Лекция 4: Строки, байты, файлы и ввод/вывод

Сергей Лебедев

sergei.a.lebedev@gmail.com

28 сентября 2015 г.

Строки

Строковые литералы

- Для объявления строки можно использовать двойные или одинарные кавычки:

```
>>> "foobar" == 'foobar'
True
```

- Зачем так много?

```
>>> '"Where art thou?"'
'"Where art thou?"'
>>> "I'm here!"
"I'm here!"
```

- Для многострочных блоков текста используют тройные кавычки:

```
>>> """foo
... bar"""
'foo\nbar'
```

```
>>> '''foo
... bar'''
'foo\nbar'
```

- Подряд идущие строковые литералы “склеиваются”:

```
>>> "foo" "bar"
'foobar'
```

Экранированные последовательности и “сырые” строки

- Строковые литералы могут содержать экранированные последовательности, например[1]:

`\'`	одинарная кавычка,
`\"`	двойная кавычка,
`\t`	символ вертикальной табуляции,
`\n`	символ переноса строки,
`\xhh`	символ с HEX кодом `hh`.

- В “сырых” строковых литералах экранированные последовательности не обрабатываются.

```
>>> print("\tell me")
    ell me
>>> print(r"\tell me")
\tell me
```

[1] Полный список поддерживаемых последовательностей можно прочитать в документации http://bit.ly/escape-sequences

USASCII code chart

$b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits ↓ / Column →	0	1	2	3	4	5	6	7
b_4 b_3 b_2 b_1 / Row ↓								
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

[2] http://en.wikipedia.org/wiki/ASCII

- Юникод — стандарт кодирования текста на разных языках.
- В стандарте версии 8.0 содержится более 120 000 символов.
- Фактически Юникод — это отображение, сопоставляющее символу[3] уникальный номер.
- Как закодировать всё это множество символов байтами?
  - Unicode Transformation Format, по N бит на каждый символ: UTF-8, UTF-16, UTF-32.
  - Universal Character Set, по N байт на каждый символ: UCS-2, UCS-4.
- Если кодировка использует более одного байта, то закодированный текст может также содержать BOM — маркер последовательности байтов, U+FEFF.

[3] Более формально — *codepoint*.

- Тип `str` — неизменяемая последовательность символов Юникод:

```
>>> s = "я строка"
>>> list(s)
['я', ' ', 'с', 'т', 'р', 'о', 'к', 'а']
```

- Отдельного типа для символов в Python нет: каждый символ строки — тоже строка:

```
>>> s[0], type(s[0])
('я', <class 'str'>)
>>> list(s[0])
['я']
```

- Как строки представляются в памяти?

  `UTF-8` `UTF-16` `UTF-32` `UCS-2` `UCS-4`

- Начиная с версии 3.3 в Python был реализован PEP-393[4], который описывает механизм адаптивного представления строк.
  - Если строка состоит из символов ASCII, то она хранится в кодировке UCS-1, то есть каждый символ представляется одним байтом.
  - Если максимальный код символа в строке меньше 2^{16}, то используется UCS-2.
  - Иначе используется UCS-4, кодирующая каждый символ четырьмя байтами.
- Несколько примеров:

```
>>> list(map(ord, "hello"))   # UCS-1
[104, 101, 108, 108, 111]
>>> list(map(ord, "привет"))  # UCS-2
[1087, 1088, 1080, 1074, 1077, 1090]
>>> ord("")                   # UCS-4, почти
127025
```

[4] http://python.org/dev/peps/pep-0393

- Python поддерживает экранированные последовательности для символов Юникода:

```
>>> "\u0068", "\U00000068"
('h', 'h')
>>> "\N{DOMINO TILE HORIZONTAL-00-00}"
''
```

- Получить символ Юникода по его коду можно с помощью функции `chr`:

```
>>> chr(0x68)
'h'
>>> chr(1087)
'п'
```

- Очевидное наблюдение:

```
>>> def identity(ch):
...     return chr(ord(ch))
...
>>> identity("п")
'п'
```

Методы работы со строками: модификаторы регистра

```
>>> "foo bar".capitalize()
'Foo bar'
>>> "foo bar".title()
'Foo Bar'
>>> "foo bar".upper()
'FOO BAR'
>>> "foo bar".lower()
'foo bar'
>>> "foo bar".title().swapcase()
'fOO bAR'
```

Методы работы со строками: выравнивание

- Группа методов, выравнивающих строку в “блоке” фиксированной длины. При этом дополнительные позиции заполняются указанным символом:

```
>>> "foo bar".ljust(16, '~')
'foo bar~~~~~~~~~'
>>> "foo bar".rjust(16, '~')
'~~~~~~~~~foo bar'
>>> "foo bar".center(16, '~')
'~~~~foo bar~~~~~'
```

- В качестве символа по умолчанию используется пробел:

```
>>> "foo bar".ljust(16)
'foo bar         '
>>> "foo bar".rjust(16)
'         foo bar'
>>> "foo bar".center(16)
'    foo bar     '
```

- Если длина “блока” меньше длины строки, то строка возвращается без изменений.

Методы работы со строками: удаление символов

- Группа методов, заканчивающихся на `strip`, удаляет все вхождения указанных символов слева, справа или с обоих концов строки:

```
>>> "]>>foo bar<<[".lstrip("]>")
'foo bar<<['
>>> "]>>foo bar<<[".rstrip("[<")
']>>foo bar'
>>> "]>>foo bar<<[".strip("[]<>")
'foo bar'
```

- По умолчанию удаляются все пробелы:

```
>>> "\t  foo bar \r\n  ".strip()
'foo bar'
```

Методы работы со строками: разделение

- Метод `split` разделяет строку на подстроки по указанному разделителю:

```
>>> "foo,bar".split(",")
['foo', 'bar']
>>> "foo,,,bar".split(",")
['foo', '', '', 'bar']
```

- Если разделитель не указан, то строка разделяется по пробелам.

```
>>> "\t  foo bar \r\n  ".split()
['foo', 'bar']
```

- Метод `partition` возвращает кортеж из трёх элементов: подстрока до вхождения разделителя, сам разделитель и подстрока после вхождения разделителя.

```
>>> "foo,bar,baz".partition(",")
('foo', ',', 'bar,baz')
>>> "foo,bar,baz".rpartition(",")
('foo,bar', ',', 'baz')
```

- С помощью метода `join` можно соединить любую последовательность строк:

```
>>> ", ".join(["foo", "bar", "baz"])
'foo, bar, baz'
>>> ", ".join(filter(None, ["", "foo"]))
'foo'
>>> ", ".join("bar")
'b, a, r'
```

- Если последовательность содержит объекты другого типа, будет ошибка:

```
>>> ", ".join(range(10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance [...]
```

Методы работы со строками: проверка на вхождение

- Вхождение подстроки идиоматично проверять с помощью операторов `in` и `not in`:

```
>>> "foo" in "foobar"
True
>>> "yada" not in "foobar"
True
```

- Также можно сравнивать префикс или суффикс строки с одной или несколькими строками:

```
>>> "foobar".startswith("foo")
True
>>> "foobar".endswith(("boo", "bar"))
True
```

- Найти место первого вхождения подстроки можно с помощью метода `find`:

```
>>> "abracadabra".find("ra")
2
>>> "abracadabra".find("ra", 0, 3)
-1
```

- Метод `index` аналогичен `find`, но, если искомая подстрока не найдена, он поднимает исключение:

```
>>> "abracadabra".index("ra", 0, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

- Для поиска места **последнего** вхождения подстроки можно воспользоваться методами `rfind` и `rindex`.

Методы работы со строками: замена подстроки

- Метод `replace` заменяет вхождения подстроки на заданную строку, по умолчанию — все:

```
>>> "abracadabra".replace("ra", "**")
'ab**cadab**'
>>> "abracadabra".replace("ra", "**", 1)
'ab**cadabra'
```

- Для множественной замены символов удобно использовать метод `translate`, который принимает словарь трансляции.

```
>>> translation_map = {ord("a"): "*", ord("b"): "?"}
>>> "abracadabra".translate(tranlation_map)
'*?r*c*d*?r*'
```

- Методы-предикаты позволяют проверить строку на соответствие некоторому формату, например:

```
>>> "100500".isdigit()
True
>>> "foo100500".isalnum()
True
>>> "foobar".isalpha()
True
```

- Другие полезные предикаты:

```
>>> "foobar".islower()
True
>>> "FOOBAR".isupper()
True
>>> "Foo Bar".istitle()
True
>>> "\r   \n\t  \r\n".isspace()
True
```

Форматирование строк: метод `format`

- В Python есть два способа форматирования строк. Первый, который мы рассмотрим, использует метод `format`:

```
>>> "{}, {}, how are you?".format("Hello", "Sally")
'Hello, Sally, how are you?'
>>> "Today is October, {}th.".format(8)
'Today is October, 8th.'
```

- `{}` обозначает место, в которое будет подставлен позиционный аргумент.
- Внутри `{}` можно опционально указать способ преобразования объекта в строку и спецификацию формата.

Преобразование объекта в строку

- В Python 3 есть три различных по смыслу способа преобразовать объект в строку:
  - `str` возвращает человекочитаемое представление объекта,
  - `repr` возвращает представление объекта, по которому можно однозначно восстановить его значение,
  - `ascii` аналогичен `repr` по смыслу, но возвращаемая строка состоит только из символов ASCII.
- Примеры:

```
>>> str("я строка")
'я строка'
>>> repr("я строка")
"'я строка'"
>>> ascii("я строка")
"'\\u044f \\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
```

- Напоминание: внутри {} можно опционально указать способ преобразования объекта в строку и спецификацию формата.
- Для преобразования объекта в строку используются первые буквы соответствующих функций:

```
>>> "{!s}".format("я строка")  # str
'я строка'
>>> "{!r}".format("я строка")  # repr
"'я строка'"
>>> "{!a}".format("я строка")  # ascii
"'\\u044f \\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
```

- Зачем нужна спецификация формата?

Форматирование строк: спецификация формата

- Спецификация формата позволяет:
  - выровнять строку в “блоке” фиксированной длины,

```
>>> "{:~^16}".format("foo bar")
'~~~~foo bar~~~~~'
```

  - привести число к другой системе счисления,

```
>>> "int: {0:d}  hex: {0:x}".format(42)
'int: 42  hex: 2a'
>>> "oct: {0:o}  bin: {0:b}".format(42)
'oct: 52  bin: 101010'
```

  - потребовать наличие знака в строковом представлении числа и зафиксировать количество знаков до или после запятой.

```
>>> "{:+08.2f}".format(-42.42)
'-0042.42'
```

- Комбинированный пример:

```
>>> "{!r:~^16}".format("foo bar")
"~~~'foo bar'~~~~"
```

Форматирование строк: позиционные и ключевые аргументы

- Внутри {} можно также явно указывать номер позиционного или имя ключевого аргумента:

```
>>> "{0}, {1}, {0}".format("hello", "kitty")
'hello, kitty, hello'
>>> "{0}, {who}, {0}".format("hello", who="kitty")
'hello, kitty, hello'
```

- Если один из аргументов — контейнер, то при форматировании можно обращаться к его элементам по индексу или ключу:

```
>>> point = 0, 10
>>> "x = {0[0]}, y = {0[1]}".format(point)
'x = 0, y = 10'
>>> point = {"x": 0, "y": 10}
>>> "x = {0[x]}, y = {0[y]}".format(point)
'x = 0, y = 10'
```

- Ещё один способ форматирования строк в Python использует оператор `%` и внешне похож на `printf`:

```
>>> "%s, %s, how are you?" % ("Hello", "Sally")
'Hello, Sally, how are you?'
>>> point = {"x": 0, "y": 10}
>>> "x = %(x)+2d, y = %(y)+2d" % point
'x = +0, y = +10'
```

- Он менее выразителен и гибок, чем `format`:
  - `%` — бинарный оператор, справа от него может быть один аргумент: кортеж или словарь,
  - каждый элемент кортежа используется только один раз,
  - нет синтаксиса для обращения к элементам контейнера или атрибутам объекта,
  - не предусмотрена возможность расширения, например, если потребуется форматировать длинные числа с плавающей точкой, то синтаксис `"%8.2f"` для них работать не будет.

- В Python есть два механизма форматирования строк:
  - с использованием метода `format` и
  - с помощью оператора `%`.
- Метод `format` объективно более лучше, поэтому рекомендуется использовать его.
- Если у вас остались сомнения, то попробуйте предсказать, что будет выведено в результате:

```
>>> "I'm a list with three arguments: %s" % [1, 2, 3]
???
>>> "I'm a tuple with three arguments: %s" % (1, 2, 3)
???
>>> "I'm a string with three characters: %s" % "abc"
???
```

- В модуле string можно найти полезные при работе со строками константы:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.punctuation
'!"#%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

- Остальное содержимое модуля string выходит за рамки сегодняшней лекции. Не стесняйтесь обращаться к документации[5], если вам интересно.

[5] http://docs.python.org/3/library/string

- Строки в Python 3 — это последовательности символов (правильнее *codepoints*) Юникод.
- В отличие от C/C++ или Java:
  - нет разницы между одинарными и двойными кавычками,
  - символ строки — это тоже строка,
  - можно использовать тройные кавычки для многострочных блоков текста.
- Обилие методов у класса `str` позволяет не изобретать велосипед при работе с текстовыми данными.
- Для форматирования строк следует использовать метод `format`, но полезно знать также про синтаксис с использованием оператора `%`.

Байты

- Тип `bytes` — неизменяемая последовательность байтов.
- Аналогично строковым литералам в Python есть литералы для байтов и “сырых” байтов:

```
>>> b"\00\42\24\00"
b'\x00"\x14\x00'
>>> rb"\00\42\24\00"
b'\\00\\42\\24\\00'
```

- Байты и строки тесно связаны:
  - строку можно закодировать последовательностью байтов

```
>>> chunk = "я строка".encode("utf-8")
>>> chunk
b'\xd1\x8f \xd1\x81\xd1\x82\xd1\x80\xd0[...]'
```

  - и из последовательности байтов можно раскодировать строку

```
>>> chunk.decode("utf-8")
'я строка'
```

- Напоминание: `"utf-8"` — одна из кодировок Юникода.

- Кодировка специфицирует преобразование текстовой строки в последовательность байтов. Стандартная библиотека Python поддерживает более сотни кодировок.
- Любители Windows наверняка узнают магическое сочетание букв в следующем примере:

```
>>> chunk = "я строка".encode("cp1251")
>>> chunk
b'\xff \xf1\xf2\xf0\xee\xea\xe0'
```

- Что будет, если указать неверную кодировку?

```
>>> chunk.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode [...]
```

- Методы `encode` и `decode` принимают опциональный аргумент, контролирующий логику обработки ошибок:
  - `"strict"` — ошибочные символы поднимают исключение,
  - `"ignore"` — ошибочные символы пропускаются,
  - `"replace"` — ошибочные символы заменяются на `"\ufffd"`.
- Пример:

```
>>> chunk = "я строка".encode("cp1251")
>>> chunk.decode("utf-8", "ignore")
' '
>>> chunk.decode("utf-8", "replace")
'? ??????'  # не совсем так
```

- Если не указывать кодировку, то Python будет использовать системную кодировку по умолчанию:

```
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'  # на Linux и OS X
```

- Байты поддерживают большинство операций, специфичных для строк, за исключением:
  - метода `encode`, кодирующего строку в байты,
  - метода `format`, форматирующего строку для вывода,
  - некоторых предикатов, о которых мы не говорили.
- Также стоит помнить, что тип `bytes` соответствует последовательности байтов, а не символов, то есть:

```
>>> "boo" in b"foobar"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Type str doesn't support the buffer API
>>> b"foobar".replace("o", "")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected bytes, bytearray or [...]
```

- Байты в Python 3 — это специальный тип для последовательности байтов.
- Байты — это не строки, а строки — это не байты, но:
  - строку можно кодировать байтами,
  - а байты можно декодировать в строку.
- Большинство операций, поддерживаемых строками, реализованы и для байтов:

```
>>> b"foo bar".title().center(16, b"~")
b'~~~~Foo Bar~~~~~'
```

Файлы и ввод/вывод

- Текстовые и бинарные файлы в Python — это две большие разницы.
- Создать объект типа файл можно с помощью функции `open`, которая принимает один обязательный аргумент — путь к файлу:

```
>>> open("./HBA1.txt")
<[...] name='./HBA1.txt' mode='r' encoding='UTF-8'>
```

- Аргументов у функции `open` довольно много, нас будут интересовать:
  - `mode` — определяет, в каком режиме будет открыт файл, возможные значения:
    - `"r"`, `"w"`, `"x"`, `"a"`, `"+"`,
    - `"b"`, `"t"`.
  - для текстовых файлов можно также указать `encoding` и `errors`.

Примеры использования функции open

- Открыть бинарный файл для чтения и записи:

```
>>> open("./csc.db", "r+b")
<_io.BufferedRandom name='./csc.db'>
```

- Открыть текстовый в файл в кодировке `"cp1251"` для добавления, игнорируя ошибки кодирования:

```
>>> open("./admin.log", "a", encoding="cp1251",
...      errors="ignore")
<[...] name='./admin.log' mode='a' encoding='cp1251'>
```

- Создать новый текстовый файл в системной кодировке и открыть его для записи:

```
>>> open("./lecture4.tex", "x")
<[...] name='./lecture4.tex' mode='x' encoding='UTF-8'>
```

- Метод `read` читает не более, чем `n` символов из файла:

```
>>> handle = open("./HBA1.txt")
>>> handle.read(16)
'MVLSGEDKSNIKAAWG'
```

- Методы `readline` и `readlines` читают одну или все строчки соотвественно. Можно указать максимальное количество символов, которые надо прочитать:

```
>>> handle = open("./HBA1.txt")
>>> len(handle.readline())
143
>>> handle.readline(16)
'MVLSGEDKSNIKAAWG'
>>> handle.readlines(16)
['KIGGHGAEYGAEALERMFASFPTTKTYFPHFDVSHGSAQVKGHGKKVADA'
 'LANAAGHLDDLPGALSALSDLHAHKLRVDPVNFKLLSHCLLVTLASHHPA'
 'DFTPAVHASLDKFLASVSTVLTSKYR\n']  # \n!
```

[6] Все примеры используют текстовый файл. Семантика методов в случае бинарного файла аналогична.

- Метод `write` записывает строку в файл:

```
>>> handle = open("./example.txt", "w")
>>> handle.write("abracadabra")
11  # количество записанных байт.
```

  Неявного добавления символа окончания строки при этом не происходит.

- Записать последовательность строк можно с помощью метода `writelines`:

```
>>> handle.writelines(["foo", "bar"])
```

Методы работы с файлами: другие

```
>>> handle = open("./example.txt", "r+")
>>> handle.fileno()
3
>>> handle.tell()
0
>>> handle.seek(8)
>>> handle.tell()
8
>>> handle.write("something unimportant")
>>> handle.flush()
>>> handle.close()
```

- Интерпретатор Python предоставляет три **текстовых** файла, называемых стандартными потоками ввода/вывода: `sys.stdin`, `sys.stdout` и `sys.stderr`.
- Для чтения `sys.stdin` используют функцию `input`:

```
>>> input("Name: ")
Name: Pumpkin
'Pumpkin'
```

```
>>> input()
42
'42'
```

- Для записи в `sys.stdout` или `sys.stderr` — функцию `print`:

```
>>> print("Hello, `sys.stdout`!", file=sys.stdout)
Hello, `sys.stdout`!
>>> print("Hello, `sys.stderr`!", file=sys.stderr)
Hello, `sys.stderr`!
```

- Функция print позволяет изменять разделитель между аргументами:

```
>>> print(*range(4))
0 1 2 3
>>> print(*range(4), sep="_")
0_1_2_3
```

- указывать последовательность, которой заканчивается вывод:

```
>>> print(*range(4), end="\n--\n")
0 1 2 3
--
```

- и форсировать вызов flush у файла, в который осуществляется вывод:

```
>>> handle = open("./example.txt", "w")
>>> print(*range(4), file=handle, flush=True)
```

Модуль io с высоты птичьего полёта

- В модуле `io` реализованы базовые классы для работы с текстовыми и бинарными данными.
- Класс `io.StringIO` позволяет получить файловый объект из строки, а `io.BytesIO` из байтов:

```
>>> import io
>>> handle = io.StringIO("foo\n\bar")
>>> handle.readline()
'foo\n'
>>> handle.write("boo")
>>> handle.getvalue()
'foo\nboo'
```

- Аналогичный пример для `io.BytesIO`:

```
>>> handle = io.BytesIO(b"foobar")
>>> handle.read(3)
b'foo'
>>> handle.getvalue()
b'foobar'
```

- Файлы бывают текстовые и бинарные, их можно читать и в них можно писать.
- Методы работы с файлами глобально не отличаются от других языков:

```
>>> handle = open("./example.txt", "w")
>>> handle.write("foobar")
6
>>> handle.close()
```

- Стандартные потоки ввода/вывода в Python тоже представляются в виде **текстовых** файлов. Для работы с ними удобно использовать функции `input` и `print`.
- В модуле `io` реализована иерархия низкоуровневых классов для работы с вводом/выводом. Полезные для обычных людей классы: `io.StringIO` и `io.BytesIO`.