Теоретический материал к занятию Строки и их методы

Строка – это сложный тип данных, представляющий собой последовательность символов. Строки в языке программирования Python могут заключаться как в одиночные, так и в двойные кавычки. Однако, начало и конец строки должны обрамляться одинаковым типом кавычек.

```
>>> a = 'Python'
>>> b = 'эквилибристика'
```

1. Строковый тип данных str() и индексация символов в строке.

В Python строковый тип данных имеет название str (сокращение от string — струна, ряд).

В последовательностях важен порядок символов, у каждого символа в строке есть уникальный порядковый номер – индекс. Можно обращаться к конкретному символу в строке и извлекать его с помощью оператора индексирования, который представляет собой квадратные скобки с номером символа в них. Например,

```
>>> a = 'программирование'
>>> a[2]
'о'
```

☞*Обратите внимание:*

символы в строке нумеруются с 0, а в обратном порядке – с -1.

П	Р	И	В	Е	Т
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Индекс может быть и отрицательным, тогда отсчет символов ведется с конца строки.

```
>>> a = 'ПРИВЕТ'
>>> a[-3]
'В'
```

2. Функции len() и str()

Существует специальная функция len(), позволяющая измерить длину строки. Результатом выполнения данной функции является число, показывающее количество символов в строке. Например,

```
>>> print(len('эквилибристика'))
14
```

☞*Обратите внимание:*

При подсчете длины строки считаются все символы, включая пробелы.

Иногда работать со строками намного проще, чем с числами. Даже если в условии задачи сказано, что дается число, нам ничто не мешает работать с ним как со строкой. Для этого используется функция str(). Например,

```
num1 = 2024 # целое число
num2 = 20.24 # число с плавающей точкой
s1 = str(num1) # преобразовали целое число в строку '2024'
s2 = str(num2) # преобразовали число с плавающей точкой в строку '20.24'
```

3. Действия со строками

a. Конкатенация строк

Для строк существуют операции конкатенации (+), которая обеспечивает объединение («склеивание») двух или более строк в одну строку. Например,

```
>>> a = 'Почему '
>>> b = 'коровы '
>>> c = ' не летают?'
>>> print(a + b + c)
Почему коровы  не летают?
```

☞*Обратите внимание:*

Операция сложения строк в отличие от операции сложения чисел не является коммутативной, то есть, от перестановки мест слагаемых-строк результат меняется!

b. Умножение строки на число

Достаточно необычна операция дублирования (*), которая указывает на число повторений символов в одной строке. Например,

```
>>> a = 'Нас не догонят! '
>>> b = a * 3
>>> b
'Нас не догонят! Нас не догонят! Нас не догонят! '
```

☞*Обратите внимание:*

Строку можно умножать на число, но нельзя умножать на строку.

Примечания

Примечание 1. Тройные кавычки в Python используются для многострочного (multiline) текста. Например,

```
>>> text = '''Python is an interpreted, high-level, general-purpose programming lang
    uage. Created by Guido van Rossum and first released in 1991, Python design
... philosophy emphasizes code readability with its notable use of significant white
    space.'''
```

Примечание 2. На первый взгляд может показаться странным, что можно использовать как одинарные, так и двойные кавычки, однако такой подход позволяет очень легко добавлять в строку нужные кавычки:

```
>>> s1 = 'Мы можем использовать в одиночных кавычках двойные кавычки "Война и мир"'
... s2 = "Мы можем использовать в двойных кавычках одиночные кавычки 'Война и мир'"
... print(s1)
... print(s2)
```

Результатом выполнения такого кода будет:

Мы можем использовать в одиночных кавычках двойные кавычки "Война и мир"

Мы можем использовать в двойных кавычках одиночные кавычки 'Война и мир'/

c. Оператор in

В Python есть специальный оператор in, который позволяет проверить, что одна строка находится внутри другой. Тем самым, оператор in определяет, является ли одна строка подстрокой другой.

```
s = 'программирование'
if 'а' in s:
   print('Введенная строка содержит символ а')
else:
   print('Введенная строка не содержит символ а')
```

d. Срезы строк

Можно из строки извлекать не один символ, а несколько, т.е. получать срез (подстроку). Оператор извлечения среза из строки выглядит так: [X : Y].

X – это индекс начала среза, а Y – окончание среза, причем символ с номером Y в срез уже не входит. Если отсутствует первый индекс, то срез берется от начала до второго индекса; при отсутствии второго индекса, срез берется от первого индекса до конца строки. Например,

```
>>> x = 'Python'
>>> x[0:3]
'Pyt'
>>> x[:3]
'Pyt'
>>> x[-4:]
'thon'
>>> x[:]
'Python'
```

Кроме того, можно извлекать символы не подряд, а через определенное количество символов. В таком случае оператор индексирования выглядит так: [X : Y : Z];

Z – это шаг, через который осуществляется выбор элементов. Например,

```
>>> x = 'Престиж'
>>> x[0:7:2]
'Петж'
>>> x[::2]
'Петж'
>>> x[::3]
'Псж'
```

Строки в языке Python непосредственным присваиванием невозможно изменить. Попытка изменить символ в определенной позиции или подстроку вызовет ошибку:

```
>>> a = 'Pyhthon'
>>> a[2]= ''
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a[2]= ''
TypeError: 'str' object does not support item assignment
>>>
```

4. Методы работы со строками

Метод — это специализированная функция. Как и функция, он вызывается для выполнения отдельной задачи, но метод вызывается для определенного объекта, к которому она обращается по имени. Метод вызывается командой вида **имя_объекта.имя_метода (параметры).**

В Python все методы работы со строками можно разделить на 4 группы:

1) Преобразование строк
2) Классификация строк
3) Конвертация регистра
4) Поиск, подсчет и замена символов

Для решения задач будем использовать один из методов, приведенных в таблице:

Функция или метод	*Назначение*
str in S	Проверка на вхождение подстроки в строку
S.find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
S.rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
S.replace(шаблон, замена)	Замена
S.isdigit()	Состоит ли строка из цифр
S.isalpha()	Состоит ли строка из букв
S.isalnum()	Состоит ли строка из цифр или букв
S.islower()	Состоит ли строка из символов в нижнем регистре
S.isupper()	Состоит ли строка из символов в верхнем регистре
S.istitle()	Начинаются ли слова в строке с заглавной буквы
S.upper()	Преобразование строки к верхнему регистру
S.lower()	Преобразование строки к нижнему регистру
S.join(список)	Сборка строки из списка с разделителем S
S.split(символ)	Разбиение строки по разделителю
ord(символ)	Символ в его код ASCII
chr(число)	Код ASCII в символ
S.capitalize()	Переводит первый символ строки в верхний регистр, а все остальные - в нижний
S.strip([chars])	Удаление пробельных символов в начале и в конце строки
S.swapcase()	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
S.title()	Первую букву каждого слова переводит в верхний регистр, а все остальные - в нижний

Рассмотрим применение этих методов в задачах:

1) Преобразование строк

Три самых используемых метода из этой группы – **join(), split() и partition().** Метод **join()** незаменим, если нужно преобразовать список или кортеж в строку:

```
>>> spisok = ['Я ', 'изучаю ', 'Python ']
>>> stroka = ''.join(spisok)
>>> print(stroka)
Я изучаю Python
```

При объединении списка или кортежа в строку можно использовать любые разделители:

Метод **split()** используется для обратной манипуляции – преобразования строки в список:

```
>>> text = 'Овен Телец Близнецы Рак Лев Дева Весы Скорпион Стрелец Козерог Водолей Рыбы '
>>> spisok = text.split()
>>> print(spisok)
['Овен', 'Телец', 'Близнецы', 'Рак', 'Лев', 'Дева', 'Весы', 'Скорпион', 'Стрелец', 'Козерог', 'Водолей', 'Рыбы']
```

По умолчанию **split()** разбивает строку по пробелам.

```
>>> color = 'каждый, охотник, желает, знать, где, сидит, фазан'
>>> spisok = color.split(',')
>>> print(spisok)
['каждый', ' охотник', ' желает', ' знать', ' где', ' сидит', ' фазан']
```

Но можно указать любой другой символ – и на практике это часто требуется:

```
>>> kort = ['Я ', 'изучаю ', 'Django ']
>>> stroka = '***'.join(kort)
>>> print(stroka)
Я ***изучаю ***Django
```

Метод **partition()** поможет преобразовать строку в кортеж:

```
>>> text = 'Python - простой и понятный язык'
>>> kort = text.partition('и')
>>> print(kort)
('Python - простой ', 'и', ' понятный язык')
```

В отличие от **split(), partition()** учитывает только первое вхождение элемента-разделителя (и добавляет его в итоговый кортеж).

2) Оценка и классификация строк

В Python много встроенных методов для оценки и классификации текстовых данных. Некоторые из этих методов работают только со строками, в то время как другие универсальны. К последним относятся, например, функции **min()** и **max():**

```
>>> text = '12345'
>>> print(max(text))
5
>>> print(min(text))
1
```

В Python есть специальные методы для определения типа символов. Например, isalnum() оценивает, состоит ли строка из букв и цифр, либо в ней есть какие-то другие символы:

```
>>> text = 'abracadabra123456'
>>> print(text.isalnum())
True
>>> text1 = '125 $'
>>> print(text1.isalnum())
False
```

Метод **isalpha()** поможет определить, состоит ли строка только из букв, или включает специальные символы, пробелы и цифры:

```
>>> text2 = 'password123'
>>> print(text.isdigit())
False
>>>
>>> text = 'программирование'
>>> print(text.isalpha())
True
```

С помощью метода isdigit() можно определить, входят ли в строку только цифры, или там есть и другие символы:

```
>>> text = 'программирование'
>>> print(text.isdigit())
False
```

Методы **islower()** и **isupper()** определяют регистр, в котором находятся буквы. Эти методы игнорируют небуквенные символы:

```
>>> text = 'abracadabra'
>>> print(text.islower())
True
>>> text2 = 'Python bytes'
>>> print(text2.islower())
False
>>> text3 = 'PYTHON'
>>> print(text3.isupper())
True
```

Метод **isspace()** определяет, состоит ли анализируемая строка из одних пробелов, или содержит что-нибудь еще:

```
>>> stroka = '   '
>>> print(stroka.isspace())
True
>>> stroka2 = ' a '
>>> print(stroka2.isspace())
False
```

3) Конвертация регистра

Строки относятся к неизменяемым типам данных, поэтому результатом любых манипуляций, связанных с преобразованием регистра или удалением (заменой) символов будет новая строка.

Из всех методов, связанных с конвертацией регистра, наиболее часто используются на практике два – **lower()** и **upper().** Они преобразуют все символы в нижний и верхний регистр соответственно:

```
>>> text = 'Этот текст надо написать заглавными буквами'
>>> print(text.upper())
ЭТОТ ТЕКСТ НАДО НАПИСАТЬ ЗАГЛАВНЫМИ БУКВАМИ
>>> text = 'зДесь ВСЕ буквы рАзныЕ, а НУжНы Строчные'
>>> print(text.lower())
здесь все буквы разные, а нужны строчные
```

Для того, чтобы преобразовать текст, чтобы с заглавной буквы начиналось только первое слово предложения, необходимо использовать метод **capitalize():**

```
>>> text = 'предложение должно начинаться с ЗАГЛАВНОЙ буквы'
>>> print(text.capitalize())
Предложение должно начинаться с заглавной буквы
```

Методы **swapcase()** и **title()** используются реже. Первый заменяет исходный регистр на противоположный, а второй – начинает каждое слово с заглавной буквы:

```
>>> text = 'пРИМЕР иСПОЛЬЗОВАНИЯ swapcase'
>>> print(text.swapcase())
Пример Использования SWAPCASE
>>> text2 = 'тот случай, когда нужен метод title'
>>> print(text2.title())
Тот Случай, Когда Нужен Метод Title
```

4) Поиск, подсчет и замена символов

Методы **find()** и **rfind()** возвращают индекс стартовой позиции искомой подстроки. Оба метода учитывают только **первое** вхождение подстроки. Разница между ними заключается в том, что **find()** ищет первое вхождение подстроки с начала текста, а **rfind()** – **с конца**:

```
>>> text = 'пример текста, в котором нужно найти текстовую подстроку'
>>> print(text.find('текст'))
7
>>> print(text.rfind('текст'))
37
```

Такие же результаты можно получить при использовании методов **index()** и **rindex()** – правда, придется предусмотреть обработку ошибок, если искомая подстрока не будет обнаружена:

```
>>> text = 'Хочется еще попить чаю'
>>> print(text.index('еще'))
8
>>> print(text.rindex('чай'))
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(text.rindex('чай'))
ValueError: substring not found
```

Если нужно определить, начинается ли строка с определенной подстроки, поможет метод **startswith():**

```
>>> text = 'Жили да были дед с бабой'
>>> print(text.startswith('Жили'))
True
```

Чтобы проверить, заканчивается ли строка на нужное окончание, используют **endswith():**

```
>>> text = 'В конце всех ждал хэппи-енд'
>>> print(text.endswith('енд'))
True
```

Для подсчета числа вхождений определенного символа или подстроки применяют метод **count()** – он помогает подсчитать как общее число вхождений в тексте, так и вхождения в указанном диапазоне:

```
>>> text = 'Жили-были три китайца: Як, Як-цедрак, Як-цедрак-цедрак-цедрони'
>>> print(text.count('и'))
6
>>> print(text.count('и', 9, 25))
2
```

Методы **strip(), lstrip()** и **rstrip()** предназначены для удаления пробелов. Метод **strip()** удаляет пробелы в начале и конце строки, **lstrip()** – только слева, rstrip() – только справа:

```
>>> text = 'здесь есть пробелы и слева, и справа  '
>>> print('***', text.strip(), '***')
*** здесь есть пробелы и слева, и справа ***
>>>
>>> print('***', text.lstrip(), '***')
*** здесь есть пробелы и слева, и справа   ***
>>>
>>> print('***', text.rstrip(), '***')
*** здесь есть пробелы и слева, и справа ***
```

Метод **replace()** используют для замены символов или подстрок. Можно указать нужное **количество замен,** а сам символ можно заменить на пустую подстроку – проще говоря, **удалить**:

```
>>> text = 'таракан'
>>> text = text.replace('а', 'у',1)
>>> text
'туракан'
>>> text = text.replace('а', 'у')
>>> text
'турукун'
```

Данные методы и функции работы со строками пригодятся для решения задач.