**What this script does:**

1. Requests the GitHub Topics page and checks the status code (200 means success).
2. Prints a snippet of the HTML for confirmation.
3. Saves the full HTML to variable webpage.html.
4. Parses it using BeautifulSoup.
5. Extracts **topic titles** and **descriptions** using their respective HTML tags/classes.
6. Builds a pandas DataFrame showing the extracted data.

**1. Importing libraries**

* **requests** — used to send HTTP requests and receive responses from web servers.
* **BeautifulSoup** (from the bs4 package) — used to parse and navigate the HTML structure of a webpage.
* **pandas** — used to organize and analyze the extracted data in a tabular format.

**2. Fetching the webpage content**

The program defines the URL of the website (the GitHub Topics page) and uses the requests.get() function to send an HTTP GET request to that URL.  
The server responds with the page’s HTML content.  
The program stores this response in a variable ‘response’ for further use.

**3. Verifying the HTTP request**

Every response from a web server includes a **status code** that indicates whether the request succeeded.  
The program prints this code — it should be 200, meaning *OK*.  
If it were, for example, 404, that would mean *page not found*, and the script would need to handle it differently.  
This check ensures the web request didn’t fail silently.

**4. Previewing the content**

Before proceeding, the script prints the **first 100 characters** of the HTML content to verify that something meaningful (like a valid HTML structure) was received.  
This helps confirm that the request didn’t return an error page or empty data.

**5. Saving the webpage to a file**

The entire HTML content is then written to a local file called **webpage.html**.

 open(...) — команда для открытия (или создания) файла.

 "webpage.html" — имя файла, в который мы будем писать. Если такого файла нет, он будет создан.

 "w" — режим **write (записи)**.  
Это значит, что содержимое файла (если он уже существовал) будет полностью перезаписано новыми данными.

 encoding=response.encoding — задаёт **кодировку текста**   
Это важно, чтобы символы в HTML (например, буквы с ударениями или иврит) сохранились корректно.  
The script uses the same encoding that the server specified in its response, which ensures that characters like symbols or non-English letters appear correctly when opened later.  
This step allows the script (or a human) to inspect the saved page if needed.

**6. Reading and parsing the HTML**

After saving, the script reopens the webpage.html file and loads its content into **BeautifulSoup**.  
BeautifulSoup parses the HTML, creating a structured tree representation of the page — this makes it easy to search for specific tags, classes, and text.

1. **open("webpage.html", "r", ...)**
   * open(...) — команда для открытия файла.
   * "webpage.html" — имя файла, который мы ранее сохранили.
   * "r" — режим **read (чтения)**, то есть мы не изменяем файл, а просто читаем его содержимое.
   * encoding=response.encoding — указывает, в какой кодировке нужно прочитать файл,  
     чтобы все символы (в том числе не латинские) отобразились корректно.  
     Это та же кодировка, что использовалась при записи.
2. **as file:**  
   — создаёт переменную file, через которую можно получить доступ к содержимому файла.
3. **with**  
   — автоматически закроет файл после завершения блока кода,  
   даже если возникнет ошибка. Это делает работу с файлами безопасной и аккуратной.

**🔹 soup = BeautifulSoup(file, "html.parser")**

Внутри блока происходит самое важное — **создаётся объект BeautifulSoup**, который будет анализировать HTML.

1. **BeautifulSoup(...)** — это конструктор, создающий объект для парсинга HTML или XML.  
   Он превращает «сырую строку HTML» в **convenient tree view**, где можно:
   * находить теги (<p>, <div>, <a>, и т.д.),
   * фильтровать по классам и атрибутам,
   * extract text, links, and other data.
2. **Первый аргумент (file)** — это источник данных.  
   Здесь file — это открытый файл (его содержимое BeautifulSoup может читать как поток).  
   Можно было бы также передать file.read(), но BeautifulSoup сам умеет читать файлоподобные объекты.
3. **Второй аргумент "html.parser"** — это **тип парсера**,  
   that is, a mechanism that disassembles HTML code.  
   не требует установки дополнительных **библиотек**.

**🔹 What happens in the end**

• The webpage file.html opens in reading mode.

BeautifulSoup analyzes the HTML code and builds the internal structure of the document.

• The soup variable is now an object

through which you can easily search for elements on a page:

1. for example, all <p> with a certain class, all links <a>, headings <h1>, etc.

**7. Identifying target data**

After the HTML page was loaded and disassembled using BeautifulSoup, all its code became available as a tree of elements (DOM).

Now you can search for specific parts of the page — for example, all paragraphs with a certain class.

GitHub uses CSS classes to design the elements on the page. These classes are also convenient to use for data extraction, because they indicate exactly where the necessary information is located.

The script identifies two types of information to extract from the page:

* **Topic titles** (the names of the topics, such as “Machine Learning” or “Python”).
* **Topic descriptions** (short explanations that appear below each title).

## 🔹 First line: search for titles (topics)

### 👉 titles = soup.find\_all("p", class\_="f3 lh-condensed mb-0 mt-1 Link--primary")

Разберём по частям:

1. **soup.find\_all(...)**  
   Это метод BeautifulSoup, который **находит все элементы** в HTML, соответствующие заданным условиям.  
   Он возвращает список найденных тегов (не текст, а именно объекты BeautifulSoup, представляющие HTML-теги).
2. **Первый аргумент "p"**  
   Это говорит: «Ищи все теги <p> (параграфы)».
3. **class\_="f3 lh-condensed mb-0 mt-1 Link--primary"**  
   Здесь указано дополнительное условие — выбираем только те <p>-теги, у которых атрибут class равен "f3 lh-condensed mb-0 mt-1 Link--primary".

These classes are special CSS tags that GitHub uses to style the theme text (names like "Machine learning", "Python", "Blockchain", etc.).На странице GitHub Topics именно такие <p>-теги содержат названия тем.

1. **Результат**  
   В переменной titles будет **список всех HTML-элементов <p>**, которые содержат названия тем.  
   Each item in the list can be processed later, for example, to get only the text using **.text.**

## 🔹 Second line: search for topic descriptions

### 👉 descriptions = soup.find\_all("p", class\_="f5 color-fg-muted mb-0 mt-1")

Теперь ищем **другие <p>-теги** — те, которые содержат описания тем.

1. Опять используется метод find\_all("p", ...).
2. На этот раз фильтр по классу другой: "f5 color-fg-muted mb-0 mt-1".  
   This is the design class that GitHub applies to the description text under each topic.  
   Например:
   * «Libraries and tools for machine learning»
   * «Discussions and implementations of algorithms»  
     и т. д.

3. The result is a list of HTML elements <p> containing descriptions.

## 🔹 Что происходит дальше (в следующих строках)

После этих двух строк у нас есть:

* titles → список элементов, где каждый <p> — это название темы.
* descriptions → список элементов, где каждый <p> — это текст-описание.

Next, clean text is extracted from these objects using the .text.strip() method, and two lists of strings are obtained, ready for analysis or saving.

## 🔹 Почему используется find\_all, а не find

Метод find\_all возвращает **все совпадения**, а не только первое.  
Если бы использовался find, программа получила бы только один заголовок и одно описание, а не весь список тем.

**8. Extracting and cleaning the data**

Using BeautifulSoup’s search functions, the script collects all matching elements for titles and for descriptions.

Чтобы достать именно **видимый текст**, используется метод .text:

* element.text → возвращает весь текст внутри тега, игнорируя HTML-теги внутри.  
  Например, если тег выглядит так:
* <p class="f3 lh-condensed mb-0 mt-1 Link--primary">Machine Learning</p>

то element.text вернёт строку "Machine Learning".

* .strip() → removes unnecessary spaces and line breaks at the beginning and end of a line.
* Это важно, потому что в HTML часто остаются символы табуляции, пробелы или переносы строки, которые мешают работать с данными.

As a result, something like a loop is executed for all the items in the lists.:

1. We take one <p> object from the list.

2. Extract the text from it using **.text** function.

3. Trim spaces and line breaks using .strip().

4. Save the result to a new list of lines.

После этого получаем два **чистых списка текста**:

* titles\_list → названия всех тем.
* descriptions\_list → описания всех тем.

Эти списки уже можно использовать для анализа, отображения или преобразования в DataFrame.

It then extracts only the **text content** from those elements, removing surrounding whitespace.  
The result is two Python lists — one containing all the topic names, and the other containing all the corresponding descriptions.

**9. Verifying extracted data**

To confirm that the extraction worked correctly, the script prints:

* The number of items in each list (to ensure it matches expectations, like 30 topics).
* The actual contents of both lists (to visually inspect that the right information was captured).

**10. Structuring the data**

Next, the script creates a **dictionary** where each key represents a category of information:  
one for the titles and another for the descriptions.  
The values under these keys are the lists extracted earlier.  
This dictionary format is ideal for converting the data into a structured table later.

**11. Creating a DataFrame**

The program then converts this dictionary into a **pandas DataFrame**, which functions like a spreadsheet in memory.  
Each row represents one GitHub topic, with columns for “Title” and “Description.”  
This makes the data much easier to view, analyze, and export.

**12. Displaying the result**

Finally, the script prints the DataFrame to the console so the user can see a neatly formatted table of the extracted topics and their descriptions.  
This serves as confirmation that the data pipeline — from fetching the web page to parsing and organizing the information — worked as intended.

**Summary of the Workflow**

1. **Connect to the website** → Fetch HTML.
2. **Check the connection** → Ensure response status is 200.
3. **Save locally** → Keep a copy of the page.
4. **Parse the page** → Use BeautifulSoup to interpret HTML.
5. **Locate key data** → Identify HTML tags/classes for topics and descriptions.
6. **Extract and clean** → Pull out the visible text.
7. **Organize results** → Store in a dictionary and convert to a DataFrame.
8. **Display output** → View structured information in a readable table.