**Строка — неизменяемая последовательность символов**

Во многих языках программирования имеются такие структуры данных, которые позволяют хранить набор объектов, доступных по индексу (номеру вхождения). Например, статический массив в С++ имеет фиксированный размер и содержит группу объектов одного типа.

Последовательности в Python позволяют хранить набор объектов как одного типа данных, так и разных. Причем некоторые из этих наборов (например, списки) могут также увеличиваться или уменьшаться в размерах.

Одним из самых распространенных видов последовательностей является строка — последовательность символов. Как вам уже известно, в Python существуют изменяемые и неизменяемые типы данных. Строка (str) является неизменяемым (immutable) типом данных. Как было рассмотрено ранее, при создании переменной, вначале будет создан объект, в состав которого входит уникальный идентификатор, тип и значение. Только после этого переменная может ссылаться на уже созданный объект.

В Python есть несколько способов задания строк.

str1 = 'Hello1'

str2 = "Hello2"

str3 = '''Многострочные

строки 1'''

Если в кавычках ничего не записано:

str0 = ""

то это будет пустая строка, не содержащая никаких символов.

При работе с переменными в Python всегда следует помнить, что это лишь ссылки на соответствующие объекты. Из этого следует, что если у нас определена некая строка:

str1 = "сообщение"

и мы другой переменной присвоим первую:

str2 = str1

то получим две ссылки на один и тот же объект (одну строку). То есть, копирование строки здесь не происходит!



**Операторы и функции для строк**

Первый оператор + – это соединение двух строк (или как еще говорят, **конкатенация** строк). Он используется так:

str1 = 'Hello'

str2 = "world!"

msg = str1+str2

**print**(msg)

**строка 1-5**

Здесь сначала будут идти символы первой строки, а затем – второй. В результате у нас формируется новый объект, содержащий эти символы. Но у нас оба слова слились в одно. Как можно было бы добавить между ними пробел? Это можно сделать так:

msg = str1+" "+str2

то есть, мы можем последовательно соединять множество строк между собой. И здесь важно знать, что соединять между собой можно только строки.

можно преобразовав число в строку с помощью функции

<p align=center>str(<аргумент>)

msg = "число = "+str(dig)

Далее, в Python довольно просто выполняется дублирование строки. Предположим у нас есть вот такая строка:

one = 'ай '

и мы хотим ее размножить n раз. Это можно сделать так:

msg = one\*10

**print**(msg)

**строка 6-8**

Разумеется, число копий должно определяться исключительно целыми числами,

msg = one\*3.5

Далее, для определения длины строки, то есть, числа символов в строке (включая управляющие символы), используется функция

**len(<строка>)**

Например:

N = len(msg)

**print**(msg, N)

Затем, для проверки наличия в строке той или иной подстроки, используется оператор **in:** Он возвращает True, если подстрока присутствует и False, если отсутствует. Например:

s = "abcdefg0123"

"abc" **in** s

'0' **in** s

'43' **in** s

**строка 13-17**

Для сравнения строк между собой можно использовать оператор сравнения

**<строка 1> == <строка 2>**

Данный оператор возвращает True, если строки равны и False, если не равны. Пример:

"abc" == 'abc'

"ABC" == 'abc'

Обратите внимание, строка, записанная заглавными буквами – это в Python уже другая строка и оператор сравнения для них возвращает False.

Для сравнения неравенства строк используется оператор не равно:

**<строка 1> != <строка 2>**

Он возвращает True, если строки не равны и False в противном случае.

Разумеется, эти операции сравнения обычно используются в условном операторе if или операторах циклов while и for, о которых мы уже с вами говорили.

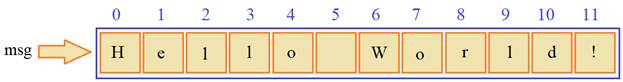
Следующая функция **ord(<символ>)**

Ord — это функция которая возвращает целое число, представляющее переданный ей символ. Это означает, что каждый символ имеет некоторое целочисленное значение, эти символы не могут быть прочитаны компьютером непосредственно, он сначала преобразует каждый символ в код ASCII.

ord("a")

**Индексы и срезы строк**

На самом деле в Python строка представляются как упорядоченная коллекция символов. И ключевое слово здесь – «упорядоченная». Это значит, что у каждого символа в строке есть свой порядковый номер – индекс, по которому мы можем его получить. Например, когда мы создаем строку то формируется следующая коллекция:



Каждый символ имеет свой индекс, начиная с нулевого. Первый символ в Python всегда имеет нулевой индекс.

Например:

msg[0]

msg[6]

и так далее. Но, если указать неверный индекс, например:

msg[12]

то возникнет ошибка. Поэтому здесь следует быть аккуратным и не выходить за пределы этого списка.

В частности, последний «рабочий» индекс можно определить с помощью функции len – длины строки:

То есть, к последнему индексу мы можем обратиться так:

msg[len(msg)-1]

Но это не очень удобно. Поэтому разработчики языка Python решили, что отрицательные индексы будут означать движение по строке с конца в начало. И предыдущую запись можно переписать так:

msg[-1]

То есть, у строк есть еще такие отрицательные индексы:

Также в Python можно использовать доступ к отдельному символу непосредственно у строкового литерала:

"abcd"[1]

"abcd"[-1]

Иногда это бывает удобно.

## Срезы

Часто в программировании требуется выбрать не один какой-то символ, а сразу несколько. Для этого используются так называемые срезы. Их работу проще показать на конкретных примерах. Пусть у нас есть наша строка:

msg = "Hello World!"

и мы хотим выделить последнее слово «World!». Для этого в квадратных скобках указывается начальный индекс и через двоеточие – конечный. Если мы запишем все вот в таком виде:

msg[6:11]

то получим результат «World» без восклицательного знака. Дело в том, что последний индекс исключается из интервала, то есть, интервал определяется как

[6: 11)

Поэтому, мы должны записать срез так:

msg[6:12]

В Python допускается не указывать начальное или конечное значения, или даже, оба из них. Например:

msg[:5]

выделяет слово «Hello», а вот так:

msg[6:]

получим «World!». Наконец, записав все в таком виде:

msg[:]

получим ту же самую строку, не копию!

В срезах на Python можно дополнительно указывать шаг через двоеточие.

Например, так:

msg[::2]

мы здесь ставим еще одно двоеточие и указываем шаг 2, то есть, идем через символ: «HloWrd». Также это можно комбинировать с граничными значениями:

msg[:5:2]

msg[6::2]

msg[1:6:2]

и использовать отрицательный шаг:

msg[::-1]

в этом случае символы будут перебираться в обратном порядке.

ПРАКТИКА

**Подвиг 1.**

Напишите программу последовательного ввода двух слов. Сформируйте новую строку, продублировав первое слово дважды, а второе - трижды (все слова в результирующей строке должны идти через пробел). Результат выведите на экран.

Программу следует реализовать без использования F-строк, а с применением оператора дублирования строк.

**Sample Input:**

hello python

**Sample Output:**

hello hello python python python

**Подвиг 2.**

Напишите программу ввода строки и отображения на экране ее первого и последнего символа в виде одной строки.

**Sample Input:**

I love Python

**Sample Output:**

In

**Подвиг 3.**

Напишите программу отображения всех символов с нечетными индексами из введенной строки.

**Sample Input:**

Krivov

**Sample Output:**

rvv

**Подвиг 9.**

Вводятся два слова. Длина первого меньше второго. Необходимо обрезать второе слово до длины первого и отобразить обрезанное слово на экране.

**Sample Input:**

Hello Krivov

**Sample Output:**

Krivo

# **Основные методы строк**

В Python строки являются объектами и у этих объектов есть методы, то есть, функции, выполняющие определенные действия:

Для примера, предположим, у нас имеется такая, уже классическая строка:

string = "Hello World!"

и мы собираемся для нее вызвать метод

String.upper()

который возвращает строку со всеми заглавными буквами. Для этого, пишется сама строка, ставится точка и записывается имя метода. В конце обязательно ставим круглые скобки:

string.upper()

Вот по такому синтаксису вызываются различные методы строк. Причем, сама переменная string продолжает ссылается на ту же самую неизмененную строку «Hello World!».

строки – это неизменяемые объекты, поэтому метод upper возвращает новую строку с заглавными буквами, не меняя прежней.

Если бы нам потребовалось изменить строку, на которую ссылается переменная string, то это можно сделать так:

string = string.upper()

В этом случае переменная станет ссылаться на новый строковый объект с заглавными буквами, а прежний будет автоматически удален сборщиком мусора (так как на него не будет никаких внешних ссылок).

Также этот метод мы можем вызвать непосредственно у строкового литерала:

"hello".upper()

Ну и раз уж мы затронули метод upper, который переводит буквы в верхний регистр, то отметим противоположный ему метод: **String.lower()**

который, наоборот, преобразует все буквы в строчные. Например:

string.lower()

возвращает строку «hello world!». Соответственно, сама строка здесь остается прежней, измененным является новый строковый объект, который и возвращает метод lower.

По такому принципу работают все методы при изменении строк. Следующий метод **String.count(sub[, start[, end]])** возвращает число повторений подстроки sub в строке String.

Два необязательных аргумента:

* start – индекс, с которого начинается поиск;
* end – индекс, которым заканчивается поиск.

В самом простом случае, мы можем для строки

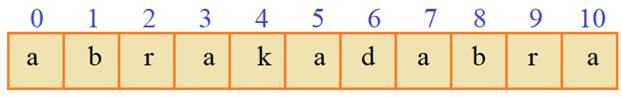
msg = "abrakadabra"

определить число повторений сочетаний «ra»:

msg.count("ra")

получим значение 2 – именно столько данная подстрока встречается в нашей строке.

Теперь предположим, что мы хотим начинать поиск с буквы k, имеющей индекс 4.



Тогда метод следует записать со значением start=4:

msg.count("ra", 4)

и мы получим значение 1. Далее, укажем третий аргумент – индекс, до которого будет осуществляться поиск. Предположим, что мы хотим дойти до 10-го индекса и записываем:

msg.count("ra", 4, 10)

и получаем значение 0. Почему? Ведь на индексах 9 и 10 как раз идет подстрока «ra»? Но здесь, также как и в срезах, последний индекс исключается из рассмотрения. То есть, мы говорим, что нужно дойти до 10-го, не включая его. А вот если запишем 11:

msg.count("ra", 4, 11)

то последнее включение найдется.

Следующий метод **String.find(sub[, start[, end]])** возвращает индекс первого найденного вхождения подстроки sub в строке String. А аргументы start и end работают также как и в методе count. Например:

msg.find("br")

вернет 1, т.к. первое вхождение «br» как раз начинается с индекса 1. Поставим теперь значение start=2:

Метод find ищет первое вхождение слева-направо. Если требуется делать поиск в обратном направлении: справа-налево, то для этого используется метод

**String.rfind(sub[, start[, end]])**

который во всем остальном работает аналогично find.

msg.rfind("br")

возвратит 8 – первое вхождение справа.

Наконец, третий метод, аналогичный find – это: **String.index(sub[, start[, end]])**

Он работает абсолютно также как find, но с одним отличием: если указанная подстрока sub не находится в строке String, то метод приводит к ошибке:

msg.index("brr")

тогда как find возвращает -1.

Спрашивается: зачем нужен такой ущербный метод index? В действительности такие ошибки можно обрабатывать как исключения и это бывает полезно для сохранения архитектуры программы: когда неожиданные ситуации обрабатываются единым образом в блоке исключений. Но, обо всем этом речь пойдет позже.

Следующий метод

**String.replace(old, new, count=-1)**

Выполняет замену подстрок old на строку new и возвращает измененную строку. Например, в нашей строке, мы можем заменить все буквы a на o:

msg.replace("a", 'o')

на выходе получим строку «obrokodobro». Или, так:

msg.replace("ab", "AB")

Используя этот метод, можно выполнять удаление заданных фрагментов, например, так:

msg.replace("ab", "")

Третий необязательный аргумент задает максимальное количество замен. Например:

msg.replace("a", 'o', 2)

Заменит только первые две буквы a: «msg.replace("a", 'o', 2)». При значении -1 количество замен неограниченно.

Следующие методы позволяют определить: из каких символов состоит наша строка. Например, метод **String.isalpha()** возвращает True, если строка целиком состоит из букв и False в противном случае. Посмотрим как он работает:

msg.isalpha()

вернет True, т.к. наша строка содержит только буквенные символы. А вот для такой строки:

"hello world".isalpha()

мы получим False, т.к. имеется символ пробела.

Похожий метод **String.isdigit()** возвращает True, если строка целиком состоит из цифр и False в противном случае. Например:

"5.6".isdigit()

т.к. имеется символ точки, а вот так:

"56".isdigit()

получим значение True. Такая проверка полезна, например, перед преобразованием строки в целое число:

dig = input("Введите число: ")

**if**(dig.isdigit()):

   dig = int(dig)

**print**(dig)

**else**:

**print**("Число введено неверно")

Следующий метод **String.rjust(width[, fillchar = ‘ ‘])** возвращает новую строку с заданным числом символов width и при необходимости слева добавляет символы fillchar:

d="abc"

d.rjust(5)

Получаем строку «  abc» с двумя добавленными слева пробелами. А сама исходная строка как бы прижимается к правому краю. Или, можно сделать так:

d.rjust(5, "-")

Получим строку «--abc». Причем вторым аргументом можно писать только один символ. Если записать несколько, то возникнет ошибка:

d.rjust(5, "-\*")

Если ширина width будет меньше длины строки:

d.rjust(2)

то ничего не изменится.

Аналогично работает метод **String.ljust(width[, fillchar = ‘ ‘])** который возвращает новую строку с заданным числом символов width, но добавляет символы fillchar уже справа:

d.ljust(10, "\*")

Следующий метод **String.split(sep=None, maxsplit=-1)** возвращает коллекцию строк, на которые разбивается исходная строка String. Разбивка осуществляется по указанному сепаратору sep. Например:

"Иванов Иван Иванович".split(" ")

Мы здесь разбиваем строку по пробелам. Получаем коллекцию из ФИО. Тот же результат будет и при вызове метода без аргументов, то есть, по умолчанию он разбивает строку по пробелам:

А теперь предположим, перед нами такая задача: получить список цифр, которые записаны через запятую. Причем, после запятой может быть пробел, а может и не быть. Программу можно реализовать так:

digs = "1, 2,3, 4,5,6"

digs.replace(" ", "").split(",")

мы сначала убираем все пробелы и для полученной строки вызываем split, получаем список цифр.

Обратный метод **String.join(список)** возвращает строку из объединенных элементов списка, между которыми будет разделитель

fio = "Иванов Иван Иванович"

и мы хотим здесь вместо пробелов поставить запятые:

fio2 = ",".join(fio.split())

Теперь fio2 ссылается на строку с запятыми «Иванов,Иван,Иванович».

Следующий метод **String.strip()** удаляет пробелы и переносы строк в начале и конце строки. Например:

"   hello world         **\n**".strip()

возвращает строку «hello world».

Вот такие методы строк существуют в Python. Для наглядности ниже они представлены в таблице:

|  |  |
| --- | --- |
| Название | Описание |
| String.upper() | Возвращает строку с заглавными буквами |
| String.lower() | Возвращает строку с малыми буквами |
| String.count(sub[, start[, end]]) | Определяет число вхождений подстроки в строке |
| String.find(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения |
| String.rfind(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения при поиске справа |
| String.index(sub[, start[, end]]) | Возвращает индекс первого найденного вхождения |
| String.replace(old, new, count=-1) | Заменяет подстроку old на new |
| String.isalpha() | Определяет: состоит ли строка целиком из буквенных символов |
| String.isdigit() | Определяет: состоит ли строка целиком из цифр |
| String.rjust(width[, fillchar = ‘ ‘]) | Расширяет строку, добавляя символы слева |
| String.ljust(width[, fillchar = ‘ ‘]) | Расширяет строку, добавляя символы справа |
| String.split(sep=None, maxsplit=-1) | Разбивает строку на подстроки |
| String.join(список) | Объединяет коллекцию в строку |
| String.strip() | Удаляет пробелы и переносы строк справа и слева |
| String.rstrip() | Удаляет пробелы и переносы строк справа |
| String.ltrip() | Удаляет пробелы и переносы строк слева |

## Задания для самоподготовки

1. Написать программу изменения строки

"2+3+6.7 + 82 + 5.7 +1"

на строку, в которой все «+» заменены на «-» и удалены все пробелы

3. Написать программу вывода чисел 0; -100; 5.6; -3 в виде столбца:

0

-100

5.6

-3

в котором все строки выровнены по левому краю (подсказка: воспользуйтесь методом rjust).

4. В строке "abrakadabra" найдите все индексы подстроки «ra» и выведите их (индексы) в консоль.

**Сырые строки**

Для решения этой проблемы в Python предусмотрены «Сырые» строки (raw strings). Raw strings «рассматривают» обратную косую черту (\) как буквальный символ. Это полезно в случаях, когда нам необходимо работать со строкой, содержащую обратную косую черту (\), и мы не хотим, чтобы она рассматривалась как escape-последовательность. «Сырые» строки Python имеют префикс «r» или «R». Просто добавьте к строке префикс «R» или «r», и она будет рассматриваться как «raw strings».

# **F-строки и метод format**

На этом занятии мы подробнее поговорим, как в Python можно формировать строки на основе набора различных переменных и даже функций. Мы уже знаем с вами как создавать строки и соединять их между собой. Предположим, у нас есть две переменные:

age=18; name="Федор"

и мы формируем такое сообщение:

"Меня зовут "+name+", мне "+str(age)+" и я люблю язык Python."

На выходе получим:

'Меня зовут Федор, мне 18 и я люблю язык Python.'

Но это не самый удобный и распространенный способ формирования строк в Python. Гораздо удобнее использовать специальный метод строки **str.format(\*args)**

который в данном конкретном случае можно использовать так:

"Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)

Смотрите, здесь в фигурных скобках мы указываем индекс переменной, значение которой будет подставлено в это место строки. Индекс 0 – первая переменная, указанная в методе format, т.е. name, а индекс 1 – вторая переменная age. То есть, метод формат возвращает новую строку в заданном формате. И чтобы сохранить этот результат в какой-либо переменной, следует записать так:

msg = "Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)

Теперь msg ссылается на созданную строку.

Также мы можем одну и ту же переменную указывать несколько раз, например, так:

"Меня зовут {0}, мне {1} и я люблю язык Python. {0}".format(name, age)

Как видите, это удобнее обычной конкатенации строк. Но эту запись можно сделать еще понятнее, используя именованные параметры. Для этого у каждой переменной пропишем ее имя (ключ). Оно придумывается программистом, например:

format(fio=name, old=age)

и, далее, в формате строки указываются уже эти имена:

"Меня зовут {fio}, мне {old} и я люблю язык Python. {fio}".format(fio=name, old=age)

## F-строки

Начиная с версии Python 3.6, появился еще один способ форматирования строк, так называемые, F-строки. И во многих случаях он еще более удобный, чем использование метода format.

Итак, чтобы указать Python на использование механизма F-строк, перед ее литералом ставится символ ‘f’ (именно малая буква), а в фигурных скобках записывать непосредственно имена переменных:

f"Меня зовут {name}, мне {age} и я люблю язык Python."

Можем в дальнейшим оперировать им через переменную msg. Далее, внутри фигурных скобок F-строк в действительности можно записывать любые конструкции языка Python, например, арифметические операции и операции со строками:

f"Меня зовут {name.upper()}, мне {age\*2} и я люблю язык Python."

## Задания для самоподготовки

1. Пользователь через пробел вводит ФИО. На основе этой информации требуется создать строку с сообщением:

Ваши персональные данные:  
Фамилия: введенная фамилия  
Имя: введенное имя  
Отчество: введенное отчество

2. Имеется текстовый файл с содержимым:

Иван, ivan@gm.com, 18  
Татьяна, tat@gm.com, 22  
Сергей, srg@ml.ru, 33  
Федор, fr@ml.ru, 41  
Елена, el@gm.com, 27

Необходимо построчно считывать информацию и для каждой строки для лиц не старше 30 лет сформировать сообщение:

Уважаемый(ая) <имя>! Приглашаем Вас принять участие в курсах по изучению Python. Подробную информацию мы выслали на email: <email>.

**Модуль string**

В предыдущих разделах мы познакомились со многими полезными методами объекта строки (str). Большая часть из них ранее была реализована в отдельном модуле string. Однако в связи с высокой частотой их использования они были перенесены в методы объекта str, чтобы не нужно было выполнять предварительный импорт модуля.

Сейчас модуль string содержит набор констант, утилит и классов для работы с объектами str (например, возможности создания пользовательских шаблонов для настройки собственного форматирования строк). Для использования возможностей модуля string вначале необходимо выполнить его импорт

Предположим, что нам необходимо сгенерировать случайным образом логин пользователя заданной длины (например, 6 символов), который должен состоять только из символов латинского алфавита в нижнем регистре, и пароль (8 символов), который должен состоять из символов латинского алфавита и десятичных цифр.

Для этого мы воспользуемся методом **sample() модуля random,** который извлекает случайные элементы из последовательности в заданном количестве. Для объединения полученного набора символов можно использовать метод строки join().

Строка 102 -110

В состав модуля string входит единственная функция capwords(): Эта функция вначале разделяет строку strObj на слова, затем первый символ каждого слова переводит в верхний регистр после соединяет преобразованные слова в новую строку, используя указанный разделитель. Если аргумент sep не указан (его значение по умолчанию None), то начальные и конечные пробелы (если они есть) будут удалены из строки strObj, а слова новой строки будут соединены символом одного пробела.

## Язык регулярных выражений

Этот язык является универсальным и не привязан к конкретному языку программирования. То есть, изучив его, вы сможете использовать регулярные выражения и в Perl и в JavaScript и в Java и т.д. Многие современные языки программирования поддерживают возможность обработки строк с помощью таких шаблонов.

Часто требуется найти не одно какое-то конкретное написание слова, а все его множественные формы, например: еда, еду, Еда, Еду

здесь первая буква может быть малой или заглавной, а последняя или а, или у. Чтобы в нашем шаблоне охватить все эти варианты используются *символьные классы*, которые определяются квадратными скобками:

# **Квантификаторы регулярных выражений**

Продолжаем тему регулярных выражений и на этом занятии поговорим о **квантификаторах**. В общем виде они записываются в виде (без пробелов): {m,n}

где m – минимальное число совпадений с выражением; n – максимальное число совпадений с выражением.

**import** re

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"o{2,5}", text)

**print**(match)

Здесь будут выделены следующие комбинации: "Google, Gooogle, Goooooogle"

Обратите внимание, по умолчанию квантификатор находит наиболее длинные последовательности. Про такой режим еще говорят, что он ***жадный* или *мажорный***. В противоположность ему есть другой, ***минорный*** режим работы.

В этом случае ищутся последовательности минимальной длины, удовлетворяющие шаблону. Для перевода квантификатора в минорный режим после него записывается символ вопроса:

match = re.findall(r"o{2,5}?", text)

В результате получим вхождения:

['oo', 'oo', 'oo', 'oo', 'oo']

которые образуются так:

"Google, Gooogle, Goo**oo**oogle"

## Краткие формы записи квантификаторов

Квантификаторы можно записывать и в кратких формах, например:

* {m} – повторение выражения ровно m раз (эквивалент {m,m});
* {m,} – повторения от m и более раз;
* {, n} – повторения не более n раз.

Разумеется, для последних двух форм также можно использовать минорный режим:

{m,}?    {,n}?

Например:

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"Go{2,}gle", text)

Будет находить все записи слова «Google» с двумя символами ‘o’ и более. Или, так:

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"Go{,4}gle", text)

Найдет первые два слова, последнее не соответствует шаблону.

И, наконец, такой пример:

phone = "89123456789"

match = re.findall(r"8**\d**{10}", phone)

выделяет телефонные номера с первой цифрой 8 и следующими 10 цифрами.