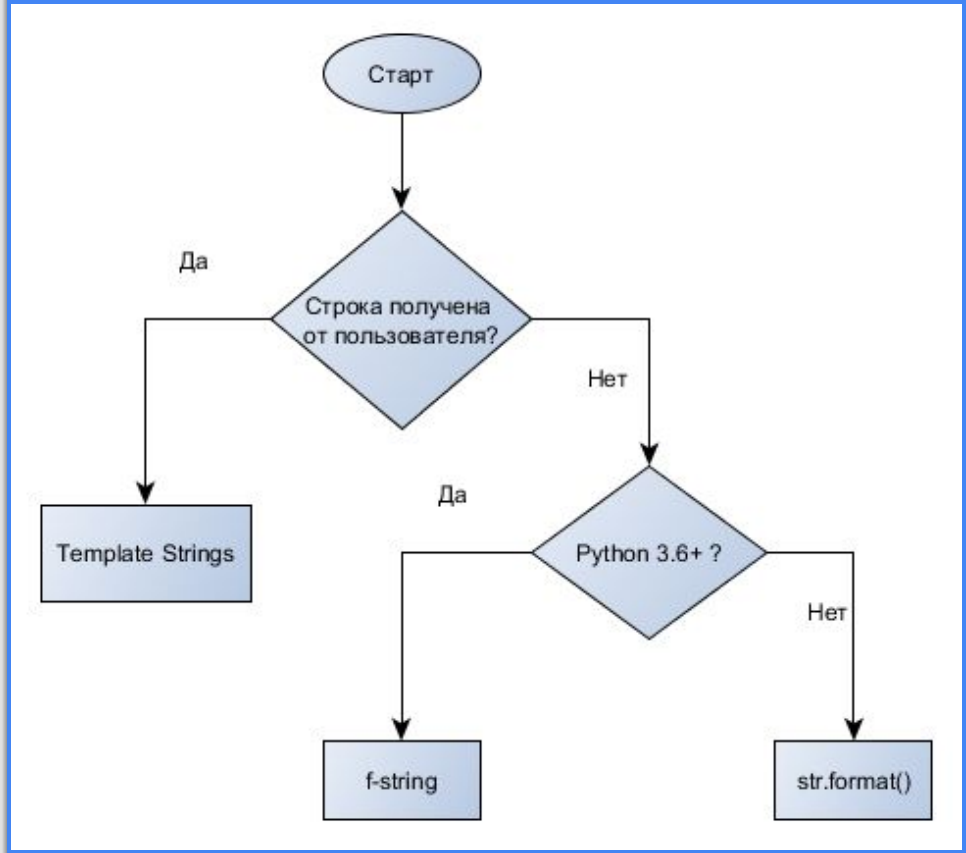
```
Hey, Eric Idle!
```

Нужно импортировать класс Template из встроенного строкового модуля Python. Шаблонные строки не являются основной функцией языка, но они предоставляются модулем строк в стандартной библиотеке.

https://docs.python.org/3/library/string.html

Каким методом форматирования строк стоит пользоваться?

Если строки получены от пользователя, используйте шаблонные строки (способ 4), чтобы избежать проблем с уязвимостью программы.



Другое отличие состоит в том, что строки шаблона не допускают спецификаторов формата.

```
templ_string = 'Hey $name, there is a $error error!'

s = Template(templ_string).substitute(name=name,
error=hex(errno))

print(s)
```

```
Hey Eric Idle, there is a 0xbadc0ffee error!
```

Отступы и выравнивание строк

Добавить отступы слева:

```
test = 'test'
print('%10s' % (test,))
print('{:>10}'.format(test))
print(f'{test:>10}')
```

```
      test
      test
      test
```

Добавить отступы справа:

```
test = 'test'
print('{:_<10}'.format(test))
print(f'{test:_<10}')

test______
test______
```

Выравнивание по центру:

Если при выравнивании по центру получается нечетное количество отступов, то нечетный будет добавлен справа.

```
test = 'test'
print('{:_^10}'.format(test))
print(f'{test:_^10}')
___test___
___test___

print('{:_^6}'.format('zip'))
_zip__
```

Усечение длинных строк

```
print('%.5s' % ('xylophone',))
print('{:.5}'.format('xylophone'))
print(f'{"xylophone":.5}')

xylop
xylop
xylop

print(f'"{"xylophone":10.5}"')
"xylop     "
```

Целые числа:

```
print('%d' % (42,))
print('{:d}'.format(42))
print(f'{42:d}')

42
42
42
```

Целые числа:

```
print(f'Показывать знак, даже если число положительное: {42:_>+6d}')
print(f'{-42:_>+54d}')
```

```
Показывать знак, даже если число положительное: ___+42
______________________________________________________-42
```

Floats:

```
print('%f' % (3.141592653589793,))
print('{:f}'.format(3.141592653589793))
print(f'{3.141592653589793:f}')

3.141593
3.141593
3.141593
```

```
pi = 3.141592653589793
print('%06.2f' % (pi))
print('{:06.2f}'.format(pi))
print(f'{pi:06.2f}')

003.14
003.14
003.14
```

```
print(f'{42:_^+10.2f}')
```

```
__+42.00__
```

Пробел означает, что для отрицательного значения будет отображен минус, а для положительного пробел.

```
print('{: d}'.format((- 23)))
print('{: d}'.format((23)))

-23
 23
```

```
print(f'{-23: d}')
print(f'{23: d}')

-23
 23
```

```
points = 19.5

total = 22

print('Correct answers: {:.2%}'.format(points/total))

print(f'Correct answers: {points/total:.2%}')
```

```
Correct answers: 88.64%
Correct answers: 88.64%
```

Datetime:

```
from datetime import datetime

s = '{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5))

print(s)
```

```
2001-02-03 04:05
```

Datetime (f-строки):

```
from datetime import datetime

dt = datetime(2001, 2, 3, 4, 5)
s = f'{dt:%Y-%m-%d %H:%M}'

print(s)
```

```
2001-02-03 04:05
```

Встроенные методы строк в python

string.capitalize()

приводит первую букву в верхний регистр, остальные в нижний.

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.

Возвращает копию s со всеми буквенными символами, преобразованными в нижний регистр:

```
s = 'everyTHing yoU Can IMaGine is rEAl'
s.lower()
```

```
'everything you can imagine is real'
```

string.swapcase()

возвращает копию s с заглавными буквенными символами, преобразованными в строчные и наоборот:

```
s = 'everyTHing yoU Can IMaGine is rEAl'
s.swapcase()
```

```
'EVERYthING YOu cAN imAgINE IS ReaL'
```

string.title()

преобразует первые буквы всех слов в заглавные.

возвращает копию, s в которой первая буква каждого слова преобразуется в верхний регистр, а остальные буквы — в нижний регистр:

```
s = 'the sun also rises'
s.title()
```

'The Sun Also Rises'

string.upper()

преобразует все буквенные символы в заглавные.

Возвращает копию s со всеми буквенными символами в верхнем регистре:

```
s = 'follow us @PYTHON'
s.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.

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.

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
```

Например, метод endswith() можно использовать для проверки окончания файла.

```
filenames = ["file.txt", "image.jpg", "str.txt"]

for fn in filenames:
    if fn.lower().endswith(".txt"):
        print(fn)
```

```
file.txt
str.txt
```

string.find(<sub>[, <start>[, <end>]])

ищет в строке заданную подстроку.

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.

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.

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.

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.

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).

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.

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
```

string.islower()

определяет, являются ли буквенные символы строки строчными.

возвращает True, если строка s не пустая, и все содержащиеся в нем буквенные символы строчные, а False если нет. Неалфавитные символы игнорируются:

```
>>> 'abc'.islower()
True
>>> 'abc1$d'.islower()
True
>>> 'Abc1$D'.islower()
False
```

string.isprintable()

определяет, состоит ли строка только из печатаемых символов.

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
```

string.isspace()

определяет, состоит ли строка только из пробельных символов.

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

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

string.istitle()

определяет, начинаются ли слова строки с заглавной буквы.

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:

```
>>> ' \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.
```

Выравнивание строк, отступы

Методы в этой группе влияют на вывод строки.

string.center(<width>[, <fill>])

выравнивает строку по центру.

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.lstrip([<chars>])

обрезает пробельные символы слева.

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.

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.

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

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.

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'
```

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.

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.

Многие из этих методов возвращают либо список, либо кортеж.

Список заключен в квадратные скобки ([ ]), а кортеж заключен в простые (( )).

string.join(<iterable>)

объединяет список в строку.

Возвращает строку, которая является результатом конкатенации объекта <iterable> с разделителем s.

.join() вызывается строка-разделитель s . <iterable> должна быть последовательностью строковых объектов.

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

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

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

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

string.partition(<sep>)

делит строку на основе разделителя.

Отделяет от s подстроку длиной от начала до первого вхождения <sep>. Возвращаемое значение представляет собой кортеж из трех частей:

```
>>> '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.

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 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']
```

string.rsplit(sep=None, maxsplit=-1)

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']
```

string.rsplit(sep=None, maxsplit=-1)

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

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

string.rsplit(sep=None, maxsplit=-1)

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.splitlines([<keep ends>])

делит текст на список строк.

Любой из следующих символов или последовательностей символов считается границей строки:

\n, \r, \r\n,\v,\x0b,\f, \x0c, \x1c,\x1d,\x1e, \x85, \u2028, \u2029

```
>>> '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']
```

string.splitlines([<keepends>])

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']
```

Заключение

Было подробно рассмотрено множество различных механизмов, которые Python предоставляет для работы со строками, включая операторы, встроенные функции и встроенные методы.

Рассмотренных стандартных возможностей для работы с текстом достаточно далеко не всегда. Например, в методах find() и replace() задается всего одна строка. В реальных задачах такая однозначность встречается довольно редко, чаще требуется найти или заменить строки, отвечающие некоторому шаблону.

Такую возможность предоставляют регулярные выражения, работа с которыми будет рассмотрена далее.