Lesson Overview

You have done it! You have made it to the last lesson! Congratulations, we are so proud of you!

Now, it is time to show us, what you have learned, because we are about to scrape!

Scraping is:

- a method of getting various data from different web pages.
- a bit more difficult, but it's a must-have skill of every experienced programmer.

To be able to scrape the web, in this lesson we'll learn about:

- what are and how to work with **HTML files**
- the basics of **internet communication**.

Our in-person lessons are over, but you can start with our **3rd big project**. This time, you will use scraping. Good luck and keep in touch, we want to know, if you could do it.

Osnova

00:43

REVIEW EXERCISES

Before lesson 12

100% z Lekce 14

Osnova

- check solution code from **home exercise in lesson 11 together**,
- have time to **answer any additional questions!**

Feel free to ask us anything. Lesson 12 will be effective only if you understand majority of topics from lessons 1-11. :)

HTML

What is HTML?

100% z Lekce 14

Osnova

made of so called **tags**. Each tag tells us, where a given page element ple of tags can be paragpraphs, headings, tables etc.

Below we have some basic HTML markup:

```
<body>
    <h1>This is a heading</h1>
    <p>And this is the paragraph number 1</p>

    <p> This is the paragraph number 2</p>
</body>
```

Browsers do not display the HTML tags, but use them to render the content of the page.

HTML Elements

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / HTML / HTML ELEMENTS

HTML is composed of elements and elements are composed of:

1. **opening tag**
2. **element content (text) - not always**
3. **closing tag**

So structurally an element could look like this

```
<opening_tag>Element Content</closing_tag>
```

The end tag is written like the start tag, but with a forward slash inserted before the tag name.

Example of an element could be a paragraph:

```
<p>This is some lorem ipsum content</p>
```

100% z Lekce 14

break (`\n` in Python), `<img>` (image placeholder).

Osnova

Tag Attributes

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / HTML / TAG ATTRIBUTES

Quite often in real-life HTML, tags are not alone in their brackets. They are followed by so called **attributes**.

Example of a tag, with attribute:

```
<a href="https://www.google.com/" target="_blank">Go to Google</a>
```

Attributes provide additional information about a given element. In the example above, we have created a hyperlink using `<a>` (anchor) tag. In order the link to work properly, we need to tell it, where it should lead. And that is the `href` **attribute**, which provides this information.

The above link looks on a web page as follows:

Go to Google

Besides the `href` attribute, we have included also the `target` attribute with value `"_blank"`, which causes the page to be opened in a new browser tab.

We should remember that attributes are **always specified in the start tag** and that they usually come in `name=value` pairs.

HTML Document Organization

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / HTML / HTML DOCUMENT ORGANIZATION

HTML's main task is to provide a structure to the information in the document. The structure is

100% z Lekce 14

The c... is ...en by the element nesting. Elements nested inside another element are c... ... element. Nesting takes place if:

Osnova

- the parent element's opening tag precedes all the children elements opening tags and
- the parent closing tag follows all the children closing tags

Example of parent-child relation could be that paragraphs and headers are enclosed in HTML document body:

```
<body>
    <h1>This is a heading</h1>
        <p>And this is the paragraph number 1</p>

        <p> This is the paragraph number 2</>
</body>
```

HTML Comments

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / HTML / HTML COMMENTS

HTML code, that is enclosed inside `<!--` and `-->` symbols will be ignored when building the web page. These markers serve as HTML comments. Therefore, the following HTML document will contain only one paragraph - the first one:

```
<body>
    <h1>This is a heading</h1>
        <p>And this is the paragraph number 1</p>

        <!--<p> This is the paragraph number 2<p/>-->
</body>
```

Osnova

arkup languages - for example XML, Tex, but HTML is the most widely used - it is the base of all the web pages on the internet.

We have presented you only with the most basic terms and document organization in order you better understand the parsing operations, we will perform with HTML documents downloaded from the web.

As there is much more to learn about HTML we recommend you to check the following resources, if you are interested in the topic:

1. W3SCHOOLS reference page
2. Mozilla Developer Network

INTERNET COMMUNICATION

100% z Lekce 14

Osnova

Work

Internet is a network of interconnected machines, which we can divide into:

1. **CLIENTS** - machines (f.e. web browser on you computer) that ask for resources (web pages, files, etc.)
2. **ROUTERS** or SWITCHES - machines that manage the traffic on the internet (moving data to correct destination)
3. **SERVERS** - machines that answer the requests and provide resources

The communication among all these machines is directed by many rules on different levels. These rules make up variouse **protocols**. Protocols describe, how set of specific tasks should be performed - what has to be done, in what order.

As we will be learning to request so called hypertext documents from servers in this section, the protocol of our interest is called HTTP (**H**yper**T**ext **T**ransfer **P**rotocol). We will describe it in a bit more detail in the following section.

HTTP Protocol

HTTP stands for Hypertext Transfer Protocol. It was designed to allow transfer of web pages, that consist of various resources:

1. HTML
2. CSS
3. JavaScript
4. JSON
5. Video

All this information is **requested** by the **Client** computer (e.g. through a web browser). The request is routed to a computer called **Server** that handles the request and whose task is to send back the requested resources in form of **response**.

Also, **Clients** do not have to always request a web page. They sometimes send data to **Servers** too. An example is sending a web form - registering or signing into a web page.

The above described actions of requesting or sending data are performed based on HTTP request methods and we will discuss them in the following section.

Request

The request content sent to a server depends on the HTTP request method used. The request content is packed into so called header and usually contains information what we are asking for, who is asking, how the data should be transferred (HTTP method) and more.

Response

On the other hand, the server response usually contains:

1. Header - additional information about the response - its length, encoding and many many more attributes
2. Body (sometimes not included) - contains the requested resource - html, css, JSON etc.

URL or URI

We are probably used to hear the term **URL** more often than the term **URI**. Acually, there is another term, that we hear even less often - **URN**. So what is the difference among the two? Or is there any?

URI (Uniform Resource Identifier)

Osnova

It is a term for a sequence of characters, that identify a given resource (where can it be found). URI is a general term, that embraces both URL and URN.

URL (Uniform Resource Locator)

URL is a sequence, that contains not only the location of the given resource (which server, what folder, etc.) but it tells us, what protocol has to be used to retrieve that resource (e.g. image).

So the `http://example.org/image1.png` URL is different from `ftp://example.org/image1.png`. The first URL specifies HTTP and the other FTP protocols, that have to be used to access `image.png`.

URN (Uniform Resource Name)

The URN provides a resource's identity, but no information about the protocol used to find it. Example of URN is therefore `example.org/image1.png`.

In our web browser, we usually use URL to tell the browser, what protocol to use and where to find the given resource.

URL Scheme

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / INTERNET COMMUNICATION / URL SCHEME

To understand URL better, let's look at its scheme. For our purposes, we will not include a complete scheme, rather a typical scheme we may encounter most often:

```
protocol://host/path?query#fragment
```

Example:

```
https://api.openweathermap.org/data/2.5/weather?q=London
```

100% z Lekce 14

Osnova	Example	Description
protocol	`https`	Describes, how the client and the server will communicate
host	`api.openweathermap.org`	This is the name under which the server can be found
path	`data/2.5/weather`	Once the serving machine is found, we need to locate the resource on its file system.
query	`q=London`	An optional sequence of **key=value** pairs separated by ampersand `&`. These are an analogy of arguments passed to a function. In this example, we are asking for data related to value London.
fragment (not present here)	`#chapter_5`	Name that starts with a character `#` references a given place in a requested document. For example a reference to a specific section on the same web page.

For more in depth information on URL syntax, check the wikipedia page dedicated to the topic

HTTP Status Codes

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / INTERNET COMMUNICATION / HTTP STATUS CODES

HTTP Status code is included in server's response to the client's request. It is a 3 digit number

There are [illegible] codes describe and therefore we will only list the groups into which they [illegible].

1xx: Informational

Request received, continuing process. We will not see this one very often.

2xx: Success

The action was successfully received, understood, and accepted. Code, that everybody, who sends a request wants, is code 200. It tells the client, that request has been successfully processed.

3xx: Redirection

Further action must be taken in order to complete the request. In reality, server tells the client, that the required resource can be found at different URL. Examples of codes are 302 and 303.

4xx: Client Error

The request contains bad syntax or cannot be fulfilled. The best known is the response 404, telling us that the server could not find the given resource.

5xx: Server Error

The server failed to fulfill an apparently valid request. Code 500 tells us, that due to the problem on the server side, the request could not be fulfilled.

To see more codes with more detailed description, you can check the wikipedia article dedicated to the subject.

HTTP Methods

HTTP eth s specify, what action a client requires from a server to be performed. Osnova H es ong other the following methods:

- **GET** - requests a given document or a video - in general a resource - this method is used to retrieve data
- **POST** - method used to send data to a server, e.g. for authentication or registration etc.
- **PUT** - similar to POST method in that we send some data to the server, but the server should update a given (existing) record or other resource to a new value, we send it
- **DELETE** - removes a given resource
- **HEAD** - asks for a response identical to that of a GET request, but without the response body
- **OPTIONS** - asks server, what methods are supported by a given resource - given web page etc.

For our purposes (to download data and to send data), we will be interested in GET and POST methods.

GET Request

If a client sends GET request, it is asking the server to send it the resource at a given URL. For example, the following is a GET request, that asks the server to send back whatever it can find at `httpbin.org//encoding/utf8` . The requesting client in this case is not a web browser, but a command line tool called **HTTPie**.

If you wish to install this tool, you can go here. Although we briefly use the tool here, it serves only as an example. The goal of this unit is not to talk about httpie but rather the **GET request**. Therefore we recommend, if you wish to use it more often, please learn how to use it in its documentation. You can of course choose only the parts of the documentation that will be of some use for **you** in particular.

```
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: close
User-Agent: HTTPie/0.9.9
Host: httpbin.org
```

We performed the above request, using HTTPie command line tool:

```
$ http GET httpbin.org/get
```

The response sent back from the server consists of:

- response header
- response body

Response header

The response header is separated from the body by a blank line. Header contains mutiple attributes telling us that:

- HTTP/1.1 200 OK - everything went well, because we get the response code 200
- Content-Length: 256 - the response body has 256 bytes
- etc.

```
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 256
Content-Type: application/json
Date: Mon, 26 Nov 2017 14:07:28 GMT
Server: gunicorn/19.9.0
Via: 1.1 vegur
```

The r[illegible]dy [illegible]ntains information a resource - in this case a json string. JSON string is [illegible]y. [illegible]y, who request the resource at this specific URL. This particular JSON string tells us:

- what attributes did the original GET request contain - `"headers"`
- ip address of the requesting machine - `"origin"`
- etc.

```
{
    "args": {},
    "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip, deflate",
        "Connection": "close",
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/0.9.9"
    },
    "origin": "85.132.197.69",
    "url": "http://httpbin.org/get"
}
```

Please note, that the capital GET word is important here. The httpbin.org site specifies page called [/get](/get), which returns the above listed response. It was the decision of the page creator to give it name **/get**.

POST Request

If a client sends POST request, it submits some data to the server. An example is filling the registration form and sending it to the server. Server then stores that information in the database.

An example of POST request header could look as follows:

```
Accept-Encoding: gzip, deflate
User-Agent: Osnova
Host: httpbin.org
Content-Type: application/json
Connection: close
name=John&surname=Smith
```

We performed the above request, using HTTPie command line tool:

```
$ http POST httpbin.org/post name='John' surname='Smith'
```

Again, the response is consists of:

- response header
- response body

Response header

The response header is separated from the body by a blank line. Header contains mutiple attributes telling us that:

- HTTP/1.1 200 OK - everything went well, because we get the response code 200
- Content-Length: 497 - the response body has 497 bytes
- etc.

```
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 497
```

Response body

The response body contains JSON, where we can see for example

- requesting machine - `"origin"`
- etc.

```
{
    "args": {},
    "data": "{\"name\": \"John\", \"surname\": \"Smith\"}",
    "files": {},
    "form": {},
    "headers": {
        "Accept": "application/json, */*",
        "Accept-Encoding": "gzip, deflate",
        "Connection": "close",
        "Content-Length": "36",
        "Content-Type": "application/json",
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/0.9.9"
    },
    "json": {
        "name": "John",
        "surname": "Smith"
    },
    "origin": "85.132.197.69",
    "url": "http://httpbin.org/post"
}
```

Please note, that the capitalized word POST is important here. The httpbin.org site specifies page called **/post**, which returns the above listed response. It was the decision of the page creator to give it name **/post**.

For More Information

Requests Library

To send requests from our programs to servers, we will use python module called **requests**. It is not part of the standard Python library and therefore we need to download it.

To install a package, that is not part of the standard library, we will use program called `pip` and we will invoke it from the terminal. This is how you should download your **requests** package:

```
$ pip install requests
```

From now on, you will be able to import module **requests** into your programs as follows:

```
import requests
```

This library is often used for so called web scraping, because of its simple use. We will learn to send GET and POST requests in the following two sections.

Downloading Data

To request a resource from a given URL (address), we need to use the `get()` method from the `requests` package:

```
>>> import requests
>>> response = requests.get('https://httpbin.org')
```

Once we have the response object, we can check, whether the resource have been returned

```
>>> s' 'us_code
    Osnova
```

Status code 200 tells us, that everything went well, and we have our response ready. To get an overview of all status codes, check the dedicated page on Mozilla Developer Network.

We can now have a look at the content of the response by using its `text` attribute:

```
>>> response.text
'<!DOCTYPE html>\n<html>\n<head>\n  <meta http-equiv=\'content-type\'
value=\'text/html;charset=utf8\'>\n  <meta name=\'generator\'
value=\'Ronn/v0.7.3 (http://github.com/rtomayko/ronn/tree/0.7.3)\'>\n
<title>httpbin(1):...
.....
.....
```

The response contains quite a lot of text, therefore we included only a part of it.

We can see, that the reponse contains HTML document. Some pages, however, do not return HTML, but they return JSON:

```
>>> response = requests.get('https://httpbin.org/get')
>>> response.status_code
200
>>> response.text
'{\n  "args": {}, \n  "headers": {\n    "Accept": "*/*", \n    "Accept-
Encoding": "gzip, deflate", \n    "Connection": "close", \n    "Host":
"httpbin.org", \n    "User-Agent": "python-requests/2.18.2"\n  }, \n
"origin": "85.132.197.69", \n  "url": "https://httpbin.org/get"\n}\n'
>>> response.json()
{'args': {}, 'origin': '85.132.197.69', 'url': 'https://httpbin.org/get',
'headers': {'Accept': '*/*', 'User-Agent': 'python-requests/2.18.2',
'Accept-Encoding': 'gzip, deflate', 'Host': 'httpbin.org', 'Connection':
'close'}}
>>>
```

In the example above, the response body stored inside the `text` attribute reminds of JSON

```
>>>                    json()
Osnova
>>>
'85.132.197.69'
```

Sending Data

To send data to the server, using POST method, we can use the `post()` method from the `requests` module.

```
>>> response = requests.post('http://httpbin.org/post', data =
{'my_message':'Hello'})
```

The response has been returned in form of JSON:

```
>>> response
<Response [200]>
>>> response.text
'{\n  "args": {}, \n  "data": "", \n  "files": {}, \n  "form": {\n
"my_message": "Hello"\n  }, \n  "headers": {\n    "Accept": "*/*", \n
"Accept-Encoding": "gzip, deflate", \n    "Connection": "close", \n
"Content-Length": "16", \n    "Content-Type": "application/x-www-form-
urlencoded", \n    "Host": "httpbin.org", \n    "User-Agent": "python-
requests/2.18.2"\n  }, \n  "json": null, \n  "origin": "85.132.197.69",
\n  "url": "http://httpbin.org/post"\n}\n'
>>> response.json()
{'form': {'my_message': 'Hello'}, 'origin': '85.132.197.69', 'json
Toto nie je dobre': None, 'headers': {'Content-Length': '16', 'Host':
'httpbin.org', 'User-Agent': 'python-requests/2.18.2', 'Content-Type':
'application/x-www-form-urlencoded', 'Accept': '*/*', 'Accept-Encoding':
'gzip, deflate', 'Connection': 'close'}, 'args': {}, 'files': {}, 'data':
'', 'url': 'http://httpbin.org/post'}
```

Osnova

HTTP, Requests

As in the previous section, you can further expand your knowledge of:

- **HTTP** - we recommend you check the Mozilla Developer Network section on HTTP,
- **requests package**, see its official documentation page.

BASICS OF HTML PARSING

Intro

HTML Parsing is a process of turning HTML string into a structure representation of the original HTML. Parsing HTML is often used in web scraping. Web scraping is extracting desired information from the internet. It is a vast area that requires knowledge of how web pages are built, data cleaning and much more. In this lesson, we will therefore only touch the topic introducing to you the most basic tool, we can use in Python for scraping the content of a web page.

This tool is a module called `bs4`. The acronym `bs` stands for Beautiful Soup, what is the name of a poem by Lewis Carroll, the author of Alice in Wonderland.

This module serves to parse html (as well as xml, which is another file format) and then to extract the information from it. Module `bs4` is a third party module and therefore we need to install it using the following command in our terminal:

```
$ pip install beautifulsoup4
```

Beautiful Soup Module

In the following few sections, we will learn the basics of how to parse and extract information from a html document. A typical workflow in which the `bs4` module is used is:

1. parse the html document by creating the soup (BeautifulSoup object)
2. extract the information from the soup

We can extract the information in various ways:

1. using methods, that will gather the target items (`find()`, `find_all()`, `select()`, etc) for us
2. looping across the document tree structure (`tag`, `children`, `descendands`, `parents`, `siblings`, etc.)

We will focus on `find_all()` method as well as on some basic navigation possibilities in the

Osnova HTML

Parsing is activity of converting messy html string, which we get back from the server, into organized html tree representation.

We will request the content of the web page `http://example.com/` , which we will see, looks a bit messy:

```
>>> r = requests.get("https://example.com")
>>> r
<Response [200]>
>>> r.text
'<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title>\n\n
<meta charset="utf-8" />\n    <meta http-equiv="Content-type"
content="text/html; charset=utf-8" />\n    <meta name="viewport"
content="width=device-width, initial-scale=1" />\n    <style
type="text/css">\n    body {\n        background-color: #f0f0f2;\n
margin: 0;\n        padding: 0;\n        font-family: "Open Sans",
"Helvetica Neue", Helvetica, Arial, sans-serif;\n        \n    }\n    div
{\n        width: 600px;\n        margin: 5em auto;\n        padding:
50px;\n        background-color: #fff;\n        border-radius: 1em;\n
}\n    a:link, a:visited {\n        color: #38488f;\n        text-
decoration: none;\n    }\n    @media (max-width: 700px) {\n        body
{\n            background-color: #fff;\n        }\n        div {\n
width: auto;\n            margin: 0 auto;\n            border-radius:
0;\n            padding: 1em;\n        }\n    }\n    </style>
\n</head>\n\n<body>\n<div>\n    <h1>Example Domain</h1>\n    <p>This
domain is established to be used for illustrative examples in documents.
You may use this\n    domain in examples without prior coordination or
asking for permission.</p>\n    <p><a

href="http://www.iana.org/domains/example">More information...</a>
</p>\n</div>\n</body>\n</html>\n'
```

Osnova

```
>>> from bs4 import BeautifulSoup as BS
```

Then we create the BeautifulSoup object from the contents of the `r.text` attrbute.

```
>>> soup = BS(r.text, "html.parser")
```

We can see that the so called class works as a function. We sent inside two arguments - the string we want to parse and the name of the parsing engine - `"html.parser"` . Now we can look, what is inside the `soup` variable:

```
>>> soup
<!DOCTYPE doctype html>
<html>
<head>
<title>Example Domain</title>
<meta charset="utf-8"/>
<meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
<meta content="width=device-width, initial-scale=1" name="viewport"/>
<style type="text/css">
    body {
....
        }
    }
    </style>
</head>
<body>
<div>
<h1>Example Domain</h1>

<p>This domain is established to be used for illustrative examples in
documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
<p><a href="http://www.iana.org/domains/example">More information...</a>
```

```
</body>
    Osnova
```

From the snippet above, it is not so visible that there are parent - children relationships among individual html elements. Understanding of these relationships is important for understanding of the document representation inside the `BeautifulSoup` object. We will therefore discuss this in the next section.

BeautifulSoup Object Structure

HTML documents are composed of HTML elements, that are organized in parent-child hierarchy. As one element, can have multiple children, the children among themselves have sibling relationship. Let's consider the following HTML:

- `<html>` element is the parent of `<head>` and `<body>` elements
- `<head>` and `<body>` are siblings
- `<body>` has its own child `<div>`, which has the following children: `<h1>`, `<p>`, `<p>`
- the second `<p>` element has another child `<a>` and `<a>` has finally its own child, the inner string `"More information..."`

```
<!DOCTYPE doctype html>
<html>
 <head>
  <title>
   Example Domain
  </title>
  <meta charset="utf-8"/>
  <meta content="text/html; charset=utf-8" http-equiv="Content-type"/
>
  <meta content="width=device-width, initial-scale=1" name="viewport"
/>
```

```
    Osnova
  <div>
   <h1>
    Example Domain
   </h1>
   <p>
    This domain is established to be used for illustrative examples i
n documents. You may use this
    domain in examples without prior coordination or asking for permi
ssion.
   </p>
   <p>
    <a href="http://www.iana.org/domains/example">
     More information...
    </a>
   </p>
  </div>
 </body>
</html>
```

The data in `BeautifulSoup` object is represented in this hierarchical manner, with the `BeautifulSoup` object at the top of the hierarchy, followed by `<html>`, `<head>` and `<body>` and so on.

Navigating the HTML

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / BASICS OF HTML PARSING / NAVIGATING THE HTML

Once we understand the parent - child and sibling relationships, we can navigate through the whole document tree. We will work, with the html from `https://example.com`

Selecting Element Directly by Tag Name

100% z Lekce 14

Osnova

ent by specifying its tag name as `soup` object's attribute. We will always a given name found in the hierarchy:

```
>>> soup.p
<p>This domain is established to be used for illustrative examples in
documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
```

```
>>> soup.a
<a href="http://www.iana.org/domains/example">More information...</a>
```

Moving on parent - child trajectory

1. contents vs children

Let's say, we would like to investigate the **direct children element** of the `<body>`. How would we do that? We select the `<body>` section by `soup.body` and then we can use its `contents` attribute:

```
>>> soup.body.contents
['\n', <div>
<h1>Example Domain</h1>
<p>This domain is established to be used for illustrative examples in
documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
<p><a href="http://www.iana.org/domains/example">More information...</a>
</p>
</div>, '\n']
```

There are 3 elements inside the returned list - two newline characters and one `<div>` element

```
>>> len(soup.body.contents)
3
```

We can get rid of newlines using a for loop:

```
...     if item == '\n':
            Osnova.body.contents.remove(item)
...
>>> soup.body.contents
[<div>
<h1>Example Domain</h1>
<p>This domain is established to be used for illustrative examples in
documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
<p><a href="http://www.iana.org/domains/example">More information...</a>
</p>
</div>]
```

We could also use a `filter()` function which we'll explain in more detail in the future lessons:

```
>>> list(filter(lambda x: x != "\n",soup.body.contents))
[<div>
<h1>Example Domain</h1>
<p>This domain is established to be used for illustrative examples in
documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
<p><a href="http://www.iana.org/domains/example">More information...</a>
</p>
</div>]
```

The `children` attribute is an iterator, which is very useful when dealing with huge html documents, where we do not want generate the whole list of children items at once. We therefore need to iterate over such object:

```
>>> for child in soup.body.children:
...     print(child)
...
<div>
<h1>Example Domain</h1>
```

```
      Osnova     ıples without prior coordination or asking for
permission.</p>
<p><a href="http://www.iana.org/domains/example">More information...</a>
</p>
</div>
```

2. descendants

The `descendants` attribute allows us to iterate over the whole descending tree, from the point where we are standing. For example, descendants of the `<div>` element in our html are all tags, that can be found between opening and closing
tags:

```
>>> for i,descendant in enumerate(soup.div.descendants):
...     print('DESCENDANT {}: {}'.format(i,descendant))
...
DESCENDANT 0:
DESCENDANT 1: <h1>Example Domain</h1>
DESCENDANT 2: Example Domain
DESCENDANT 3:
DESCENDANT 4: <p>This domain is established to be used for illustrative
examples in documents. You may use this
    domain in examples without prior coordination or asking for
permission.</p>
DESCENDANT 5: This domain is established to be used for illustrative
examples in documents. You may use this
    domain in examples without prior coordination or asking for
permission.
DESCENDANT 6:
DESCENDANT 7: <p><a href="http://www.iana.org/domains/example">More
information...</a></p>
DESCENDANT 8: <a href="http://www.iana.org/domains/example">More
information...</a>
DESCENDANT 9: More information...
DESCENDANT 10:
```

Osnova

ved downwards in the document hierarchy, but the same is possible by parent or parents attributes.

Moving between siblings

To move among the sibling elements, we can use `next_sibling`, `previous_sibling`, `next_siblings` and `previous_siblings` attributes.

See the module official documentation for more information about navigating across the parsed html document.

Extracting data from HTML

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / BASICS OF HTML PARSING / EXTRACTING DATA FROM HTML

When parsing the content of the page, we often know, what specific elements we want to target. For that purpose, the **bs4** module provides the following methods:

1. `find_all()` and its find-like derivates
2. `select()`

The `select()` method uses CSS selectors to target required elements. As we have not covered the subject of CSS selectors, we will not present use of this method here.

In this unit we will therefore discuss the method `find_all()`. This method can be used on any of the extracted tags from the soup object. It will then search the hierarchy from the given tag downwards and collect all the elements, that meet the given search criteria. The result is returned as a list of elements.

We will demonstrate the use of this method on html acquired from `https://example.com`.

Among others, we can **search by**:

100% z Lekce 14

3. [illegible]tic

Osnova

Searching by tag name or list of tag names

To search for a tag of a given name, we just pass the tag name as a string to the function call. Below we are retrieving all the `<p>` elements:

```
>>> soup.find_all('p')
[<p>This domain is established to be used for illustrative examples in
documents. You may use this

    domain in examples without prior coordination or asking for
permission.</p>, <p><a href="http://www.iana.org/domains/example">More
information...</a></p>]
```

Searching by attribute value or dictionary of attribute values

To extract elements based on associated attribute value, the method `find_all()` needs to be passed a keyword argument, where the parameter name represent the attribute name and the value is a string representing required attribute value:

```
>>> soup.find_all(charset="utf-8")
[<meta charset="utf-8"/>]
```

In case we would like to specify multiple attribute_name:value pairs or there are attributes, that **bs4** does not recognize, we need to enclose them inside a dictionary and pass them to `attrs` parameter:

```
>>> soup.find_all(attrs={"content":"text/html; charset=utf-8",'http-
equiv':"Content-type"})
[<meta content="text/html; charset=utf-8" http-equiv="Content-type"/>]
```

Searching by filtering function

The `find_all` function is in principle a filtering function. We can pass it another function that will return truthy or falsy value, for each element in the document.

Osnova

...tion, that returns value `True` if a given tag contains attribute `class` but ...ute `id`. And this is how it is used inside the call to `find_all()`:

```
>>> html = "<div class='red' id='bob'>This is the first div</div><div
class='green'>This is the second div</div>"
>>> small_soup = BS(html,"html.parser")
>>> small_soup.find_all(has_class_but_no_id)
[<div class="green">This is the second div</div>]
```

More Info - BS

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / BASICS OF HTML PARSING / MORE INFO - BS

Again, if you wish to explore **bs4** module further we recommend you visit its **documentation page**.

100% z Lekce 14

Osnova

Onsite html

Files structured using HTML are usually parsed by a third party library called **beautiful soup**. We can download it as follows:

```
$ pip install beautifulsoup4
```

And we import it as **bs4**. We actually import a class called **BeautifulSoup**.

```
>>> from bs4 import BeautifulSoup as BS
```

This class can be called as a function with input in form of the HTML string and parsing engine name. And it returns a BeautifulSoup object that represents the structure of the document:

```
>>> import requests
>>> r = requests.get('https://example.com')
>>> html = r.text
```

And here we parse the html string:

```
>>> soup = BS(html, "html.parser")
```

To understand the soup, we need to know 3 things:

Extracting information from the soup

What floats inside?:

- Tag objects
- Attribute objects

Osnova

Navigating across the soup tree

- children, contents, descendants
- parent, parents
- next sibling, previous sibling

Filtering targeted elements

- tag name

Examples

```
>>> soup.p
```

```
>>> soup.find('p')
```

```
>>> soup.find_all('p')
```

```
>>> soup.find_all(['h1','p'])
```

Tag attributes can be accessed through square bracket notation - as with dictionaries:

```
>>> a = soup.a
>>> a['href']
'http://www.iana.org/domains/example'
```

The overview of all the attributes a tag has, we use `attrs` attribute. **Attribute name** returns the `<a>` element - element that contains the given attribute. Searching by tag:

```
>>> a = soup.a
>>> a.attr
>>> a.attrs
{'href': 'http://www.iana.org/domains/example'}
```

100% z Lekce 14

```
>>> soup.find(href="http://www.iana.org/domains/example")
<a href="http://www.iana.org/domains/example">More information...</a>]
```

Osnova

Searching by tag and attribute name:

```
>>> soup.find('meta',{'http-equiv':"Content-type"})
<meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
```

We access the element content by the `string` attribute of a given tag object:

```
>>> a = soup.find(href="http://www.iana.org/domains/example")
>>> a.string
'More information...'
```

QUIZ

100% z Lekce 14

Osnova

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / QUIZ / HTML

1/7

What is html file?

A. Code that renders a web page

B. Text file that represents a web page in a hierarchical structure using tags

C. File that represents data in tabular format

D. File that stores information in dictionary-like format

Requests

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / QUIZ / REQUESTS

1/5

What is HTTP?

A. Programming language

100% z Lekce 14

Network protocol that describes how hypertext communication between client and server

Osnova

C. Type of network

D. Network protocol that describes how the so called packets are transferred across the Internet

HOME EXERCISES

Collect the Price of Gold

100% z Lekce 14

Osnova

ed to **scrape the data** out of the html string. We will collect information and then we will write the newly gained value and the current time into a csv file.

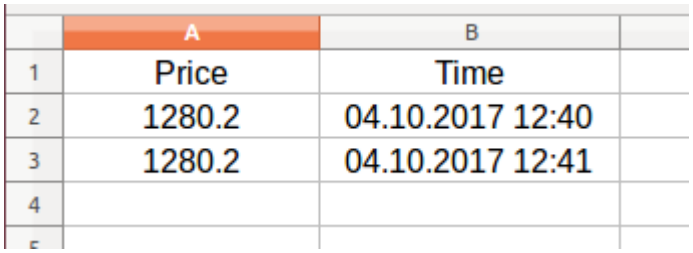
Your script should get the gold price from the Business Insider page.

Example of script in use:

```
$ python gold.py
The current price is 1,280.30
```

CSV file example:

```
Price, Time
1280.2,04.10.2017 12:40
1280.2,04.10.2017 12:41
1280.3,04.10.2017 12:45
```



	A	B
1	Price	Time
2	1280.2	04.10.2017 12:40
3	1280.2	04.10.2017 12:41
4		

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

```
import requests
import os
import csv
from datetime import datetime as dt
```

100% z Lekce 14

```
DATE_FORMAT = '%d.%m.%Y %H:%M'
URL = "https://markets.businessinsider.com/commodities/gold-price"

def request_gold_price():

    r = requests.get(URL)
    soup = BS(r.text, "html.parser")
    price = soup.find('span',{'class':'push-data'}).string

    print('The current price is {}'.format(price))

    price = ''.join([char for char in price if char!=','])

    return price

def write_data(price):

    mode = 'a' if 'gold_price.csv' in os.listdir() else 'w'

    with open('gold_price.csv', mode) as f:
        writer = csv.writer(f)

writer.writerow([float(price),dt.now().strftime(DATE_FORMAT)])


if __name__=="__main__":
    write_data(request_gold_price())
```

The script performs two main steps:

1. requests the web page, where the information about current gold price is available `request_gold_price()`
2. writes that information into a csv file

1. Function request_gold_price()

As besides making request and parsing the html, we have extracted the `<div>` element, that

Th ext ted the comma `','` from the price string in order it can be converted

Osnova

```
def request_gold_price():

    r = requests.get(URL)

    soup = BS(r.text, "html.parser")
    price = soup.find('div',{'class':'price'}).string

    print('The current price is {}'.format(price))

    price = ''.join([char for char in price if char!=','])

    return price
```

2. Function write_data()

We decided to specify `'a'` or `'w'` file mode to cover situations, when a file would already exist and we would like to rather append the information, than overwrite it entirely.

We have acquired information about the current date and time using datetime module's `now()` function. Subsequently, we have formatted the obtained datetime object using `strftime()` function.

Scrape Transparent Accounts

PYTHON ACADEMY / 12. INTRODUCTION TO SCRAPING / HOME EXERCISES / SCRAPE TRANSPARENT ACCOUNTS

Create a script that will **scrape any given transparent account** at FIO Bank

For example, we have decided to scrape this account.

100% z Lekce 14

Example script name:

Osnova

```
$ python scraping_aa.py
WRITTEN 171 LINES
RECEIVED 53182.12000000001 CZK
```

Our script has extracted all the transaction data and stored it in a csv file:

```
Datum,Částka,Typ,Název protiúčtu,Zpráva pro příjemce,KS,VS,SS,Poznámka
04.12.2018,1750.0000,Bezhotovostní příjem,KRČ ŠTEFAN,,0000,1122018,0,KRČ
ŠTEFAN
03.12.2018,-788.0000,Bezhotovostní platba,,"",,62935038,,Srazkova dan
03.12.2018,21330.0000,Bezhotovostní příjem,Štolba Tomáš,,0000,0,0,Štolba
Tomáš
....
....
```

Note that it is possible that the web site will not show all the movements, therefore you can adjust your script to request reasonable time periods.

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

```
1  import csv
2  import requests
3  import sys
4  from requests.exceptions import HTTPError
5  from bs4 import BeautifulSoup as BS
6  from datetime import datetime as dt
7  from pprint import pprint as pp
8
```

100% z Lekce 14

Osnova

```
            'f': dt(2017,1,1).strftime('%d.%m.%Y'),
             t': dt.today().strftime('%d.%m.%Y')}

def make_soup(URL,payload):
    try:
        r = requests.get(URL, params=payload)
        r.raise_for_status()
        soup = BS(r.text, "html.parser")
        return soup
    except HTTPError:
        print('Could not retrieve the page')
    except:
        print(sys.exc_info()[:1])


def extract_table(soup, target_header):
    # found by the content

    for table  in soup.find_all('table'):
        header = [child.string for child in table.thead.tr.children
if child.string != '\n']
        if header == target_header:
            return table



def extract_data(soup):
    # extracts transfers into a list of lists
    target_header = ['Datum', 'Částka', 'Typ', 'Název protiúčtu',
              'Zpráva pro příjemce', 'KS', 'VS', 'SS', 'Poznámka']

    table_to_scrape = extract_table(soup,target_header)

    transfers=[target_header]

    for row in filter(lambda x:
x!='\n',table_to_scrape.tbody.children):
        transfers.append([])
```

100% z Lekce 14

```
    for tr in rows)
```

Osnova

```
    return transfers


def write_data(data, filename):
    with open(filename,'w') as f:
        writer = csv.writer(f)
        writer.writerows(data)


    print('WRITTEN {} LINES'.format(len(data)))
    print('RECEIVED {}
CZK'.format(sum(map(float,extract_col(2,data[1:])))))

def extract_col(col_num,table):
    return [row[col_num-1] for row in table]

if __name__=="__main__":
    write_data(extract_data(make_soup(URL,payload)),
'Milos_Zeman.csv')
```

Explanation

We will begin our explanation from the last row of the above code solution:

```
if __name__=="__main__":
    write_data(extract_data(make_soup(URL,payload)),
'Milos_Zeman.csv')
```

This call tells us in what order are actions performed:

1. we first request and parse html - `make_soup()`
2. we extract the data from the **BeautifulSoup** object returned by the pervious call - `extract_data()`
3. we write the extracted data into a csv file - `write_data()`

This function sends request to a given URL and adds necessary parameters.

Osnova

URL parameters are as follows:

- `'a'` - account number
- `'f'` - the "from" date in format `'%d.%m.%Y'`
- `'t'` - the "to" date in format `'%d.%m.%Y')` . We used today's date for this parameter `dt.today()`

Once we obtain the response and not `HTTPError` has been raised, we parse the html string using **BeautifulSoup** class.

The function returns parsed html in form of a **BeautifulSoup** object

```
def make_soup(URL,payload):
    try:
        r = requests.get(URL, params=payload)
        r.raise_for_status()
        soup = BS(r.text, "html.parser")
        return soup
    except HTTPError:
        print('Could not retrieve the page')
    except:
        print(sys.exc_info()[:1])
```

Function extract_data()

This function's task is to return a list of list thus representing the table of transfers on the account. It extract the information from the `tbody` element's children elements. We have to be careful about filtering out all the newline characters as they are considered children of `tbody` too:

```
def extract_data(soup):
    # extracts transfers into a list of lists

    target_header = ['Datum', 'Částka', 'Typ', 'Název protiúčtu',
              'Zpráva pro příjemce', 'KS', 'VS', 'SS', 'Poznámka']
```

100% z Lekce 14

```
    ar  rs=[target_header]
```

Osnova

```
    for row in filter(lambda x: x!='\n',table_to_scrape.tbody.children):
        transfers.append([])
        for info in filter(lambda x: x!='\n',row.children):
            transfers[-1].append(info.attrs.get('data-value') or info.string)

    return transfers
```

As there are many tables on the web page, we need to identify the table that contain's the required data. The above function uses a helper function `extract_table()` to fulfill this task.

Function extract_table()

Identifies the target table based on the table header names. It keeps extracting and comparing headers of all the tables on the site, with the target header names.

```
def extract_table(soup, target_header):
    # found by the content

    for table  in soup.find_all('table'):
        header = [child.string for child in table.thead.tr.children if child.string != '\n']
        if header == target_header:
            return table
```

Function write_data()

Here we use context manager and `csv.writer` object to write the list of lists into a csv format.

```
def write_data(data, filename):
    with open(filename,'w') as f:
        writer = csv.writer(f)
```

100% z Lekce 14

```
    print('WRITTEN {} LINES'.format(len(data)))
else:
    print('RECEIVED {}
CZK'.format(sum(map(float,extract_col(2,data[1:])))))
```

Retrieve Weather Data

First, create an account at openweathermap.org

Then, create a script that will take parameters from command line and based on that will issue a request to openweathermap.org API.

Parameters

Users of my script are obliged to enter at least one (but if they want, they can enter more) of the following parameters

1) For location:

- `-id` : Location ID,
- `-q` : City_name | City_name,Country_code,
- `-lon` : Longitude,
- `-lat` : Latitude,

2) For forecast option `-o` the followig values are permitted

- `weather` - will provide information about the current weather
- `forecast` - will provide forecast for the following 5 days

An optional parameter `-cnt` is not always reflected by the API, but I have included it as well

Example of a report on current weather

Osnova

```
$ python weather.py -q Brno,cz -o weather
| DESCRIPTION | CLOUDS |  WIND  | TEMP_MAX | TEMP_MIN | HUMIDITY |
| few clouds  |  20%   | 2.6m/s | 13.0 °C  | 11.0 °C  |   76%    |
```

The information about the current weather `should be written into a csv file` called `weather_daily.csv` .

Handling incorrect user inputs

Incorrect required argument `-q` :

```
$ python weather.py -a Brno,cz
At least one of the following options is mandatory
('id', 'q', 'lon', 'lat')
USAGE: python weather.py <location> <weather/forecast> [<cnt limit>]
MANDATORY:
    <location> - at least one of the following
        -id:   Location ID,
        -q:    City_name | City_name,Country_code,
        -lon:  Longitude,
        -lat:  Latitude,
    <weather/forecast>
        -o:    One of the options: ("weather","forecast"),
OPTIONAL:
    <cnt limit>
        -cnt:  Limit of records to be returned
```

Missing required argument `-o`

```
$ python weather.py -q Brno,cz
At least one of the following options is mandatory
o
USAGE: python weather.py <location> <weather/forecast> [<cnt limit>]
MANDATORY:
```

```
                ty_name | City_name,Country_code,
      Osnova    ngitude,
        -lat:  Latitude,
    <weather/forecast>
        -o:    One of the options: ("weather","forecast"),
OPTIONAL:
    <cnt limit>
        -cnt:  Limit of records to be returned}
```

Missing data for a required argument `-o` :

```
$ python weather.py -q Brno,cz -o
Missing value for option -o
USAGE: python weather.py <location> <weather/forecast> [<cnt limit>]
MANDATORY:
    <location> - at least one of the following
        -id:   Location ID,
        -q:    City_name | City_name,Country_code,
        -lon:  Longitude,
        -lat:  Latitude,
    <weather/forecast>
        -o:    One of the options: ("weather","forecast"),
OPTIONAL:
    <cnt limit>
        -cnt:  Limit of records to be returned}
martin Solved $
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

```
...usage()
```

Osnova

We have also set up two collections to validate the inputs:

- `VALID_IDENTIFIERS` - dictionary of command line flags as keys. Permitted inputs are stored in the values part of the dictionary -> option `-o` has only two options permitted `"weather","forecast"`
- `MANDATORY` - tuple of tuples - each tuple represents a group of identifiers from which at least one has to be included in the launching command

Once the command line arguments are parsed, the script uses the inputs for URL parameters.

Finally the `report()` function creates a formatted string that style the data into a table.

Function parse_command_line()

Function iterates over the keys in the `VALID_IDENTIFIERS` variable. It gets the index of a given identifier among arguments in `sys.argv` . If no such option is found, then it is handled through `ValueError` . Otherwise, we extract the argument following the found flag (identifier).

Then, using the variable `values` we check, whether there are some permitted values for a given flag. If so and the user did not provide a correct option, the usage information is printed and the script is terminated using `sys.exit()`

However, if all went well, we store the flag (excluding `'-'`) and the corresponding value in the `payload` dictionary that is returned at the end of the function.

Function report()

This function extracts the required information from the json response. Then it prepares the formatted table string doing the following:

1. determines column widths
2. creates a string that will be used as template for table row
3. transposes the original args tuple of tuples into a new table

Osnova

```
import json
import requests
import sys

KEY = "49dab41e2d87d2ffb4d7d8ebf833dd93"
URL = "https://api.openweathermap.org/data/2.5/{forecast_option}"
VALID_IDENTIFIERS = {'-id':None,
                     '-q': None,
                     '-lon':None,
                     '-lat':None,
                     '-o':("weather","forecast"),
                     '-cnt':None}
MANDATORY = (('id','q','lon','lat'),('o'))


def parse_command_line(identifiers,mandatory):
    payload = {}

    for identifier,values in identifiers.items():
        try:
            i =sys.argv.index(identifier)+1
            arg = sys.argv[i]

            if values and arg not in values:
                print('Not permitted value for option {}.'
                      '\nPermitted: {}'.format(identifier,values))
                print_usage()
                sys.exit()

            param = identifier.strip('-')
            payload[param] = arg

        except ValueError:
            continue
        except IndexError:
            print('Missing value for option {}'.format(identifier))
```

100% z Lekce 14

```
39
    Osnova      p in mandatory:
41          if not any(map(lambda x: x in group, payload)):
42              print('At least one of the following options is
    mandatory\n'
43                    '{}\n'.format(group))
44              print_usage()
45              sys.exit()
46
47      return payload
48
49
50  def print_usage():
51      print('USAGE: python weather.py <location> <weather/forecast>
    [<cnt limit>]')
52      print('''
53  MANDATORY:
54      <location> - at least one of the following
55          -id:   Location ID,
56          -q:    City_name | City_name,Country_code,
57          -lon:  Longitude,
58          -lat:  Latitude,
59      <weather/forecast>
60          -o:    One of the options: ("weather","forecast"),
61  OPTIONAL:
62      <cnt limit>
63          -cnt:  Limit of records to be returned}''')
64
65
66  def report(data):
67
68      args =(('DESCRIPTION', data['weather'][0]['description']),
69             ('CLOUDS',      str(data['clouds']['all']) + '%'),
70             ('WIND',        str(data['wind']['speed']) + 'm/s'),
71             ('TEMP_MAX',    str(data['main']['temp_max']-273.15) + '
    °C'),
72             ('TEMP_MIN',    str(data['main']['temp_min']-273.15) + '
    °C'),
```

100% z Lekce 14

```
    col_widths = [len(max((k,v),key=len)) for k,v in args]
    row_template = ' | '.join(['{{:^{w}}}'.format(w=width) for width in col_widths])
    table = [[row[row_num] for row in args] for row_num in range(len(args[1]))]
    formatted_table = ['| '+ row_template.format(*row) + ' |' for row in table ]

    print('\n'.join(formatted_table))


if __name__=="__main__":

    payload = {'APPID' :KEY}
    payload.update(parse_command_line(VALID_IDENTIFIERS,MANDATORY))

    URL = URL.format(forecast_option=payload.pop('o'))
    r = requests.get(URL,params=payload)

    report(r.json())
```

Osnova

DALŠÍ LEKCE