

2. Сбор данных со сторонних сайтов

Цель занятия

В результате обучения на этой неделе вы:

- узнаете, как собирать данные из интернета
- поймете, какие существуют способы для извлечения информации из интернета
- поймете, как получить данные с помощью регулярных выражений
- научитесь собирать данные с помощью регулярных выражений

План занятия

1. [Введение в обработку данных](#)
2. [Поиск с помощью регулярных выражений](#)
3. [Символьные классы и квантификаторы](#)
4. [Сложный поиск и замена](#)

Используемые термины

Краулинг (от английского слова crawl — ползать) — последовательный обход страниц сайта.

Парсинг (от английского слова parse — обрабатывать) — извлечение нужной пользователю информации со страницы.

DOM-дерево — вложенные друг в друга теги отображаются как дочерние узлы.

CSS-селекторы — способ определить теги, к которым нужно применить правило/вытащить информацию.

XPath-выражения — выражения, работающие с веб-страницей как с XML-документом.

Регулярное выражение (regular expression) — это формальный язык, который используется для поиска подстрок в тексте.

Регулярные выражения — формальный язык для поиска подстрок.

Группа — кусок совпадения, который можно отложить отдельно и далее с ним работать.

Конспект занятия

1. Введение в обработку данных

Веб-краулинг и парсинг

Предположим, нам нужно собрать лингвистический корпус. **Корпусом** мы называем собрание текстов, которые помимо самих текстов хранит разметку, нужную исследователям, и какие-то метаданные – данные о самом тексте.

Рассмотрим пример:

Старый дом раскроет тайны?



Евгений Стариков



711



Дом Панова в 1983 году.
Фото А. И. Финогенова

В областной столице завершаются работы по реставрации дома № 38 по улице Герцена. И хотя изучение этого дома-памятника XIX века специалистами насчитывает как минимум четыре десятка лет, до сих пор не удалось ответить на многие вопросы истории здания. Более того, новая реставрация загадала новые загадки!

Три даты

В 2021 году предприниматель Герман Якимов приобрел объект культурного наследия федерального значения дом №38 по улице Герцена с обязательством взять на себя расходы по его реставрации и содержанию. На первом этаже планируется открыть цветочный магазин, на втором - культурный центр. Ранее меценат инвестировал средства в четыре памятника деревянного зодчества: «Дом с лилиями» и Дом Извощикова на улице Чернышевского, Дом Дружинина на Мальцева. Продолжается реставрация и в арендованном предпринимателем Доме Засецких. Руководителем работ на всех

Какие данные и метаданные мы можем отсюда получить?

1. Название.
2. Количество просмотров
3. Сам текст
4. Имя автора

Страниц на сайтах очень много. Если мы будем ходить на каждую из них вручную, копировать и вставлять в разные файлы, это займет слишком много времени.

Это все можно конечно автоматизировать.

Краулинг — это последовательный обход страниц сайта. От английского слова crawl — ползать.

Парсинг — извлечение нужной пользователю информации со страницы. От английского слова parse — обрабатывать.

Это два взаимосвязанных действия: вначале мы получаем страницу, затем обрабатываем ее содержимое.

Как это сделать?

- Использовать готовые краулеры
 - + проще освоить
 - + есть готовые примеры
 - + много опций на вкус и цвет
 - сложно настроить для своей задачи
 - фиксированные форматы выгрузки данных
- Написать код
 - + гибкость решения
 - + можно полностью подстроить под себя
 - возможно, это сложнее

Парсинг в Python

Алгоритм:

1. Получаем содержимое страницы
2. Придумываем, как извлечь нужное
3. Извлекаем и сохраняем в нужном формате

Инструменты, при помощи которых это можно сделать:

1. Библиотека requests
2. Извлечение информации:
 - селекторы
 - XPath выражения
 - регулярные выражения
3. Сохранить данные можно с помощью модуля os и стандартных библиотек Python

Селекторы

Контент страницы состоит из HTML-элементов:

```
<!DOCTYPE html>
<html>

<head>
  <title>Our Company</title>
</head>

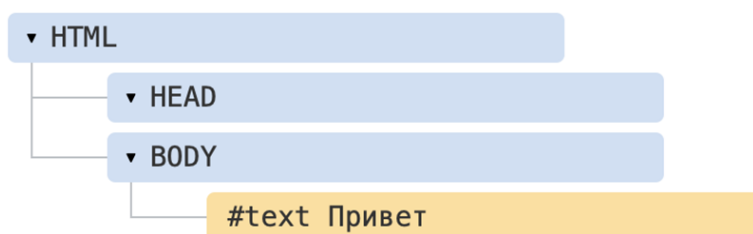
<body>

  <h1>Welcome to Our Company</h1>
  <h2>Web Site Main Ingredients:</h2>

  <p>Pages (HTML)</p>
  <p>Style Sheets (CSS)</p>
  <p>Computer Code (JavaScript)</p>
  <p>Live Data (Files and Databases)</p>

</body>
</html>
```

Теги вложены друг в друга. Таким образом, выстраивается некоторая иерархичность. Теги собираются в **ДОМ-дерево** — вложенные друг в друга теги отображаются как дочерние узлы.



CSS-селекторы — способ определить теги, к которым нужно применить правило/вытащить информацию. Изначально были придуманы для того, чтобы отделить контент страницы, который хранится в HTML файле, от оформления страницы, которая хранится в css файле. Css – каскадная таблица стилей. Это отдельные файлы. Селекторы работают с HTML тегами.

XPath-выражения — выражения, работающие с веб-страницей как с XML-документом. XML – расширенный язык разметки. В отличие от HTML XML формат завязан на хранении и отправке данных. HTML завязан на представление данных в окне браузера. XPath-выражения работают с HTML файлом, как с XML файлом:

- детальнее селекторов
- медленнее выполняются

Регулярные выражения

Регулярное выражение (regular expression) — это формальный язык, который используется для поиска подстрок в тексте.

Не обязательно думать о строке, как о строке текста. В рамках регулярных выражений мы можем думать про весь HTML документ, который мы получили, как про единую строку.

Например, regex для поиска фамилии Каддафи выглядел бы так:

```
\b(Kh?|Gh?|Qu?)[aeu](d['dt]?|t|zz|dhd)h?aff?[iy]\b
```

Арабский язык не очень хорошо транскрибируется в латиницу, поэтому появляется много вариантов. Если развернуть это регулярное выражение, получится такая картина:

$$M \begin{cases} u \\ o \end{cases} \begin{cases} \emptyset \\ ' \end{cases} a \begin{cases} mm \\ m \end{cases} \begin{cases} a \\ e \end{cases} r \begin{cases} al \\ el \\ Al \\ El \\ \emptyset \end{cases} \begin{cases} - \\ _ \\ \emptyset \end{cases} \begin{cases} Q \\ G \\ Gh \\ K \\ Kh \end{cases} \begin{cases} a \\ e \\ u \end{cases} \begin{cases} d \\ dh \\ dd \\ ddh \\ dhdh \\ dth \\ th \\ zz \end{cases} a \begin{cases} f \\ ff \end{cases} \begin{cases} i \\ y \end{cases}$$

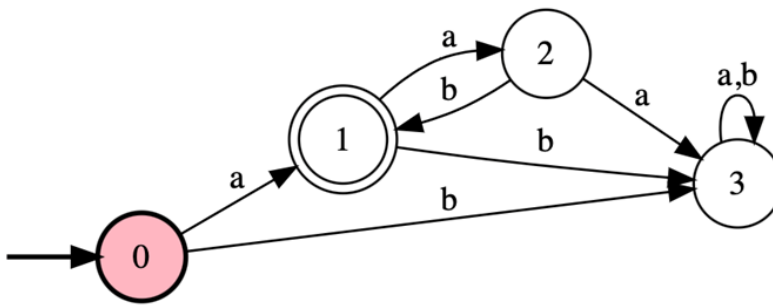
На каждую позицию существует множество вариантов. Всех их регулярные выражения учитывают.

2. Поиск с помощью регулярных выражений

Что такое регулярные выражения

Регулярные выражения — это формальный язык для поиска подстрок.

Регулярные выражения довольно тесно связаны с математикой. Концепция: конечный автомат.



Каждый кружок – некоторое состояние, в котором мы можем находиться. На ребрах графа расположены буквы. Путешествуя из одной вершины в другую, можно собирать различные слова.

Все слова, которые мы можем собрать при помощи конечного автомата, описываются регулярным выражением:

$$a(ab)^*$$

Алгоритм создания регулярного выражения

- Представляем в голове шаблон. По какой маске будем искать подстроки.
- Разбиваем его на части: изменяющиеся и неизменяющиеся. Неизменяющуюся часть мы оставляем, как есть.
- Каждую часть записываем с использованием специальных символов регулярных выражений.

Язык регулярных выражений

Язык	Значение
abc	символы abc подряд
$[abc]$	любой из символов a, b, c
$.$	любой символ
$()$	группа
$a - z$	символы от a до z
$a?$	a или отсутствие a
$a b$	a или b
$^$	начало строки

\$	конец строки
----	--------------

Пример. Все вхождения слова кот.

Лапка **кота** принадлежит **коту**, и **кот** часто использует эту лапку, чтобы умыться.

Неизменяемая часть: **кот** в разных падежах с разными окончаниями.

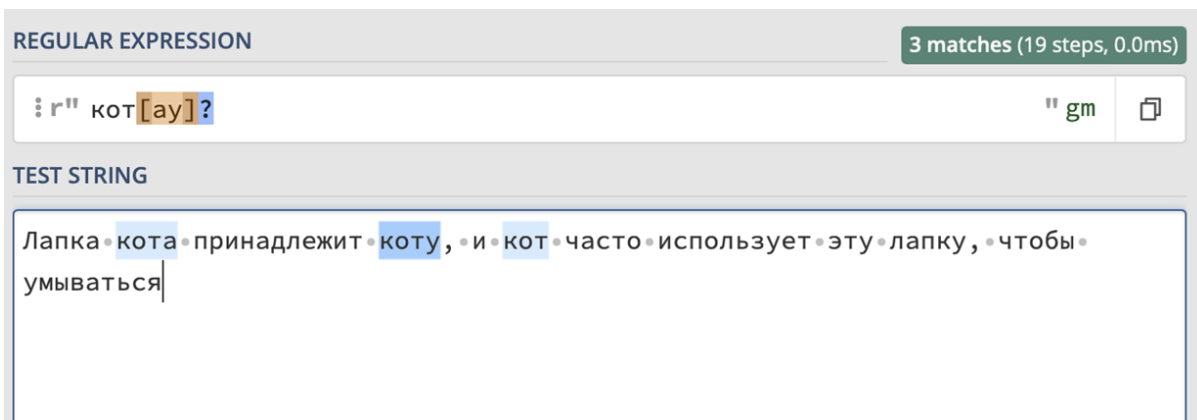
Окончания:

а, у, нулевое \Rightarrow одно из а/у или их отсутствие \Rightarrow [ау]?

Готовое регулярное выражение:

кот[ау]?

Проверим его на каком-нибудь сайте или в среде программирования.



The screenshot shows a web-based regex testing interface. At the top, it says 'REGULAR EXPRESSION' and '3 matches (19 steps, 0.0ms)'. The input field contains the regex `кот[ау]?`. Below, the 'TEST STRING' section shows the text: 'Лапка •кота• принадлежит •коту•, •и• кот• часто• использует• эту• лапку, •чтобы• умываться|'. The words 'кота', 'коту', and 'кот' are highlighted in blue, indicating successful matches.

Для проверки рекомендуется сайт regex101.com.

Проверим еще в среде программирования.

Регулярные выражения в Python

Модуль `re` встроен в Python, устанавливать его не нужно:

```
import re
```

Основные методы:

- **re.compile(regex)** — «собрать» регулярное выражение (удобно, если нужно применить `regex` много раз). Предподготовка компиляции регулярного выражения. Как аргумент принимает выражение, записанное в строку. Готовит из него некоторый полуфабрикат, который в дальнейшем можно применить к остальным строкам, для которых будем производить поиск.
- **re.search(regex, string)** — применить регулярное выражение к строке, находит первое вхождение.

- `re.findall(regex, string)` – найти все вхождения в строке.

Пример. Извлечение времени из текста.

Ввод:

В 8:00 я выключаю будильник. В 8:05 – второй, в 8:30 – третий... и так до 10:00, когда оказывается, что спать больше нельзя.

Часы могут быть записаны как одним числом, так и двумя. Если два числа, то первое – либо единица, либо отсутствует вовсе. Минуты записаны всегда двумя числами. Первое число в диапазоне от 0 до 5, второе – от 0 до 9.

```
import re

line = input()

time_str = "[12]?[0-9]:[0-5][0-9]"
regex_time = re.compile(time_str)

for time in re.findall(regex_time, line):
    print(time)
```

>>> 8:00
8:05
8:30
10:00

Приведенное в коде регулярное выражение допускает такие варианты, как 29 часов. Это не совсем корректно. Но 29 часов вряд ли попадутся в нашей строке. Будем считать, что написанное выражение – некоторая эвристика. Этого достаточно для решения задачи, но не идеально.

3. Символьные классы и квантификаторы

Пример 1. Смена формата телефонного номера.

Дано: номер в формате +XYZ XX XXX-XX-XX.

Нужно: номер в формате XYZXXXXXXXXXX.

Решение: извлечь все числа с помощью регулярного выражения

```
re.findall("[0-9]", "+41 78 227-18-19")
```

>>> ['4', '1', '7', '8', '2', '2', '7', '1', '8', '1', '9']

Получаем список, в котором каждое число – отдельный элемент. Можно соединить их между собой при помощи символов пустой строки и получить единую строку:

```
"".join(re.findall("[0-9]", "+41 78 227-18-19"))  
>>> '41782271819'
```

Но это не совсем верный способ. Можно сделать быстрее.

Символьные классы

Чтобы не писать много вариантов символов, которые по своей сути означают одно и то же, можно воспользоваться специальными знаками, которые отвечают сразу за все знаки этого класса:

Символьный класс	Значение
\d	числа
\D	не числа (все, кроме \d)
\s	пробельные символы (пробел, перенос строки, табуляция)
\S	непробельные символы (всё, кроме \s)
\w	«буквы»: латиница, цифры, знак _
\W	не-«буквы» (всё, кроме \w)

Символьный класс, в котором записана строчная буква, это сам класс. Символьный класс с заглавной буквы – это все, кроме представителей самого этого класса.

В \w кириллица не входит, её придется перечислять при помощи диапазона.

Квантификаторы

Часто бывает, что символьные классы нам нужны в каком-то количестве. Например, три цифры подряд. Для этого нам понадобятся квантификаторы.

Квантификаторы можно воспринимать, как знак умножения. Они отвечают за то, сколько раз символ повторяется.

Квантификатор	Значение
{a, b}	повторение от a до b раз (включительно)
?	повторение 0 или 1 раз

*	повторение сколько угодно раз (включая 0)
+	повторение 1 и более раз

Из-за того, что у квантификаторов * и + отсутствует верхняя граница, появляется **жадность квантификаторов**.

Представим себе строку:

я люблю торты "Медовик" и "Наполеон"

Регулярное выражение `\".*\"` (все, что находится между знаками кавычек) вернёт такое совпадение:

"Медовик" и "Наполеон"

Обратный слеш в регулярном выражении `\".*\"` называется техникой экранирования. В Python кавычки означают начало и конец предложения. Чтобы указать, что это кавычка, как знак, мы ставим перед ней знак обратного слеша.

Как получить "Медовик" и "Наполеон" отдельно? Нам нужно ограничить жадность квантификатора.

По умолчанию квантификаторы * и + считаются жадными — они остановятся на последнем найденном символе. Чтобы ограничить их, достаточно поставить знак ? перед ограничивающим символом:

`\".*\"` → `\".*?\"`

Применим новое регулярное выражение к тексту.

```
re.findall("\".*?\\"", "я люблю торты \"Медовик\" и \"Наполеон\"")
>>> ["Медовик", "Наполеон"]
```

Примеры на текстовых данных

Вернемся к примеру с телефонным номером. Теперь, когда мы знаем о существовании символьных классов,

```
re.findall("\d+", "+41 78 227-18-19")
>>> '41', '78', '227', '18', '19'
```

```
"".join(re.findall("\d+", "+41 78 227-18-19"))
>>> '41782271819'
```

Рассмотрим другой пример.

Пример 2. Пусть у нас есть некоторый текст на английском языке. Нам нужно разбить его по словам. В рамках этой задачи будем считать, что слово – последовательность символов, разделенная пробелом или другим символом. В английском языке используется только латиница. Поэтому все, что не является латиницей, для нас будет символом, по которому мы будем разделять текст.

```
re.split('\W+', 'All sentences include two parts: the subject and  
the verb (this is also known as the predicate).')
```

```
>>> ['All', 'sentences', 'include', 'two', 'parts', 'the', 'subject', 'and', 'the', 'verb', 'this', 'is', 'also',  
'known', 'as', 'the', 'predicate', '']
```

Пример 3. Выбрать из текста все e-mail адреса:

```
re.findall("[\w\.-]+@[ \w\.-]+\.\w+", "Почты бывают разные:  
email.with.dots@domain.com, email@domain-with-dashes.com,  
email-dash@domain-dash.ai")
```

```
>>> ['email.with.dots@domain.com', 'email@domain-with-dashes.com',  
'email-dash@domain-dash.ai']
```

Домашнее задание. Регулярное выражение из примера 3 можно усовершенствовать. Иногда бывают e-mail адреса, в которых несколько доменов подряд.

Регулярные выражения, группы и HTML

Рассмотрим, как мы можем использовать регулярные выражения для извлечения информации из HTML документов.

Пример 4. Выбрать имя автора из HTML-кода статьи в интернет-газете:

```
<div class="author-name">  
  <a class="italic" href="/journalist/2">Ольга Ильинская</a>  
  <a  
    class="author-email small italic"  
    href="mailto:iljinskaja@krasseever.ru"  
  >iljinskaja@krasseever.ru</a>  
>  
</div>
```

Рассмотрим строку

```
<a class="italic" href="/journalist/2">Ольга Ильинская</a>
```

Что будет меняться в строке:

- ссылка в href — нам она не нужна
- имя автора — его и извлечем

Что не будет меняться:

- открывающий тег
- классы
- закрывающий тег

Регулярное выражение:

```
<a class=\"italic\" href=\".*?\">(.*?)?</a>
```

Красным выделены неизменяющиеся части. Мы экранировали символы кавычек.

Рассмотрим изменяющуюся часть `.*?`

`href=\".*?\"` — ограничиваем последовательность символов до первого `"`

`(.*?)?<` — ограничиваем последовательность до первого `<`, запоминаем последовательность как **группу** (кусочек совпадения, который мы можем отложить отдельно и далее с ним работать).

Получим такой код:

```
html = "<a class=\"italic\" href=\"/journalist/2\">Ольга
Ильинская</a>"

regex_name = re.compile("<a class=\"italic\"
href=\\..*?\">(.*?)?</a>")

author_name = re.search(regex_name, html)

print(author_name)

> <a class="italic" href="/journalist/2">Ольга Ильинская</a>

print(author_name.group(1))

>>> Ольга Ильинская
```

У регулярных выражений есть ограничение на количество групп. Их всего может быть 6.

Существует нулевая группа. Это все совпадение, которое у нас появилось. То есть нулевая группа будет эквивалентна тому, что мы просто выведем совпадения, которые у нас сейчас находятся в переменной `author_name`.

Домашнее задание. Написать регулярное выражение, которое будет учитывать кусочек тега и какую-то информацию внутри этого тега. Поместив её в группу, ее можно извлечь и сохранить как нужно.

4. Сложный поиск и замена

Попробуем написать различные регулярные выражения. Попрактикуемся на двух текстах: перевод драмы Шекспира “Гамлет” на язык эсперанто, и поработаем с файлом текста романа Ulysses.

```
import re
```

Гамлет

Структура текста перевода “Гамлет” на эсперанто:

1. Техническая часть
2. Название текста, описание, имя переводчика
3. Список персонажей
4. Текст. Все акты и сцены подписаны.

Все цифры в тексте римские.

Напишем такое регулярное выражение, которое бы доставало все упоминания актов и сцен с их порядковыми номерами.

```
import re

with open("hamleto_esperanto.txt", "r", encoding="utf-8") as f:
    hamlet_raw = f.read()
    re_act = re.compile("AKTO [IVX]+")
    for act in re.findall(re_act, hamlet_raw):
        print(act)
```

Если мы хотим вывести и акт, и сцену отдельно

```
import re

with open("hamleto_esperanto.txt", "r", encoding="utf-8") as f:
    hamlet_raw = f.read()
    re_act_scene = re.compile("(?: AKTO|SCENO) [IVX]+")
```

```
for act in re.findall(re_act_scene, hamlet_raw):  
    print(act)
```

Допустим, мы хотим складывать акты и сцены в разные списки.

```
import re  
  
with open("hamleto_esperanto.txt", "r", encoding="utf-8") as f:  
    hamlet_raw = f.read()  
re_act_scene = re.compile("(AKTO|SCENO) [IVX]+")  
scenes_dict = {}  
scenes_cnt = 0  
for act in re.finditer(re_act_scene, hamlet_raw):  
    if act.group(1) == "SCENO":  
        scenes_dict[act.group(0)] = scenes_cnt  
        scenes_cnt += 1  
print(scenes_dict)
```

Теперь попробуем посмотреть что-нибудь, связанное с авторскими ремарками в тексте. Это некоторый текст, который находится внутри скобок. При этом текст может занимать всю строку, может – часть строки, а возможно несколько строк.

Напишем регулярное выражение, которое будет выявлять авторские ремарки.

```
import re  
  
with open("hamleto_esperanto.txt", "r", encoding="utf-8") as f:  
    hamlet_raw = f.read()  
  
re_stage = re.compile("\\(.*?\\)", flags=re.MULTILINE)  
for stage in re.findall(re_stage, hamlet_raw):  
    print(stage)
```

flags – некоторое дополнительное расширение идеи регулярных выражений, которое позволяет нам изменить паттерн. Здесь мы хотим, чтоб регулярное выражение применялось на несколько строк подряд.

Теперь можно составить словарь и посмотреть, к кому чаще всего обращается автор в ремарках.

```
import re  
  
with open("hamleto_esperanto.txt", "r", encoding="utf-8") as f:  
    hamlet_raw = f.read()
```

```
re_stage = re.compile("\([Aa]1 ([A-Z]\w+)\)")
name_freq = {}
for name in re.findall(re_stage, hamlet_raw):
    name_freq[name] = name_freq.get(name, 0) + 1
print(name_freq)
```

Можно вывести самое большое количество обращений

```
print(max(name_freq.values()))
```

Ulysses

Теперь рассмотрим роман "Ulysses".

В этом романе есть список всех глав. Попробуем написать регулярное выражение, которое будет извлекать ссылки и делать из них корректные URL ссылки.

```
import re
base_url = "https://www.gutenberg.org/cache/epub/4300/pg4300-images.html"
with open("ulysses.html", "r", encoding="utf-8") as f:
    ulysses_raw = f.read()
re_chapter_link = re.compile("<a href=\"(#\w{4}\d{2})\">(.*?)</a>",
                             flags=re.MULTILINE)
for match in re.finditer(re_chapter_link, ulysses_raw):
    chapter_name = match.group(2)
    chapter_link = base_url + match.group(1)
    print(f"{chapter_name}\n\t{chapter_link}")
```

На сайте regex101 собираем регулярные выражения:

```
<a href=\"(#\w{4}\d{2})\">(.*?)</a>
```

Попробуем избавиться от всех непонятных тегов, которые мешают восприятию. Любой тег - комбинация любых признаков, которые находятся между двумя угловыми скобками <>.

```
import re
base_url = "https://www.gutenberg.org/cache/epub/4300/pg4300-images.html"
with open("ulysses.html", "r", encoding="utf-8") as f:
    ulysses_raw = f.read()
re_chapter_link = re.compile("<a href=\"(#\w{4}\d{2})\">(.*?)</a>",
                             flags=re.MULTILINE)
for match in re.finditer(re_chapter_link, ulysses_raw):
    chapter_name = re.sub("<.*?>", '',
                          match.group(2)).replace("&mdash;", "-")
```

```
chapter_link = base_url + match.group(1)
print(f"{chapter_name}\n\t{chapter_link}")
```

Получаем, что названия глав в итоге заключены между двумя тире, а названия частей глав заключены в квадратные скобки.

Дополнительные материалы для самостоятельного изучения

1. <https://regex101.com/>
2. <https://www.gutenberg.org/cache/epub/4300/pg4300-images.html>