Модуль 10. Регулярные выражения (2 ак. ч.)

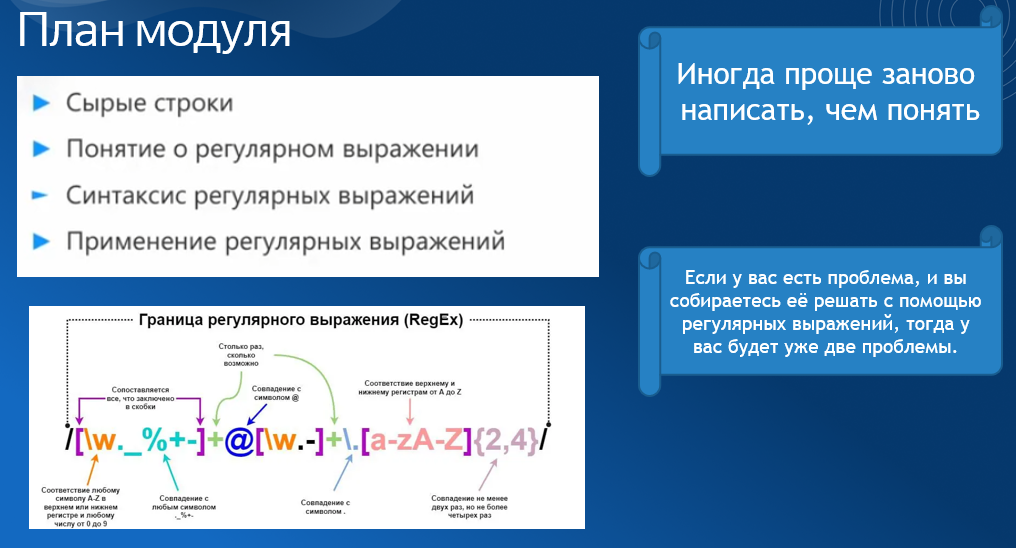
[Презентация 1](#_Toc143192441)

[Квантификаторы регулярных выражений 11](#_Toc143192442)

[Краткие формы записи квантификаторов 12](#_Toc143192443)

[Практические задания с решениями 16](#_Toc143192444)

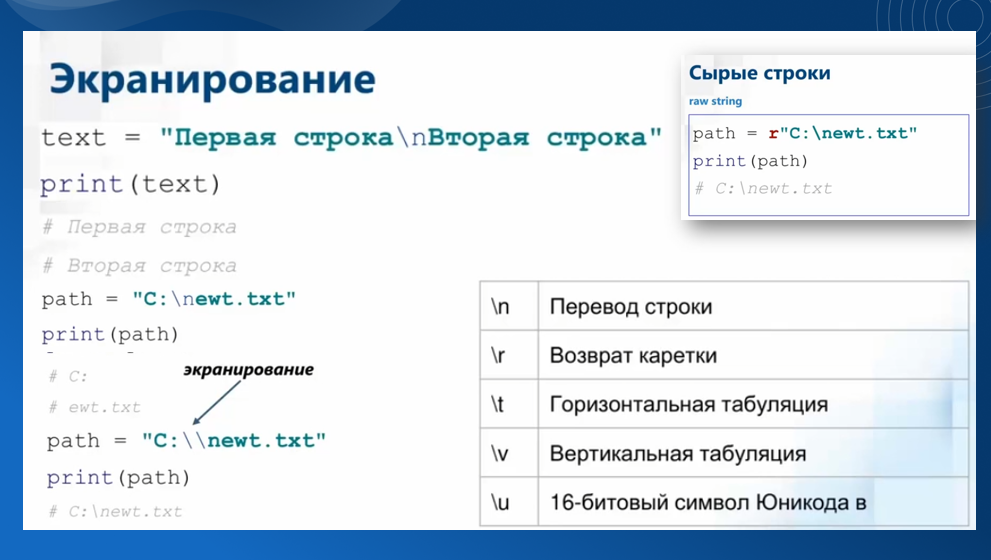
## Презентация



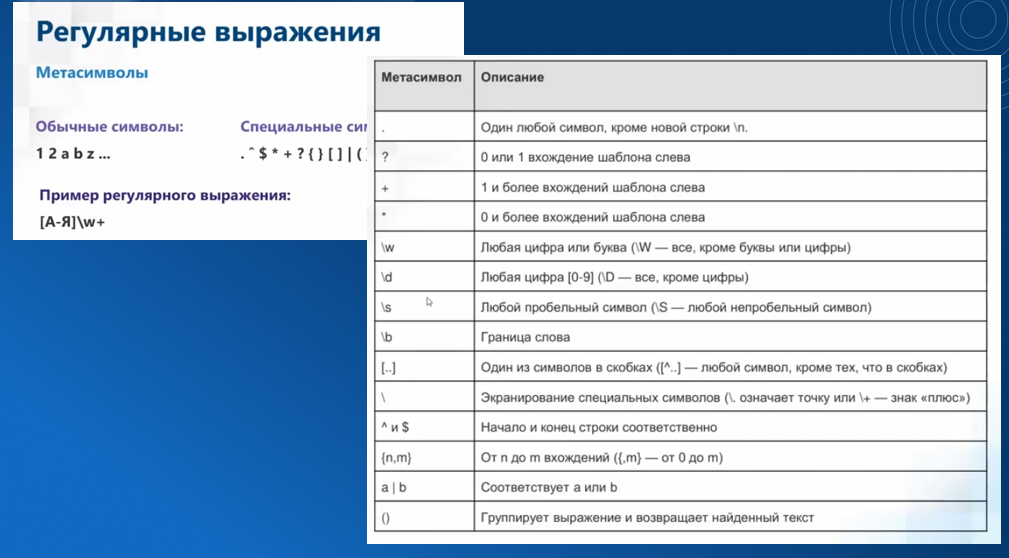
Маска числа  — это последовательность цифр, в которой могут встречаться специальные символы «?» и «\*». Символ «?» означает ровно одну произвольную цифру, символ «\*» означает произвольную (в том числе пустую) последовательность цифр.

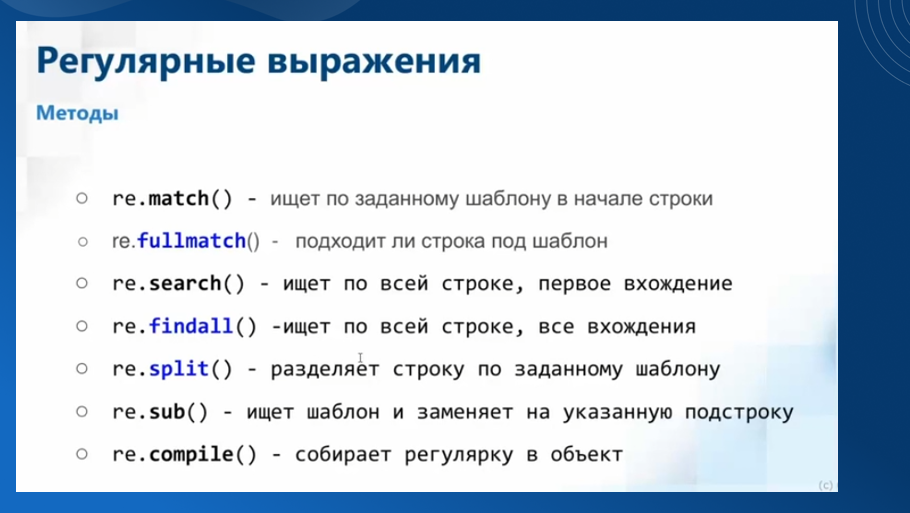
Изучив re вы легко можете его применить для:

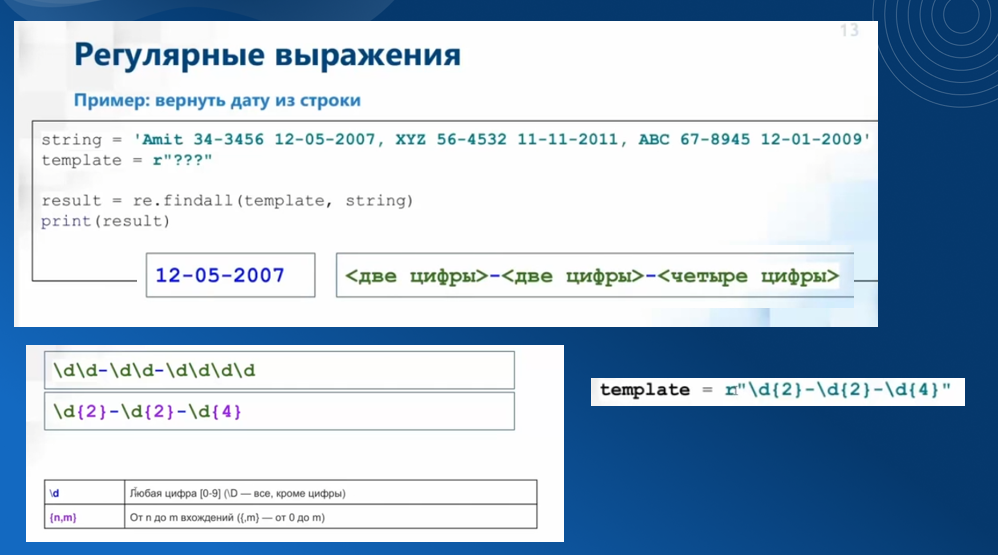
Perl, JS, Java и т.д

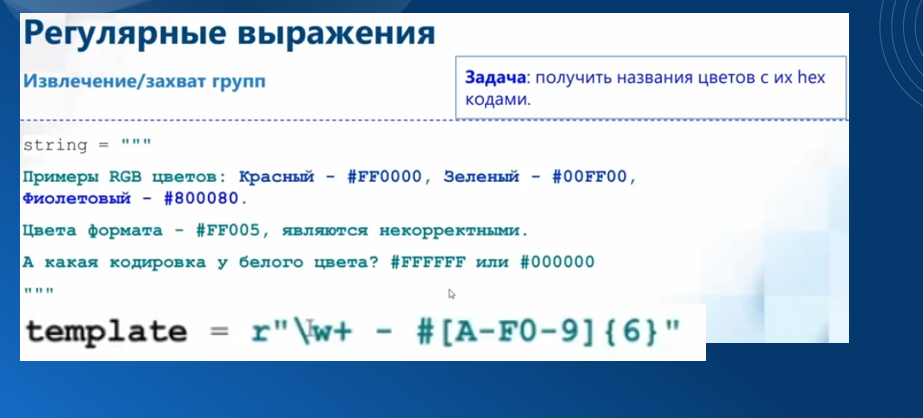


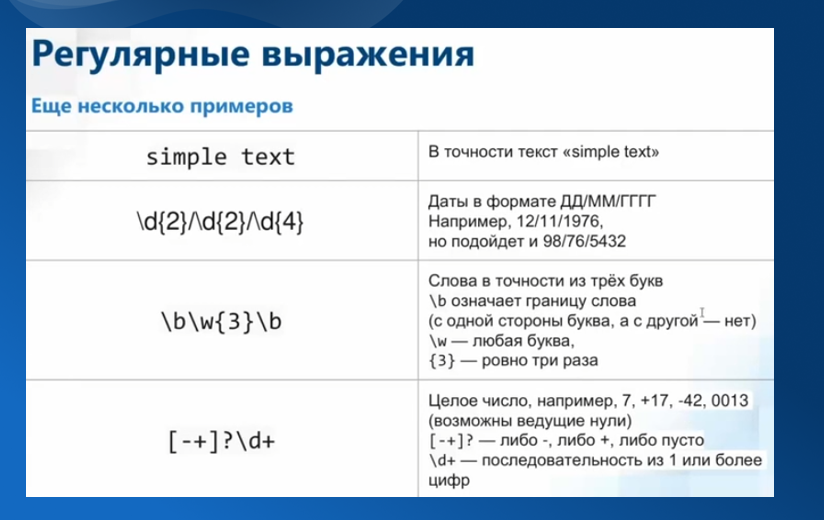
Методы re написаны написаны на языке СИ, что обеспечивает высокую скорость обработки слов

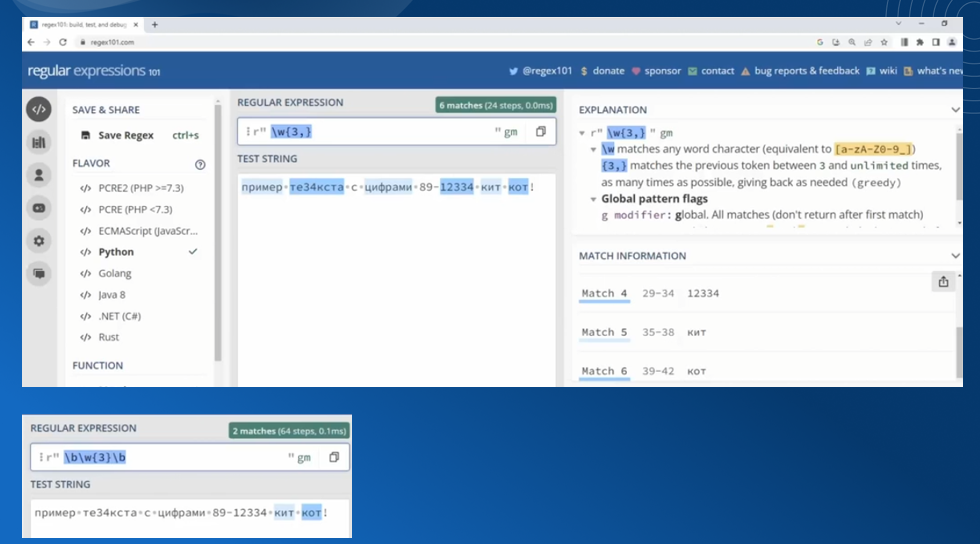


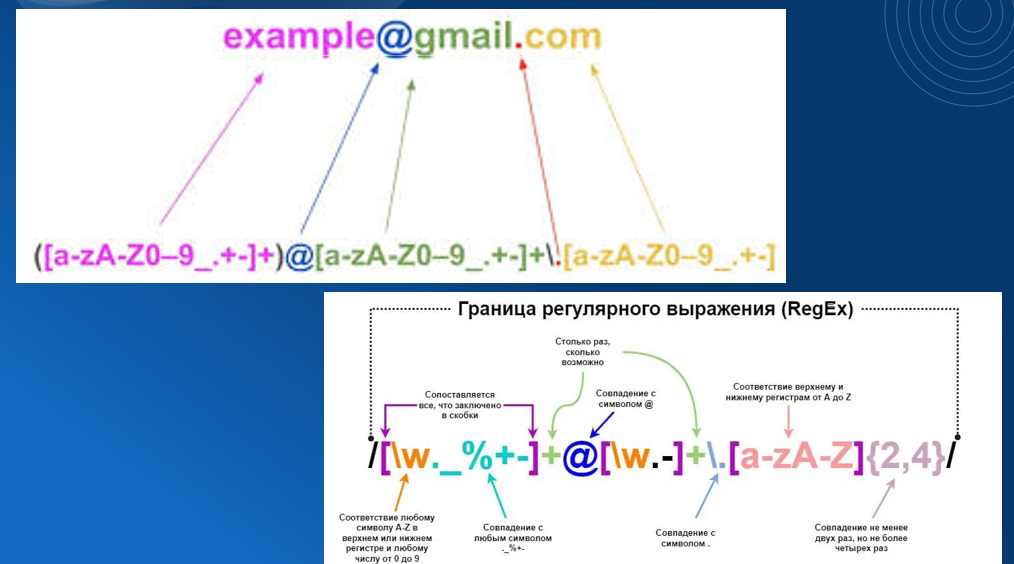












Пример 1.

**import** re  
text = **"я программист Python, мы - фулстек-программисты"**search = re.findall(**'программист'**, text)  
print (search)

результат:

['программист', 'программист']

Расширим функционал

text = "я программист Python, мы - фулстек-программисты, Python-аналитик"  
text1 = "программист Python"  
  
search\_all = re.findall('программист', text)  
print (search\_all)  
print()  
  
search\_split = re.split('программист', text)  
print (search\_split)  
print()  
  
search = re.search('программист', text)  
print (search)  
print()  
  
search\_match = re.match('программист', text1)  
print (search\_match)

Какая проблема? Нам нужно только отдельные слова. Решение

search = re.findall(**'\\bпрограммист\\b'**, text)

Но два слэша писать не принято, так как они используются в шаблонах регулярных выражений. Поэтому:

text = "я программист Python, мы - фулстек-программисты, Python-программист-аналитик"  
  
search1 = re.findall(r'\bпрограммист\b', text)  
*#\b = границы слова*print(search1)

Но! Как найти склонения у программиста

text = "я Программист Python, мы - фулстек-программисты, у Python-программиста-аналитика"

search1 = re.findall(r'\b[Пп]рограммист[аы]\b', text)

Результат

['программисты', 'программиста']

Где программист с большой буквы?

Спрятался за звездочкой

search1 = re.findall(r'\b[Пп]рограммист[аы]\*\b', text)

или так

search1 = re.findall(r'\b[Пп]рограммист[аы]?\b', text)

# ? может быть, а может и не быть

Задача

text = "стеклянный, стекляный"

Написать шаблон для поиска стеклянных слов  
Решение:

search2 = re.findall(r"стеклянн?ый", text)  
print(search2)

И, наконец, такой пример:

phone = "89123456789, 89883137443, +79123456789, +59123456789"

Найти все которые начинаются на 8 и дальше ? сколько цифр в номере телефона?  
  
match = re.findall(r"8[0-9]{10}", phone) #1 способ  
match = re.findall(r"8\d{10}", phone) #1 способ  
  
print(match)

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

Для квантификаторов {0,} и {1,} существуют специальные символы:

* ? – от нуля до одного (аналог {0,1});
* \* – от нуля и до «бесконечности» (в действительности, большого числа – от 32767), соответствует квантификатору {0,};
* + – от единицы и до «бесконечности» (также большого числа – от 32767), соответствует квантификатору {1,}.

Все эти сокращения также можно использовать в минорном режиме:

??    \*?    +?

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. Например:

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"

{m,}?    {,n}?

Но нам нужно поймать гуглов?

text = "Google, Gooogle, Goooooogle"

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

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

А если так, то:

text = "Google, Gooogle, Goooooogle"

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

Найдет первые два слова, последнее не соответствует шаблону.

Далее, допустим у нас имеется строка с данными:

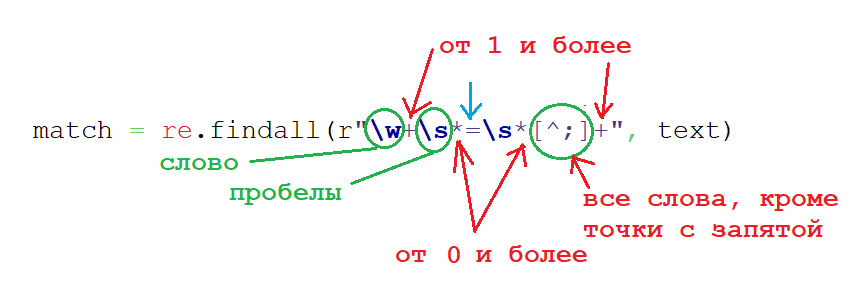
text = "author=Пушкин А.С.; title = Евгений Онегин; price =200; year= 2001"

и мы хотим выполнить парсинг (разбор) по ключам и значениям. Это можно реализовать так:

match = re.findall(r"**\w**+**\s**\*=**\s**\*[^;]+", text)

на выходе получим следующий список:

['author=Пушкин А.С.', 'title = Евгений Онегин', 'price =200', 'year= 2001']



Конечно, в данном конкретном случае этот же список можно получить гораздо проще:

d = text.split(";")

**print**(d)

Но, в регулярных выражениях мы, во-первых, точно определяем шаблон данных: ключ=значение. И, во-вторых, с этими данными можно выполнять тонкую обработку, например, сразу выделить и ключ и значение:

match = re.findall(r"(**\w**+)**\s**\*=**\s**\*([^;]+)", text)

В результате, получим список кортежей:

[('author', 'Пушкин А.С.'), ('title', 'Евгений Онегин'), ('price', '200'), ('year', '2001')]

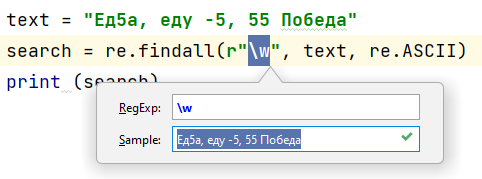
О том, что такое круглые скобки и как они работают, мы поговорим позже. Здесь я просто хотел показать преимущество регулярных выражений перед обычными строковыми методами.

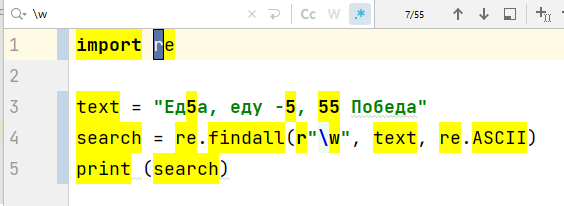
Сервис для проверки регулярны выражений

<https://regex101.com/>

<https://pythex.org/>

средствами pycharm - Alt+Enter (Win)





**Как импортировать ВСЁ!**

**from** string **import** printable  
  
print(printable)

Результат:

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

Далее рассмотрим примеры минорных квантификаторов. Часто это бывает необходимо при разборе HTML-документа. Например, у нас имеется следующий текст:

text = "Картинка <img src='bg.jpg'> в тексте</p>"

и мы хотим выделить фрагмент с тегом <img …>. Если записать выражение так:

match = re.findall(r"<img.\*>", text)

то результат будет следующий:

"<img src='bg.jpg'> в тексте</p>"

Видите, мажорный квантификатор дошел до последней угловой скобки и захватил «лишнюю» часть текста. Решить эту задачу можно несколькими способами. В самом простом случае записать минорный квантификатор:

match = re.findall(r"<img.\*?>", text)

тогда результатом будет строка:

"<img src='bg.jpg'>"

Или же, можно указать символьный класс, не содержащий угловую скобку:

match = re.findall(r"<img[^>]\*>", text)

Результат будет тем же. И здесь может возникнуть вопрос: какой вариант регулярного выражения лучше? В действительности, оба могут быть использованы.

Далее, вы увидите еще много таких случаев, когда одно и то же решение может быть реализовано различными выражениями. Умение составлять короткие и понятные шаблоны – это часть искусства программирования. И, как любое искусство, его постижение приходит с опытом и длится всю жизнь.

Здесь я лишь показываю основные моменты, кирпичики языка регулярных выражений, но умение использовать их на практике приходит только с опытом. Поэтому, больше программируйте, тренируйтесь, развивайтесь и только это приведет вас к желаемым результатам. А теория – это лишь основа, отправная точка для дальнейшего развития. И мы продолжаем ее изучение.

Теперь, смотрите. Оба приведенных варианта, с точки зрения HTML-разметки могут приводить к некорректным результатам, например, просто записи тега

<img>

Это неверная строка, т.к. у этого тега обязательно должен присутствовать атрибут src:

<img src="путь к картинке">

Давайте улучшим наше выражение, чтобы оно учитывало этот атрибут. Это можно сделать так:

match = re.findall(r"<img**\s**+[^>]\*?src**\s**\*=**\s**\*[^>]\*>", text)

Мы здесь берем начало тега <img, затем, обязательно должен идти один или несколько пробелов, далее, могут быть записаны другие атрибуты, потом указываем, что должен быть атрибут src, за которым следует символ = и, затем, идем до ближайшей закрывающей скобки. Такой вариант регулярного выражения будет корректно выделять разные теги img:

"<p>Картинка <img alt='картинка' src='bg.jpg'> в тексте</p>"  
"<p>Картинка <img src='bg.jpg'> в тексте</p>"  
"<p>Картинка <img src='bg.jpg' title='картинка'> в тексте</p>"

Но, вот такие варианты зафиксированы не будут:

"<img>"  
"<p>Картинка <img src2='bg.jpg'> в тексте</p>"  
"<p>Картинка <img alt='картинка'> в тексте</p>"

И другие, где нет атрибута src.

## Практические задания с решениями

**Задание 1.** Вам дана последовательность строк.  
Выведите строки, содержащие "**cat**" в качестве подстроки хотя бы два раза.

**Входные данные:**

catcat

cat and cat

catac

cat

ccaatt

**Выходные данные:**

catcat

cat and cat

Решение 1

s = [**'catcat'**, **'cat and cat'**, **'catac'**, **'cat'**, **'ccaatt'**]  
  
text = **'cat cat'**pattern1 = **r"(cat).\*\1"**# print(re.findall(pattern,text))  
  
**for** string **in** s:  
 f\_all = re.findall(pattern1,string)  
 **if** f\_all:  
 print(string)

Решение 2

pattern2 = **r"cat.\*cat"**

Решение 3

if len(re.findall(r"cat", line)) > 1:

**Задание 2** Вам дана последовательность строк.  
Выведите строки, содержащие "**cat**" в качестве **слова**.

**Входные данные:**

cat

catapult and cat

catcat

concat

Cat

"cat"

!cat?

**Выходные данные:**

cat

catapult and cat

"cat"

!cat?

pattern2 = r"\bcat\b"

**Задача 3.** Вам строка из разных чисел.

Вывести все отрицательные числа, имеющие 2 знака после запятой.

Входные данные:

s =”-2.4 -5.82 -2.73 10 24.3 180 -3.14”

Выходные данные:

-5.82 -2.73 -3.14

pattern2 = r"-\d\*.\d{2}”

**Задача 4.** С помощью регулярных выражений напишите валидатор e-mail адресов

Пример:

prettyandsimple@example.com

very.common@example.com

disposable.style.email.with+symbol@example.com

other.email-with-dash@example.com

x@example.com

example-indeed@strange-example.com

admin@mailserver1

example@localhost

example@s.solutions

user@com

user@localserver

Abc.example.com

A@b@c@example.com

john..doe@example.com

john.doe@example..com

Основные правила, которые определяют корректность email-адреса:

Наличие символа @:

Валидный email-адрес обязательно должен содержать символ @, который разделяет адрес электронной почты на две части – локальную и доменную.

Локальная часть:

Локальная часть email-адреса идет перед символом @ и может содержать буквы, цифры и определенные специальные символы, такие как точка, подчеркивание и дефис.

Доменная часть:

Доменная часть email-адреса идет после символа @ и может содержать буквы, цифры и точки. Домен может также иметь поддомены, разделенные точками.

\b[A-Za-z\_.-][+@[A-Za-z0-9.-]+\.[a-z]{2,3}](mailto:+@[A-Za-z0-9.-]+\.%5ba-z%5d%7b2,3%7d)

[\w\_.%+-]+@[\w.-]+\.[a-z]{2,4}\b

Или просто условная проверка

\S+@\S+\.\S+

**Задача 5.** Нужно настроить регулярное выражение для "Сложного пароля": от 6 символов с использованием цифр, спец. символов, латиницы, наличием строчных и прописных символов.\*

 и проверить работу регулярного выражения на соответствие следующим требованиям.

* содержит хотя бы одно число;
* содержит хотя бы один спецсимвол;
* содержит хотя бы одну латинскую букву в нижнем регистре;
* содержит хотя бы одну латинскую букву в верхнем регистре;
* состоит не менее, чем из 8 вышеупомянутых символов.

|  |  |
| --- | --- |
| Все пароли | Валидные |
| ''' 123456789aB 120980@aA absghk4D abc1&23FF 123ABCac rtG3FG!Tr^e aA1!\*!1Aa oF^a1D@y5e6 enroi#$rkdeR#$092u aA1@ aa1@#bbcc пароль password qwerty lOngPa$$W0Rd ''' | 120980@aA  abc1&23FF  rtG3FG!Tr^e  aA1!\*!1Aa  oF^a1D@y5e6  enroi#$rkdeR#$092u  lOngPa$$W0Rd |

lst = '''  
123456789aB  
120980@aA  
absghk4D  
abc1&23FF  
123ABCac  
rtG3FG!Tr^e  
aA1!\*!1Aa  
oF^a1D@y5e6  
enroi#$rkdeR#$092u  
aA1@  
aa1@#bbcc  
пароль  
password  
qwerty  
lOngPa$$W0Rd  
'''  
  
pattern = re.compile(r'(?=.\*[0-9].\*)(?=.\*[a-z].\*)(?=.\*[A-Z].\*)(?=.\*[!@#$%^&\*].\*)(\b.{8,}\b)')  
  
  
  
  
search = pattern.findall(lst)  
print(\*search,sep='\n')

**Задание 6.** Вам дана последовательность строк.

Выведите строки, содержащие две буквы "**z**﻿", между которыми ровно три символа.

**Входные данные:**

zabcz

zzz

zzxzz

zz

zxz

zzxzxxz

**Выходные данные:**

zabcz

zzxzz

s = [**'zabcz'**,**'zzz'**,**'zzxzz'**,**'zz'**,**'zxz'**,**'zzxzxxz'**]  
# print (s)  
  
pattern2 = **r"z.{3}z"**

**Задание 7.** Вам дана последовательность строк.

Выведите строки, содержащие обратный слеш "**\**﻿".

s = [**"\w denotes word character"**,**"No slashes here"**]

**Входные данные:**

\w denotes word character

No slashes here

**Выходные данные:**

\w denotes word character

pattern2 = **r"\\.\*"**

**Задание 8.** Вам дана последовательность строк.  
Выведите строки, содержащие слово, состоящее из двух одинаковых частей (тандемный повтор).

**Входные данные:**

blabla is a tandem repetition

123123 is good too

go go

aaa

**Выходные данные:**

blabla is a tandem repetition

123123 is good too

s = [**"blabla is a tandem repetition"**, **"123123 is good too"**, **"go go"**, **"aaa bbb"**]  
# print (s)  
  
  
pattern2 = **r"\b(\w+)\1\b",**

# print(re.findall(pattern,text))

**Задание 9.** Вам дана последовательность строк.  
В каждой строке замените **первое** вхождение **слова**, состоящего только из латинских букв "**a**" (регистр не важен), на слово "**argh**".

**Примечание:**

Обратите внимание на параметр **count** у функции **sub**.

**Входные данные:**

There’ll be no more "Aaaaaaaaaaaaaaa"

AaAaAaA AaAaAaA

**Выходные данные:**

There’ll be no more "argh"

argh AaAaAaA

s = [**"There’ll be no more \"Aaaaaaaaaaaaaaa\""**,**"AaAaAaA AaAaAaA"**]  
  
  
pattern2 = **r"\b([aA])+\b"  
  
for** line **in** s:  
 f\_all = re.sub(pattern2,**'argh'**, line, count=1)  
 print(f\_all)

**Задание 10.** Вам дана последовательность строк.  
В каждой строке поменяйте местами две первых буквы в каждом слове, состоящем хотя бы из двух букв.

Буквой считается символ из группы **\w**.

**Входные данные:**

this is a text

"this' !is. ?n1ce,

**Выходные данные:**

htis si a etxt

"htis' !si. ?1nce,

s = [**"this is a text"**,**"\"this' !is. ?n1ce,"**]  
  
print(s)  
  
  
pattern1 = **r"\b(\w)(\w)"  
  
for** line **in** s:  
 f\_all = re.sub(pattern1,**r'\2\1'**, line)  
 print(f\_all)

**Задание 11.** Вам дана последовательность строк.  
В каждой строке замените все вхождения нескольких одинаковых букв на одну букву.

Буквой считается символ из группы **\w**.

**Входные данные:**

attraction

buzzzz

**Выходные данные:**

atraction

buz

import sys,re

s = [**"attraction"**,**"buzzzz"**]

pattern1 = r"(\w)\1+"

for line in s:

line = line.rstrip()

f\_all = re.sub(pattern1,r'\1', line)

print(f\_all)

Задание 11. Вам дана последовательность строк.  
Выведите строки, содержащие двоичную запись числа, кратного **3**.

Двоичной записью числа называется его запись в двоичной системе счисления.

**Примечание 2:**

﻿

Данная задача очень просто может быть решена приведением строки к целому числу и проверке остатка от деления на три, но мы все же предлагаем вам решить ее, не используя приведение к числу.

**Входные данные:**

0

10010

00101

01001

Not a number

1 1

0 0

**Выходные данные:**

0

10010

01001

s = [**'0'**, **'10010'**, **'00101'**, **'01001'**, **'Not a number'**, **'1 1'**, **'0 0'**]  
  
  
pattern1 = **r"([0-1]\*)\S"  
  
for** line **in** s:  
 f\_all = re.fullmatch(pattern1, line)  
 **if** f\_all:  
 **if** int(line,2) % 3 == 0:  
 print (line)

Далее поговорим о группировках и сохранении результатов поиска, используя регулярные выражения. О чем здесь речь? Давайте представим, что нам нужно из текста

lat = 5, lon=7

выделить данные в формате: ключ=значение. Выражение можно записать так:

**import** re

text = "lat = 5, lon=7"

match = re.findall(r"**\w**+**\s**\*=**\s**\***\d**+", text)

**print**(match)

Однако, это же шаблон будет работать, например, и с такими данными:

text = "pi=3, a = 5"

а мы бы хотели, чтобы учитывались только ключи lat и lon. Для этого их нужно явно указать в нашем правиле. Сделаем это следующим образом:

match = re.findall(r"lat**\s**\*=**\s**\***\d**+|lon**\s**\*=**\s**\***\d**+", text)

Смотрите, мы записали два шаблона через символ |, который в регулярных выражениях означает ИЛИ. То есть, в качестве ключа можно брать lat или lon (одно из двух) и далее, должно идти число. Запустим программу вот с таким текстом:

text = "lat = 5, lon=7, a=5"

и на выходе получаем только данные для lat и lon:

['lat = 5', 'lon=7']

Но у этого выражения есть недостаток: мы дважды записываем одно и то же после имен ключей. Давайте это оптимизируем. Здесь нам на помощь приходят группирующие скобки:

match = re.findall(r"(?:lat|lon)**\s**\*=**\s**\***\d**+", text)

Мы здесь использовали несохраняющую группировку двух литералов lat или lon. Вот эти два символа ?: сразу после открывающей круглой скобки как раз и указывает на несохраняющую группировку. Что это такое я поясню чуть позже. А сейчас, если запустить программу, то увидим прежний результат:

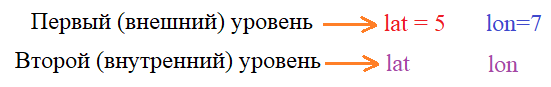
['lat = 5', 'lon=7']

и, при этом, наше правило записано без дублирующих элементов. Интерпретировать все это можно так: сначала проверяется наличие литералов lat или lon, а затем, после них должен стоять знак равенства и числа.

Теперь, давайте уберем символы ?: из наших скобок:

match = re.findall(r"(lat|lon)**\s**\*=**\s**\***\d**+", text)

В этом случае мы получаем сохраняющую группировку. То есть, для каждого найденного вхождения отдельно будет сохраняться соответствующий ключ. Визуально это можно представить так:



И, если запустить программу, то в консоли увидим как раз этот второй уровень:

['lat', 'lon']

Чтобы увидеть оба уровня, поставим сохраняющие скобки вокруг всего выражения:

match = re.findall(r"((lat|lon)\s\*=\s\*\d+)", text)

У нас получится следующий результат:

[('lat = 5', 'lat'), ('lon=7', 'lon')]

Фактически, здесь во второй уровень записывается сначала вся найденная фраза, а затем, включаются внутренние скобки и добавляются ключи. И, смотрите, используя этот функционал, мы можем отдельно выделить ключ и значения, записав шаблон в таком виде:

match = re.findall(r"(lat|lon)**\s**\*=**\s**\*(**\d**+)", text)

На выходе сразу получим список из кортежей с парами: ключ, значение:

[('lat', '5'), ('lon', '7')]

и это очень удобно для дальнейшей обработки этих данных. Вот так работают круглые скобки: они выполняют сразу две функции – группировку и сохранение результатов поиска. Или, же можем использовать группировку без сохранения, добавляя внутрь скобок символы

(?:<выражение>)

Давайте в качестве практического примера рассмотрим парсинг вот такого xml-файла:

<!DOCTYPE xmlmap>

<xmlmap>

 <parametrs>

  <name>map</name>

  <scale>750000</scale>

  <issuedate>20060828</issuedate>

  <correctiondate>20060828</correctiondate>

 </parametrs>

 <object code="43" >

  <attribute number="133" value="3000000" />

  <attribute number="174" value="20" />

  <primitive pointType="2" name="2" >

   <point lon="40.8482" lat="52.6274" />

   <point lon="40.8559" lat="52.6361" />

…

Нам здесь нужно у записи point выделить долготу (lon) и широту (lat). В самом простом варианте это можно сделать так:

**with** open("map.xml", "r") **as** f:

    lat = []

    lon = []

**for** text **in** f:

        match = re.findall(r"<point**\s**+[^>]\*?lon=([**\"\'**])([0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])([0-9.,]+)**\1**", text)

**print**(match)

**print**(lon, lat, sep="**\n**")

Мы здесь открываем файл на чтение с использованием менеджера контекста а, затем, построчно считываем из него информацию. Для каждой строки применяем регулярное выражение, выделяя атрибуты lon и lat. На выходе получаем следующий результат:

…  
[]  
[]  
[]  
[('"', '40.8482', '"', '52.6274')]  
[('"', '40.8559', '"', '52.6361')]  
[('"', '40.8614', '"', '52.651')]  
…

В тех строчках, где нет данных атрибутов, имеем пустую коллекцию, а где формат совпадает, получаем четыре значения из сохраняющих скобок. Здесь нам нужны значения с индексом 1 – для lon и 3 – для lat. И, записывая все «в лоб», получаем такую программу:

**with** open("map.xml", "r") **as** f:

    lat = []

    lon = []

**for** text **in** f:

        match = re.findall(r"<point**\s**+[^>]\*?lon=([**\"\'**])([0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])([0-9.,]+)**\1**", text)

**if** len(match) > 0:

            lon.append(match[0][1])

            lat.append(match[0][3])

**print**(lon, lat, sep="**\n**")

Но, как вы понимаете, это не лучшая реализация поставленной задачи. Что если регулярное выражение изменится и индексы станут другими? Придется поправлять весь программный код, связанный с этим шаблоном! Это не очень удобно. Здесь лучше использовать имена сохраняющих групп, а затем, обращаться к данным по этим именам. Поэтому перепишем программу так:

**with** open("map.xml", "r") **as** f:

    lat = []

    lon = []

**for** text **in** f:

        match = re.search(r"<point**\s**+[^>]\*?lon=([**\"\'**])(?P<lon>[0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])(?P<lat>[0-9.,]+)**\1**", text)

**if** match:

            v = match.groupdict()

**if** "lon" **in** v **and** "lat" **in** v:

                lon.append(v["lon"])

                lat.append(v["lat"])

**print**(lon, lat, sep="**\n**")

Мы здесь воспользовались другим методом модуля re – search, который возвращает не просто коллекцию, а объект, из которого можно получить словарь, содержащий коллекцию сохраненных именованных групп (об этом методе подробнее речь пойдет позже). Далее, мы проверяем: было ли найдено совпадение и если да, то дополнительная проверка на наличие ключей lon и lat в полученном словаре. При истинности обеих проверок, добавляем в наши коллекции соответствующие данные.

Здесь использование именованных групп оправдано, т.к. создает универсальность текста программы, независимо от вида регулярного выражения.