УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Зорана Гајић

САВРЕМЕНЕ БИБЛИОТЕКЕ ЗА ПРИКУПЉАЊЕ ПОДАТАКА СА ВЕБ-СТРАНИЦА

мастер рад

Београд, 2023.

Ментор:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Весна Маринковић, доцент
Универзитет у Београду, Математички факултет

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

Велика захвалност менторки на саветима и мотивацији и породици на подршци.

Наслов мастер рада: Савремене библиотеке за прикупљање података са веб-страница

Резиме: Прикупљање података са веб-страница има кључну улогу у многим областима истраживања и пословања. За ефикасно прикупљање и парсирање података, неопходно је разумети савремене библиотеке и технике. Овај рад има за циљ да сагледа савремене библиотеке и технике за ефикасно прикупљање и парсирање података са веб-странице и да истакне њихове предности и недостатке, фокусирајући се на њихову имплементацију и функционалности.

Кључне речи: прикупљање података са веб-страница, веб-скрејпинг, парсирање *HTML* кода, библиотека *BeautifulSoap*, библиотека *Selenium*, библиотека *Scrapy*, алат *SPLASH*

Садржај

Глава 1

Увод

У данашњем дигиталном добу, велика количина података се налази на интернету, а веб-странице су богат извор информација за многе области истраживања и пословања. Међутим, прикупљање података са веб-страница може бити изазовно због различитих фактора као што су динамичност веб-страница, идентификација елемената у оквиру *HTML* кода и заштита података.

Аутоматизација процеса прикупљања података са веб-страница игра кључну улогу у откривању и праћењу трендова, анализи конкуренције, истраживању тржишта, предвиђању потрошачких преференција и многим другим областима. Уместо ручног прегледања и бележења података са великог броја веб-страница, аутоматизација омогућава брзо и ефикасно прикупљање података у великим количинама. Ово не само да штеди време и ресурсе, већ такође смањује могућност грешака и обезбеђује доследност у процесу прикупљања података.

Приступи и алати за прикупљање података са веб-страница су разноврсни и прилагођени специфичним потребама корисника. Постоје библиотеке и софтверски алати који омогућавају аутоматско претраживање веб-страница, екстракцију података и њихово складиштење у жељеном формату. Ови алати често користе технике попут веб-скрејпинга, анализе *HTML* структуре и употребе регуларних израза како би идентификовали и издвојили релевантне податке. Примери библиотека које омогућавају аутоматизацију процеса прикупљања података су: *BeautifulSoup* [13], *Selenium* [8] и *Scrapy* [6].

BeautifulSoup је Пајтон библиотека која се користи за парсирање *HTML* и *XML* документа. Омогућава једноставно извлачење података из *HTML*

страница. Библиотека *BeautifulSoup* пружа моћне функционалности за претраживање и манипулацију *HTML* структурама, олакшавајући проналажење, извлачење и обраду жељених података.

Selenium је популарна библиотека за аутоматизацију Веб-прегледача. Омогућава програмско управљање Веб-прегледачем за симулирање корисничких интеракција са веб-страницом. Ово се доминантно користи у контексту аутоматизације тестирања веб-апликација. Библиотека *Selenium* омогућава програмерима и тест инжењерима да аутоматски интерагују са Веб-прегледачима, симули- рају корисничке акције и проверавају очекиване резултате. Аутоматизација тестирања помоћу библиотеке *Selenium* омогућава ефикасно откривање грешака, смањује време и ресурсе потребне за ручно тестирање, као и обезбеђује доследност у извршавању тестова. Поред аутоматизације у тестирању, библиотека *Selenium* се користи и за прикупљање података јер омогућава прикупљање података који се динамички генеришу или су доступни само након одређених корисничких акција, као што су кликови на дугмад или попуњавање формулара.

Scrapy је моћна библиотека за прикупљање података са веб-страница. Помаже у претраживању, извлачењу и складиштењу података на структуриран начин. *Scrapy* омогућава брзо и ефикасно прикупљање велике количине података са веб-страница.

У глави 2 су детаљно разматрани изазови са којима је могуће се сусрести приликом прикупљања података са веб-страница, као и процес идентификовања елемената у *HTML* коду. У глави 3 је дат детаљан преглед наведених библиотека и алата који се користе за прикупљање података. Глава 4 је фокусирана на поређење наведених библиотека, док је у глави 5 изложен закључак на основу представљених информација.

Глава 2

Прикупљање података са веб-страница

Веб[1] (енг. *World Wide Web, WWW*) представља највећи извор података у историји човечанства, али се већина ових података састоји од неструктурираних информација, што може отежати њихово прикупљање [7]. На многим веб-сајтовима забрањено је копирање и преузимање података, али на сајтовима на којима је преузимање података дозвољено, ручно копирање може потрајати данима или недељама.

Веб скрејпинг (енг. *Web scraping*) представља аутоматизовани процес који омогућава издвајање података са различитих веб-страница и њихово чување у структурираном формату ради тренутне употребе или касније анализе. Постоје различити програмски језици који пружају подршку за имплементацију Веб скрејпинга, од којих су најпопуларнији: Пајтон (енг. *Python*), Јава (енг. *Java*) и Руби (енг. *Ruby*).

Поступак прикупљања информација састоји се од неколико фаза, које су приказане на слици 2.1. Прва фаза је проналажење одговарајуће веб-странице за прикупљање података (детаљније објашњено у одељку 2.1) и одређивање информација које су потребне за прикупљање. Након тога, потребно је послати *HTTP*[2] (енг. *Hypertext Transfer Protocol*) захтев на жељену веб-страницу и преузети изворни кôд *HTML* странице. Пре него што се парсира *HTML* кôд, потребно је пронаћи најбољи начин за индексирање жељених елемената, а за-

[1]Светска мрежа, познатија као Веб, систем је међусобно повезаних, хипертекстуалних докумената који се налазе на интернету.

[2]*HTTP* је мрежни протокол који припада слоју апликације референтног модела ОСИ, представља главни и најчешћи метод преноса информација на Вебу.

тим парсирати изворни кôд *HTML* странице и извршити неопходну радњу са добијеним информацијама [12].

Слика 2.1: Фазе прикупљања и употребе података

2.1 Изазови у процесу прикупљања података

Веб скрејпинг се сматра корисним процесом за добијање увида у податке. Међутим потребно је пазити на правне аспекте, како би се избегли легални проблеми. Важно је напоменути поштовање фајла *robots.txt* који представља политику веб-сајта. Овај фајл може садржати одредбе које забрањују приступ и прикупљање података са одређених делова веб-страница. Правилно разумевање закона о ауторским правима, заштити података и других релевантних прописа је од суштинске важности како би се осигурало законито и етичко прикупљање података. Најчешће се фајл *robots.txt* проналази на нивоу основног директоријума. Уколико фајл садржи линије попут ових приказаних у наставку, то значи да веб-сајт не жели да се прикупљају подаци са њега:

```
User-agent: *
Disallow:/
```

Да би прикупљање података било успешно, од суштинског значаја је квалитет добијених података. Како би се добили квалитетни подаци, потребно је да је сам веб-сајт исправан, односно да не садржи неисправне линкове, јер се веб скрејпинг обично изводи преко целог веб-сајта, а не само преко одређених страница.

Када се ради о пројектима великих размера и обимних база података, један од честих изазова јесте складиштење података. Овај изазов је повезан са ефикасним прикупљањем, обрадом и анализом велике количине података који се могу прикупити путем веб скрејпинга са различитих извора. Овај проблем може бити решен употребом већ постојећих платформи за складиштење.

У наставку ће бити описане најчешће заштите од напада на веб-странице који представљају изазове за процес веб скрејпинга:

1. *CAPTCHA* (енгл. *Completely Automated Public Turing test to tell Computers and Humans Apart*).

   CAPTCHA је технологија која се користи за проверу и потврду да је корисник веб-странице заиста човек, а не програм [14]. Провера се постиже приказивањем изазова, на пример слике са текстом или бројевима које је потребно препознати. Изазов је обично лак људима за решавање, али је тежак за програме који то треба брзо и аутоматски да реше. Корисници обично морају да унесу решење изазова како би потврдили да су људи и како би им био дозвољен приступ подацима на веб-страницама.

2. Захтеви за аутентификацију.

   Пријава корисника на веб-страницу може да представља велики изазов приликом веб-скрејпинга динамичних веб-страница. Уобичајени процес пријаве обухвата уношење корисничког имена и лозинке у одговарајућа поља на веб-страници, а затим клик на дугме за пријављивање. Приликом аутоматизације овог процеса могу се јавити потешкоће, али се оне могу решити уз помоћ библиотека као што су *Selenium* [8] и *Scrapy* [6].

3. Блокирање *IP* (енгл. *Internet Protocol address*) адреса.

   Веб-странице могу блокирати *IP* адресе које се повезују са прекомерним бројем захтева или са ботовима који су идентификовани као нежељени. Ово може бити привремено или трајно.

4. Провера корисничког агента.

Сваки *HTTP* захтев у заглављу шаље корисничког агента (енг. *user agent*). Коришћењем овог подешавања веб-сајт идентификује претраживач који му приступа: његову верзију и платформу. Уколико се користи исти кориснички агент у сваком захтеву, веб-сајт може лако да открије да је у питању аутоматизовани приступ страници.

5. Праћење учесталости прикупљања података.

   Како би се избегло преузимање садржаја са веб-странице у превеликој количини или превеликој брзини, веб-сајтови могу имплементирати ограничења фреквенције за ботове. Ова ограничења имају за циљ да контролишу број захтева по јединици времена и максималну брзину преузимања.

   Важно је разумети да ова ограничења нису постављена да би се спречило легитимно прикупљање података, већ да би се заштитио веб-сајт од претераног оптерећења. Уколико веб-сајт има велики обим података или има ограничене ресурсе, ограничења фреквенције су неопходна како би се осигурала стабилност и доступност сајта за све кориснике.

2.2 Идентификација елемената у оквиру *HTML* кода

Веб скрејпинг технологије подразумевају различите методе и библиотеке за издвајање података са веб-страница. У оквиру ових технологија користе се: регуларни изрази (енг. *Regular Expressions, RegEx*), тагови (енг. *tags*), *CSS* (енг. *Cascading Style Sheets*) селектори и *XPath* [3] (енг. *XML Path Language*).

Препоручени редослед идентификације елемената у оквиру *HTML* кода током веб скрејпинга је следећи:

1. Преко идентификатора — ако елементи имају јединствени идентификатор, најбрже и најпоузданије је користити овај начин идентификације.

2. По имену класе — ако се елементи налазе у истој класи, могу се идентификовати преко имена класе. Ово је корисно када је потребно издвојити групу елемената са заједничким стилом или функционалношћу.

3. По таговима — ако је неопходно издвојити све елементе са одређеним тагом, овај начин идентификације је најбољи.

4. *CSS* селектори — ако постоје елементи који немају јединствен идентификатор, али имају јединствен *CSS* стил, могу се идентификовати преко *CSS* селектора.

5. Регуларни изрази — ако је неопходно издвојити елементе на основу текста који се налази у њима.

6. *XPath* — ово је најопштији начин идентификације елемената у *HTML* коду.

Идентификатори и имена класа

Када елементи на веб-страници имају јединствен идентификатор, најбржи и најпоузданији начин иденфитикације је коришћење тог идентификатора. Идентификатор је обично атрибут `id` у *HTML* коду и служи као јединствено име за одређени елемент. Користећи идентификатор, могуће је директно циљати жељени елемент без потребе за додатним филтрирањем или селекцијом.

Ако се елементи на веб-страници налазе у истој класи, могуће је идентификовати их на основу имена класе. Класа је атрибут `class` у *HTML* коду и користи се за груписање елемената са сличним стилом или функционалношћу. Коришћење имена класе омогућава издвајање групе елемената са заједничким карактеристикама. Ово је корисно када је неопходно издвојити више елемената одједном.

Када користимо идентификаторе или имена класа за издвајање елемената са веб-страница, важно је разумети да ови идентификатори или класе нису универзални и да се могу разликовати између различитих сајтова. За сваки сајт је потребно проучити структуру *HTML* кода и идентификовати одговарајуће идентификаторе или класе које је неопходно издвојити.

Одређивање тачних идентификатора или класа за сваки сајт може бити процес који захтева проучавање *HTML* структуре, инспекцију елемената и анализу *CSS*-а. Ово је специфично за сваки појединачни сајт и не постоји универзална аутоматска метода за проналажење свих потребних идентификатора или класа.

Како бисте одредили које идентификаторе или класе треба користити, треба да се користе следеће методе:

- Проучавање *HTML* кода и идентификација атрибута `id` и `class` који се користе за циљане елементе.
- Анализа *CSS*-а како бисте идентификовали стилове или селекторе који су повезани са жељеним елементима.
- Употреба инспектора веб-прегледача који омогућава преглед и интерактивно истраживање *HTML* структуре и стилова.

Важно је напоменути да је приступ издвајању података путем идентификатора или имена класа осетљив на промене у структури или дизајну веб-страница. Ако се структура *HTML* кода или стилови сајта промене, идентификатори или класе које смо користили можда више неће бити тачни. Стога је важно редовно проверавати исправност и ажурирати кôд за издвајање података уколико се страница мења.

Тагови

Тагови играју кључну улогу у прикупљању података са веб-страница јер помажу у идентификацији и издвајању одређених информација из изворног кода *HTML* страница. Тагови у *HTML* коду се користе за дефинисање структуре веб-странице. Сваки таг представља одређени елемент или секцију странице, као што су заглавља, пасуси, слике и линкови.

У наставку су наведени тагови који се најчешће користе:

- `<html>` — Означава почетак и крај HTML документа.
- `<body>` — Представља садржај документа који је видљив кориснику.
- `<h1>` до `<h6>` — Користе се за дефинисање наслова.
- `<p>` — Користи се за дефинисање параграфа текста.
- `<a>` — Ствара хиперлинк (енгл. *hyperlink*) до друге веб-странице.
- `<ul>` и `<li>` — Користе се за стварање неуређене листе ставки.
- `<ol>` и `<li>` — Користе се за стварање уређене листе ставки.

- `<div>` — Користи се за дефинисање одељка документа у сврху стилизовања.
- `<span>` — Користи се за дефинисање малог дела текста у сврху стилизовања.

CSS селектори

CSS селектори се могу користити у процесу сакупљања података са веб-страница како би се идентификовали и издвојили одређени елементи. Овакав приступ је посебно користан када се ради са веб-страницама које не поседују јасну структуру и организацију.

CSS селектори раде на принципу идентификације елемената према њиховом имену ознаке, имену класе или идентификатору. На пример, селектор `div[class='imeKlase']` се користи за издвајање свих *div* елемената који имају класу *imeKlase*.

Регуларни изрази

Регуларни изрази представљају метод за усклађивање специфичних образаца у зависности од датих комбинација, који се могу користити као филтери за добијање жељеног резултата. У прикупљању података регуларни изрази се често користе за поређење шаблона и издвајање података, за локализовање и издвајање специфичних података из *HTML* или *XML* докумената. Једна од најзначајнијих предности регуларних израза јесте у њиховој универзалности, тј. могу се применити на било коју врсту података.

У многим програмским језицима, регуларни изрази се подржавају кроз уграђене библиотеке или модуле. Модул *re* програмског језика Пајтон пружа подршку регуларним изразима за поређење шаблона и издвајање података.

У наставку је дат пример регуларног израза који се може користити за претраживање и издвајање свих веб-адреса из изворног кода *HTML* странице. Конкретно, тражи се почетак хипервезе *a* која садржи атрибут *href*, а затим се издваја веб-адреса из овог атрибута и ставља у групу. У овом примеру регуларни израз користи знакове за постављање групе, односно за издвајање дела шаблона. Унутар овог израза, веб-адреса се ставља у групу помоћу парова заграда.

```
1 regex_pattern = r"<a\s+(?:[^>]*?\s+)?href=\"([^\"]*)\""
```

Језик *XPath*

Језик *XPath* представља флексибилан начин адресирања различитих делова *XML*[3] (енг. *Extensible Markup Language*) документа који су у формату *XML* или неком сличном формату. То га чини погодним за навигацију кроз објектни модел било ког таквог документа[4] (енг. *Document Object Model, DOM*), уз помоћ језика *XPath* (енг. *XPathExpression*). Израз у језику *XPath* дефинише образац за одабир скупа чворова и садржи преко 200 уграђених функција [3]. Овај језик је дефинисао *WWW* конзорцијум. У овом раду ће се језик *XPath* користити за одабир елемената са изворног кода *HTML* страница.

Синтакса језика *XPath*

Језик *XPath* користи изразе путања за избор чворова у *XML* документу. Чвор се одабира праћењем путање или корака.

Неки корисни примери израза путања су наведени у наставку:

`//h2` — Издваја све елементе *h2*.

`//div//p` — Издваја све елементе *p* који се налазе унутар блока *div*.

`//ul/li/a` — Издваја све линкове који се налазе унутар неуређених листи.

`//ol/li[2]` — Издваја други елемент уређене листе.

`//div/*` — Издваја све неурђене елементе који се налазе унутар блокова *div*.

`//*[@id=,,id'']` — Издваја елемент са идентификатором *„id"*.

`//*[@class=,,class'']` — Издваја све елементе са класом *„class"*.

`//a[@name or @href]` — Издваја све линкове који имају атрибут *name*, атрибут *href* или оба.

`//a[last()]` — Издваја последњи линк.

`//table[count(tr)=1]` — Издваја табеле које имају само један ред у њима.

[3] *XML* представља проширив (мета) језик за означавање (енгл. *markup*)

[4] Објектни модел документа представља хијерархијски приказ структуре веб-сајта.

`//*` — Издваја све елементе.

`//a/text()` — Издваја текст линка.

`./a` — Тачка издваја тренутни чвор.

Неки корисни примери функција у оквиру израза путања су наведени у наставку:

`string(n)` — Конвертује друге типове података у ниску. На пример, уколико је n број 42, онда ће резултат бити ниска „42”.

`number(n)` — Конвертује друге типове података у број. На пример, уколико је n ниска „42”, онда ће резултат бити број 42.

`contains(a, b)` — Проверава да ли се одређена ниска појављује унутар друге ниске. Први аргумент је ниска у којем се врши претрага, а други аргумент је ниска која се тражи. На пример, уколико је a ниска „abcdefg”, а b ниска „bcd”, онда ће резултат бити вредност *true*.

`starts-with(a, b)` — Проверава да ли одређена ниска почиње задатом подниском, односно да ли ниска a почиње са ниском b. На пример, уколико је a ниска „abcdefg”, а b ниска „abc”, онда ће резултат бити вредност *true*.

`ends-with(a, b)` — Проверава да ли одређена ниска завршава задатом подниском, односно да ли се ниска a завршава са ниском b. На пример, уколико је a ниска „abcdefg”, а b ниска „efg”, онда ће резултат бити вредност *true*.

Глава 3

Преглед библиотека за прикупљање података са веб-страница

У оквиру рада биће извршено детаљно прикупљање података са веб-странице *https://www.audible.com/search*. На главној страници *Audible* веб-сајта налази се бочна секција са списком категорија књига. Свака категорија представља одређену тематску групу књига, као што су „Уметност и забава", „Биографије и мемоари", „Посао и каријера" итд. Унутар сваке категорије, постоји списак књига које припадају тој теми. Да би се приступило свим књигама у једној категорији, потребно је прећи кроз све странице кроз пагинацију. Пагинација омогућава прелазак на следећу или претходну страницу, како би се приказале све доступне књиге у тој категорији. Проласком кроз све категорије и њиховим пагинацијама, могуће је прикупити информације о свим доступним књигама на веб-страници *Audible*.

3.1 Библиотека *BeautifulSoap*

Библиотека *BeautifulSoap* [13] је Пајтон библиотека која се користи за парсирање и претраживање *HTML* и *XML* докумената. Ова библиотека подржава различите врсте навигације кроз *HTML* и *XML* документе, као што су претраживање по имену тагова, претраживање по садржају тагова, претраживање по атрибутима тагова и слично. Једна од главних особина библиотеке *BeautifulSoap* је да је компатибилна са различитим парсерима, укључујући

html.parser [4], *lxml* [5] и *html5lib* [9]. За разлику од других библиотека које ће се касније разматрати, ова библиотека не може сама да приступи веб-страници и потребни су јој помоћни модули.

Библиотека *BeautifulSoap* има многе карактеристике које олакшавају њену употребу. Библиотека се лако инсталира помоћу наредбе *pip* и има једноставан интерфејс (енг. *interface*).

Инсталација

Библиотека *BeautifulSoap* се може инсталирати користећи алат за инсталирање библиотека за програмски језик Пајтон звани *pip* [1]. Неопходно је покренути следећу наредбу из командне линије:

```
pip3 install bs4
```

Ова наредба ће преузети и инсталирати најновију верзију библиотеке *BeautifulSoap*. Након успешне инсталације, неопходно је увести библиотеку у Пајтон кôд користећи следећу наредбу:

```
from bs4 import BeautifulSoup
```

Провера динамичности веб-странице

Многе веб-странице, укључујући веб-страницу *audible.com*, која се анализира у овом раду, користе динамичке технологије које омогућавају промену садржаја без освежавања целе странице, што представља изазов при парсирању таквих страница. У овом контексту, библиотека *BeautifulSoap* се најчешће користи за анализу *HTML* или *XML* кода веб-страница, али због динамичности неких страница, могуће је да се не ухвате све промене на страници. Због тога се користе библиотеке попут *Selenium* [8] и *Scrapy* [6] за праћење промена у реалном времену.

Прикупљање *HTML* кода веб-странице

Библиотека *BeautifulSoup* не представља самосталну библиотеку за прикупљање података са веб-страница. Да би се преузео *HTML* кôд веб-странице

неопходно је инсталирати библиотеку *Requests-HTML*, која омогућава креирање *HTTP* захтева на одређену веб-страницу и за одговор добија *HTML* кôд те странице.

Постоји неколико метода од значаја у библиотеци *Requests-HTML* [11]:

- `get(url, params=None, **kwargs)`

  Ова метода шаље *HTTP GET*[1] захтев на наведену веб-адресу.

- `post(url, data=None, json=None, **kwargs)`

  Ова метода шаље *HTTP POST*[2] захтев на наведену веб-адресу.

- `put(url, data=None, **kwargs)`

  Ова метода шаље *HTTP PUT*[3] захтев на наведену веб-адресу.

Додатни параметри *kwargs*[4] омогућавају спецификацију додатних опција.

Неколико уобичајених опционалних параметара из више наведених метода у оквиру библиотеке *Requests-HTML*:

- `params`

  Опциони параметар *params* се користи за слање додатних параметара у *URL* у облику упитних параметара приликом слања *HTTP GET* захтева. Упитни параметри се додају на крај *URL*-а након знака „?” и обично се састоје од имена параметра и његове вредности раздвојених знаком „=”. На пример, параметар може да се искористи за филтрирање резултата, сортирање или специфицирање странице у случају пагинације.

- `data`

  Опциони параметар *data* се користи за слање података у телу захтева приликом слања *HTTP POST* или *HTTP PUT* захтева. Подаци могу бити у облику ниске или у облику речника који ће бити аутоматски кодиран у одговарајући формат.

[1] *HTTP GET* захтев је метод комуникације у *HTTP* протоколу који се користи за захтевање ресурса са сервера.

[2] *HTTP POST* захтев је метод комуникације у *HTTP* протоколу који се користи за слање података серверу ради креирања или ажурирања ресурса.

[3] *HTTP PUT* захтев је метод комуникације у *HTTP* протоколу који се користи за ажурирање постојећег ресурса на серверу.

[4] *kwargs* је скраћеница за ”*keyword arguments*” и представља специјални елемент у програмском језику Пајтон који омогућава преношење произвољног број названих аргумената у функцији.

- `json`

  Опциони параметер *json* се такође користи за слање података у телу захтева приликом слања *HTTP POST* или *HTTP PUT* захтева, али уместо обичне ниске, подаци се шаљу као *JSON* објекат. Библиотека *Requests-HTML* ће аутоматски серијализовати *JSON* објекат и поставити одговарајуће заглавље захтева.

- `headers`

  Опциони параметер *headers* се користи за специфицирање додатних заглавља *HTTP* захтева. Могуће је користити овај параметар за постављање специфичних заглавља као што су кориснички агент или слање ауторизационог токена.

- `timeout`

  Опциони параметер *timeout* се користи за постављање временског ограничења за чекање на одговор сервера.

Кôд приказан на листингу 3.1 представља кôд у програмском језику Пајтон који преузима *HTML* кôд веб-странице. Важно је знати да преузимањем веб-странице помоћу Пајтон библиотеке *Requests-HTML*, постоји могућност да се деси да страница није доступна на серверу (или да је дошло до грешке у њеном преузимању), или да сервер није доступан.

```
1 import requests
2
3 url = 'https://www.audible.com/search'
4 try:
5     response = requests.get(url)
6 except requests.exceptions.RequestException:
7     print("Error fetching page")
8     exit()
9
10 html = response.text
```

Listing 3.1: Прикупљање *HTML* кода веб-странице

Парсирање *HTML* кода веб-странице

Пајтон нуди разне библиотеке за парсирање *HTML* кода, од којих су две најзаступљеније: *lxml* и *html.parser*. Парсер *lxml* је најбржи парсер веб-

страница према званичној документацији библиотеке *BeautifulSoup* [13], који може да анализира велике и сложене документе. Парсер *html.parser* је уграђени Пајтон парсер који је намењен да ради са мањим и једноставнијим *HTML* документима [11].

Да би се извршило парсирање добијеног *HTML* кода веб-странице, прво је неопходно креирати објекат *BeautifulSoup* уз помоћ добијеног *HTML* кода и жељеног парсера. Осим наведеног корака, у Пајтон коду на листингу 3.2 је приказано да резултат креирања објекта *BeautifulSoup* нуди издвајање наслова и текста веб-странице, поред разних других информација.

```
1 from bs4 import BeautifulSoup
2 ...
3 # html objekat je dobijen kao povratna vrednost funckije get modula
      requests
4 ...
5 soup = BeautifulSoup(html, 'lxml')
6 print(soup.text)
7 print(soup.title.text)
```

Listing 3.2: Креирање објекта *BeautifulSoup*

Добијени објекат *BeautifulSoup* такође омогућава приступ различитим деловима *HTML* кода користећи методе као што су `find()` и `find_all()`. Метода `find()` користи се када је потребно пронаћи први елемент у *HTML* коду који одговара одређеном тагу или класи. Ова метода враћа први пронађени елемент који одговара постављеним критеријумима, док се метода `find_all()` користи када је потребно пронаћи све елементе у *HTML* коду који одговарају одређеном тагу или класи. Ова метода враћа листу свих пронађених елемената који одговарају постављеним критеријумима.

Кôд приказан на листингу 3.3 прикупља податаке о књигама са веб-странице *Audible*. Прво је неопходно преузети *HTML* садржај веб-странице (кôд приказан на листингу 3.1), а затим креирати објекат *BeautifulSoup* за парсирање *HTML* садржаја (кôд приказан на листингу 3.2). Затим се проналази елемент *div* са класом *„adbl-impression-container”*, унутар којег се проналазе сви елементи *li* са класом *„productListItem”*. За сваку књигу у листи, извлачи се наслов, аутор, датум издања и цена, који се затим додају у одговарајуће листе.

```
1 ...
```

```
2 # soup objekat je dobijen kao povratna vrednost prilikom kreiranja
      objekta tipa BeautifulSoup
3 ...
4 container = soup.find('div', class_='adbl-impression-container')
5 book_list = container.find_all('li', class_='productListItem')
6
7 for book in book_list:
8     book_titles.append(book.find('h3', class_='bc-heading').text.strip
      ())
9     book_authors.append(book.find('li', class_='authorLabel').a.text.
      strip())
10     book_release_dates.append(substr_after_colon(book.find('li',
      class_='releaseDateLabel').text.strip()))
11     book_prices.append(extract_regular_price(book.find('div', class_='
      adblBuyBoxPrice').text.strip()))
```

Listing 3.3: Издвајање наслова, аутора, датума издања и цене књиге из *HTML* кода веб-странице

Прикупљање података са више веб-страница

Када се користи библиотека *BeautifulSoup* за прикупљање података са више веб-страница, могу се јавити проблеми у вези са аутоматским прикупљањем података са свих жељених страница. Када се прикупљају подаци са једне странице, обично се користи функција `get` модула *requests* (листинг 3.1) за дохват *HTML* кода и затим се креира објекат `BeautifulSoup` (листинг 3.2) за анализу *HTML* кода и издвајање неопходних података. Међутим, ако се подаци прикупљају са више страница, неопходно је итерирати кроз све странице и аутоматски дохватити *HTML* кôд за сваку страницу. На пример, ако странице имају адресе које се разликују само по броју странице, може да се искористи петља која пролази кроз све адресе и дохвата *HTML* кôд сваке странице.

Прилагођавање овог процеса је за сваки веб-сајт специфично. Сваки веб-сајт може имати различиту структуру *HTML* кода и различите *URL* адресе. Аутоматизација овог процеса може бити изводљива у неким случајевима, на пример, ако постоји доследан шаблон *URL* адреса или правилност у структури *HTML* кода. Међутим, често је потребно прилагодити кôд за сваки веб-сајт како би се успешно прикупили подаци. Осим коришћења петље и генерисање *URL* адреса, постоје и друге честе опције које се могу користити за

прикупљање података са више веб-страница. Једна опција је употреба *API*[5] (енг. *Application Programming Interface*) уколико је доступан. *API* омогућава програмски приступ подацима са веб-странице, што може бити ефикасан начин за прикупљање жељених података. Такође треба размотрити коришћење других библиотека као што су *Selenium* и *Scrapy* које пружају напредне могућности за прикупљање података.

На пример, да би се прикупиле све информације о књигама са веб-странице *Audible*, потребно је проћи кроз све категорије и пагинацију на свакој од тих категорија. Коришћењем методе `find` библиотеке *BeautifulSoup*, проналази се елемент *div* који садржи листу елемената *li*, који представљају веб-адресе за сваку од категорија. Затим је потребно итерирати кроз листу веб-адреса, учитати *HTML* кôд за сваку веб-адресу и пронаћи пагинациони елемент из којег се извлачи број последње странице. Након тога, пролази се кроз све странице одабране категорије. Са сваке странице је могуће извући информације о насловима, ауторима, датумима издања и ценама књига.

Важно је напоменути да библиотека *BeautifulSoup* не симулира интеракцију са Веб-прегледачем, што означава да итерирање кроз категорије и кроз странице се врши преласком са једне веб-адресе на другу веб-адресу уочавањем обрасца у веб-адреси. Другим речима, за пагинацију је уочено да се вредност параметра *page* мења између *page=1*, *page=2*, *page=3* и слично.

```
1 ...
2 # soup objekat je dobijen kao povratna vrednost prilikom kreiranja
      objekta tipa BeautifulSoup
3 ...
4 pagination = soup.find('ul', class_='pagingElements')
5 pages = pagination.find_all('li', class_='bc-list-item')
6 last_page = pages[-2].text
7
8 for page in range(1, int(last_page) + 1):
9     ...
10    # html objekat je dobijen kao povratna vrednost funckije get
      modula requests
11    ...
12    html = audible_shared.fetch_html(f'{website}?page={page}')
13    soup = BeautifulSoup(html, 'lxml')
14    container = soup.find('div', class_='adbl-impression-container')
```

[5] *API* представља скуп правила, протокола и инструкција које омогућавају комуникацију између различитих софтверских компоненти.

```
15      titles, authors, release_dates, prices = audible_shared.
        collect_books_info(container)
16      book_titles.extend(titles)
17      book_authors.extend(authors)
18      book_release_dates.extend(release_dates)
19      book_prices.extend(prices)
```

Listing 3.4: Прикупљање података са више веб-страница

3.2 Библиотека *Selenium*

Библиотека *Selenium* је популарна библиотека програмског језика Пајтон која се користи за ефикасну аутоматизацију интеракције са веб-страницама. Она омогућава симулирање корисничке интеракције са веб-страницама, као што су уношење текста, кликтање, претраживање елемената и прикупљање података.

Библиотека *Selenium* пружа богат скуп функција за претрагу елемената на веб-страници, као што су проналажење елемената по идентификатору, имену, класи, ознаци или изразу *XPath*. Ово омогућава једноставну манипулацију одређеним деловима веб-страница. Још једна корисна особина ове библиотеке је могућност руковања чекањима и интеракцијом са динамичким елементима странице. На пример, могуће је да се сачека да се одређени елемент учита пре него што се изврше следеће наредбе.

Инсталација

Библиотека *Selenium*, слично библиотеци *BeautifulSoup*, се може инсталирати користећи алат *pip*. Неопходно је покренути следећу наредбу из командне линије:

```
pip3 install selenium
```

Након успешне инсталације, неопходно је увести библиотеку у Пајтон кôд користећи следећу наредбу:

```
import selenium
```

Улога драјвера

Управљачки програм или драјвер (енг. *driver*) је програм који омогућава комуникацију између програма вишег нивоа, као што је апликација, и рачунарске опреме. Када се користи библиотека *Selenium*, није могуће директно комуницирати са Веб-прегледачем (енг. *web browser*), већ је неопходно користити драјвер који ће посредовати у комуникацији између кода и Веб-прегледача и омогућити контролу над Веб-прегледачем користећи библиотеку *Selenium*. За сваки Веб-прегледач постоји одређени драјвер који се користи са библиотеком *Selenium*. На пример, за Веб-прегледач Гугл кроум (енг. *Google Chrome*) се користи драјвер *ChromeDriver*, док се за Веб-прегледач Мозила фајерфокс (енг. *Mozilla Firefox*) користи драјвер *GeckoDriver*.

Како би се омогућило коришћење драјвера у Пајтон коду, потребно је да се преузме одговарајућа верзија драјвера за неопходни Веб-прегледач. Након тога треба навести путању до драјвера и инстанцирати драјвер коришћењем модула *webdriver* из библиотеке *Selenium*. Наведени кôд на листингу 3.5 представља претходно описане кораке за случај када је коришћен Веб-прегледач Гугл кроум. Након тога, кôд може да отвори веб-адресу и управља истом. На крају, линија `driver.quit()` затвара Веб-прегледач и ослобађа коришћене ресурсе.

```
1 from selenium.webdriver.chrome.service import Service
2 from selenium import webdriver
3
4 path = '/usr/local/bin/chromedriver_mac64_arm64/chromedriver'
5 service = Service(executable_path=path)
6 driver = webdriver.Chrome(service=service)
7 website = 'https://www.audible.com/search'
8 driver.get(website)
9 ...
10 # ostatak koda za manipulaciju veb-stranicom
11 ...
12 driver.quit()
```

Listing 3.5: Прикупљање *HTML* кода веб-странице помоћу библиотеке *Selenium*

Headless режим

Headless режим се односи на извршавање програма у позадини, без потребе за приказивањем корисничког графичког интерфејса и интеракције са корисником путем миша и тастатуре. Ова врста извршавања програма је корисна у различитим контекстима и служи за разне сврхе, а једна од њих је веб-скрејпинг.

Веб-скрејпинг је процес прикупљања података са веб-страница. Програм који ради у *headless* режиму може аутоматски посетити веб-странице, извршавати одређене акције и прикупљати податке без потребе за приказивањем страница кориснику. На пример, може се извршити претраживање и прикупљање информација са различитих веб-страница, без приказа слика, дугмади или падајућих менија на екрану. Иако се ови елементи не приказују, и даље је могуће навигирати између веб-страница, кликнути на било који елемент и извршавати сличне акције.

За коришћење *headless* режима у Пајтон коду са библиотеком *Selenium*, потребно је конфигурисати драјвер за одговарајући Веб-прегледач и поставити опцију за *headless* извршавање, што је приказано у коду на листингу 3.6

```
1 from selenium.webdriver.chrome.options import Options
2 options = Options()
3 options.add_argument('--headless')
4 driver = webdriver.Chrome(service=service, options=options)
```

Listing 3.6: Омогућавање *headless* режима

Имплицитно и експлицитно чекање

Постоје два основна метода чекања у оквиру библиотеке *Selenium*: имплицитно чекање и експлицитно чекање. Обе методе се користе како би се осигурало да се одређена радња изврши тек након што се испуни одређени услов, као што је приказивање одређеног елемента на веб-страници или завршетак одређене акције.

Експлицитно чекање је доступно у оквиру библиотеке *Selenium* за императивне програмске језике и омогућава коду да заустави извршавање програма или замрзне нит све док се не испуни услов који му се преда. Услов се прове-

рава са одређеном учесталошћу све док се не истакне време чекања. То значи да ће, све док услов не врати вредност *falsy*, покушавати и чекати [8].

Модул *WebDriverWait* омогућава чекање одређеног временског периода док се одређени услови не испуне на веб-страници. За инстанцирање класе `WebDriverWait` неопходно је проследити два аргумента у конструктор: инстанцу објекта `WebDriver` (који представља веб-драјвер за аутоматско управљање Веб-прегледачем) и време чекања у секундама. Затим се може употребити метод `until` објекта `WebDriverWait` са прослеђеним аргументом који представља жељени услов који треба да се испуни. На пример, кôд који је приказан на листингу 3.7 чека да се дугме на локацији датог израза *XPath* учини кликабилним пре него што се настави са извршавањем кода.

```
1 ...
2 # driver predstavlja instancu Chrome veb drajvera koja se koristi za
      automatizaciju pregledaca
3 ...
4 next_page = driver.find_element(By.XPATH, value='//span[contains(
      @class, "nextButton")]')
5 next_page.click()
```

Listing 3.7: Симулација клика на елемент

Модул *expected_ conditions* садржи различите услове који проверавају одређене карактеристике елемената на веб-страници. На пример, за проверу да ли је одређен елемент кликабилан може да се искористи метод `expected_conditions.element_to` а за проверу да ли је елемент видљив на страници може да се искористи метод `expected_conditions.visibility_of_element_located`. Ови услови се користе у комбинацији са објектом модула *WebDriverWait* како би се сачекали одређени услови пре него што се настави са извршавањем кода. Коришћење оба модула показало се јако корисно у случају постојања интерактивних елемената на страницама које се динамички учитавају или ако је неопходно проверити одређене карактеристике пре него што се настави са прикупљањем података са веб-странице.

Имплицитно чекање се поставља само једном и примењује глобално на све радње које извршава драјвер. Када се користи имплицитно чекање, драјвер ће чекати одређено време пре него што баци изузетак *ElementNotVisibleException* или *NoSuchElementException* уколико не може пронаћи елемент. Имплицитно чекање подразумева да *WebDriver* периодично претражује *DOM* у одређеном временском периоду када покушава да пронађе било који елемент. Ово може

бити корисно када одређени елементи на веб-страници нису одмах доступни и захтевају неко време да се учитају [8].

У оквиру библиотеке *Selenium* такође постоји и такозвани *FluentWait*. Инстанца *FluentWait* дефинише максимално време чекања на услов, као и учесталост провере услова [8].

Лоцирање елемената

Библиотека *Selenium* дефинише два главна метода за ефикасно лоцирање елемената на веб-страницама:

`findElement` — За резултат враћа један елемент који одговара задатом критеријуму.

`findElements` — За резултат враћа листу елемената који задовољавају дати критеријум.

Оба ова метода прихватају аргумент у облику стратегије лоцирања елемента. Стратегија лоцирања одређује на који начин ће се елемент пронаћи на веб-страници. Неке од често коришћених стратегија су:

`ID` — Лоцирање елемента по јединственом идентификатору.

`NAME` — Лоцирање елемента по његовом имену атрибута.

`XPATH` — Лоцирање елемента помоћу израза *XPath* који пружа путању до елемента.

`TAG_NAME` — Лоцирање елемента по називу ознаке.

`CLASS_NAME` — Лоцирање елемента по називу *CSS* класе.

`CSS_SELECTOR` — Лоцирање елемента помоћу *CSS* селектора.

Помоћу ових стратегија за лоцирање елемената, могуће је тачно идентификовати жељене елементе на веб-страници и извршити различите операције над њима. Једна од операција може бити кликтање на елемент и то је приказано у коду на листингу 3.7. Такође, могуће је изабрати опцију из падајуће листе користећи методе `select_by_visible_text()` или `select_by_value()` уз употребу класе `Select`. Приказан кôд на листингу 3.8 проналази падајућу

листу за избор начина сортирања на веб-страници, чека да буде кликабилна и одабира опцију са вредношћу *popularity-rank* из те листе.

```
1 from selenium.webdriver.support.ui import Select, WebDriverWait
2 ...
3 # driver predstavlja instancu Chrome veb drajvera koja se koristi za
      automatizaciju pregledaca
4 ...
5 refinement_dropdwon_wait = WebDriverWait(driver, 20).until(EC.
      element_to_be_clickable((By.XPATH, "//select[@aria-labelledby='
      sortBy']")))
6 refinement_dropdown = Select(refinement_dropdwon_wait)
7 refinement_dropdown.select_by_value('popularity-rank')
```

Listing 3.8: Одабир опције из падајућег менија

.

Читање садржаја елемента са веб-странице се може извршити користећи методу `text`. Кôд приказан на листингу 3.9 користи методу `text` за издвајање текста из елемената који представљају наслов, аутора и датум издавања књиге.

```
1 book_titles = []
2 book_authors = []
3 book_release_dates = []
4 book_prices = []
5 ...
6 # driver predstavlja instancu Chrome veb drajvera koja se koristi za
      automatizaciju pregledaca
7 ...
8 container = driver.find_element(By.CLASS_NAME, value='adbl-impression-
      container ')
9 book_list = container.find_elements(By.XPATH, value='.//li[contains(
      @class, "productListItem")]')
10
11 for book in book_list:
12     book_titles.append(book.find_element(By.XPATH, value='.//h3[
      contains(@class, "bc-heading")]').text.strip())
13     book_authors.append(book.find_element(By.XPATH, value='.//li[
      contains(@class, "authorLabel")]').find_element(By.TAG_NAME, 'a').
      get_attribute('innerHTML'))
14     book_release_dates.append(helpers.substr_after_colon(book.
      find_element(By.XPATH, value='.//li[contains(@class, "
```

```
releaseDateLabel")]').text.strip()))
```

Listing 3.9: Читање садржаја елемента

.

Прикупљање података са више веб-страница

Начин имплементације за прикупљање свих жељених података зависи од структуре конкретног *HTML* кода веб-странице. Постоји могућност да веб-страница користи пагинацију или бесконачан скрол.

За веб-сајт који користи пагинацију, подаци се могу прикупити на следећи начин:

1. Учитати почетну страницу.
2. Идентификовати елементе који садрже жељену информацију и прикупити податке са веб-странице.
3. Проверити да ли постоји навигациони елемент за прелазак на следећу страницу.
4. Ако постоји, извршити клик на навигациони елемент за прелазак на следећу страницу.
5. Сачекати да се учита следећа страница.
6. Поновити кораке 2—5 све док се не прикупе подаци са свих страница у пагинацији.

У коду на листингу 3.10 је приказан процес прикупљања података са више веб-страница. Овај кôд проналази пагинацију на платформи *Audible*, прикупља информације о књигама са сваке странице и чува у одређеним листама. Користи се петља *while* за итерацију кроз све странице пагинације, а *WebDriverWait* се користи за чекање приказа одређеног елемента на свакој страници пре прикупљања инфромација. Након тога, користи се метод `click()` за прелазак на следећу страницу.

```
1 ...
2 # driver predstavlja instancu Chrome veb drajvera koja se koristi za
      automatizaciju pregledaca
3 ...
```

```
4
5 pagination = driver.find_element(By.XPATH, value='//ul[contains(@class
      , "pagingElements")]')
6 pages = pagination.find_elements(By.TAG_NAME, value='li')
7 last_page = int(pages[-2].text)
8 current_page = 1
9
10 while current_page <= last_page:
11     container = WebDriverWait(driver, 20).until(EC.
      presence_of_element_located((By.CLASS_NAME, 'adbl-impression-
      container ')))
12     titles, authors, release_dates, prices = audible_shared.
      collect_books_info(container)
13     book_titles.extend(titles)
14     book_authors.extend(authors)
15     book_release_dates.extend(release_dates)
16     book_prices.extend(prices)
17     current_page += 1
18
19     try:
20         next_page = driver.find_element(By.XPATH, value='//span[
      contains(@class, "nextButton")]')
21         next_page.click()
22     except:
23         pass
```

Listing 3.10: Пагинација у оквиру библиотеке *Selenium*

Кључни корак у овој имплементацији је итерирање кроз све странице пагинације, прикупљање података са сваке странице и прелазак на следећу страницу све док се не прикупе сви жељени подаци.

За веб-сајт који користи бесконачан скрол, подаци се могу прикупити на следећи начин:

1. Учитати почетну страницу.
2. Идентификовати елементе који садрже жељену информацију и прикупити податке са веб-странице.
3. Извршити скрол на дно веб-странице користећи функционалности библиотеке *Selenium*.
4. Сачекати да се учитају нови подаци.

5. Поновити кораке 2—5 све док се не прикупе сви подаци.

Кључни корак у овој имплементацији је непрекидно скроловање на дно странице и прикупљање података који се динамички учитавају.

Када је у питању веб-страница *Audible*, користи се библиотека *Selenium* за аутоматизацију прегледача како би се симулирао клик на дугме *Next Page* и прелазак са једне веб-странице на другу веб-страницу. Неопходно је проћи кроз све категорије на веб-страници, а затим кроз све странице унутар сваке категорије. Са сваке странице се прикупљају исти подаци као у случају коришћења библиотеке *BeautifulSoap*. Коришћењем драјвера из библиотеке *Selenium*, кôд симулира клик на дугме *Next Page* како би се прешло на следећу страницу све док се не дође до последње странице унутар категорије.

Изазови аутентикације и аутоматизације

При веб скрејпингу, изазови аутентикације и аутоматизације односе се на проблеме који се јављају приликом приступа и пријаве на веб-странице. Веб-странице захтевају аутентификацију корисника, обично путем корисничког имена и лозинке, пре него што дозволе приступ одређеним подацима. Уколико није успешно извршена пријава на веб-страницу, приступ циљаним подацима је обично онемогућен. Да би спречиле аутоматизовани приступ и веб-скрејпинг, веб-странице могу користити различите технике, као што је *CAPTCHA*. Ове мере могу онемогућити успешну пријаву приликом веб скрејпинга.

У коду на листингу 3.11 је приказано како да се превазиђе проблем пријављивања на веб-страницу користећи програмски језик Пајтон и библиотеку *Selenium*. Кôд аутоматски попуњава поља за унос корисничког имена и лозинке на веб-страници користећи функцију `send_keys`. Након што су унети подаци, неопходно је искористити функцију `send_keys(Keys.ENTER)` како би се симулирао притисак тастера *Enter* и послала форма за пријаву.

```
1 from selenium.webdriver.common.keys import Keys
2 ...
3 # driver predstavlja instancu Chrome veb drajvera koja se koristi za
      automatizaciju pregledaca
4 ...
5
6 # Find username and password inputs
```

```
7 username_field = driver.find_element(By.ID, value="username")
8 password_field = driver.find_element(By.ID, value="password")
9
10 # Enter user name and password
11 username_field.send_keys("your_username")
12 password_field.send_keys("your_password")
13
14 # Submitting the login form
15 password_field.send_keys(Keys.ENTER)
```

Listing 3.11: Пријављивање на веб-страници

.

3.3 Библиотека *Scrapy*

Scrapy је ефикасна библиотека за прикупљање података са веб-страница, која подржава брзо и паралелно прикупљање велике количине података, чинећи је идеалном библиотеком за прикупљање, индексирање или истраживање веб садржаја. Базира се на појму „паукова". Паук је програмски модул у библиотеци *Scrapy* који дефинише како се претражују и извлаче подаци са веб-страница.

Библиотека *Scrapy* такође подржава паралелно извршавање паука, што омогућава брже прикупљање података са више веб-страница истовремено. Такође обезбеђује механизме за управљање аутентификацијом.

Инсталација

Библиотека *Scrapy*, слично библиотеци *Selenium*, се може инсталирати користећи алат *pip*. Неопходно је покренути следећу наредбу из командне линије:

```
pip3 install scrapy
```

Након успешне инсталације, потребно је креирати *Scrapy* пројекат. За креирање пројекта, треба се позиционирати у жељени директоријум у оквиру терминала и извршити следећу команду која ће аутоматски генерисати почетне директоријуме и датотеке које су потребне.

```
scrapy startproject project_name
```

Паук

Паук (енг. *Spider*) је основна јединица у библиотеци *Scrapy* која претражује веб-странице, преузима податке и даље их обрађује. Паук користи веб-адресе и *HTTP* захтеве да преузме веб-странице, парсира *HTML* садржај, извуче податке и складишти у жељеном формату. Такође, омогућава навигацију кроз веб-странице и прилагођавање понашања према захтевима. Паук је кључна компонента за аутоматизовано извлачење података са веб-страница помоћу библиотеке *Scrapy*.

Scrapy шаблони

Библиотека *Scrapy* садржи два основна шаблона за креирање паука: *Spider* и *CrawlSpider*. Шаблон *Spider* омогућава ручно дефинисање логике претраге и прикупљања података, док је шаблон *CrawlSpider* оптимизован за претрагу веб-страница са више нивоа линкова.

Шаблон *Spider*

Шаблон *Spider* је основни шаблон паука у библиотеци *Scrapy*, који пружа флексибилност за ручно дефинисање логике претраге и прикупљања података са веб-страница. При коришћењу овог шаблона, прво је потребно креирати класу паука која наслеђује класу *Spider*. У овој класи се дефинише метод `start_requests` за генерисање почетних веб-адреса које паук треба да посети. Након тога, када паук добије одговор са веб-странице, користи се метод `parse` за обраду одговора и преузимање података са веб-странице. Могуће је користити различите селекторе као што су *XPath* и *CSS* за проналажење елемената на страници. Овај шаблон такође омогућава подешавања атрибута као што су `allowed_domains` и `start_urls`.

Атрибут `allowed_domains` се користи за ограничавање паука само на одређене домене. Потребно је дефинисати листу домена које паук треба да посети. Паук ће игнорисати све веб-адресе које не припадају овим доменима. На пример, ако се постави атрибут `allowed_domains` на *[„www.audible.com"]*, паук ће посетити само веб-адресе које припадају домену *www.audible.com*. Ово је корисно уколико је неопходно ограничити паука на одређени веб-сајт.

Атрибути `start_urls` и `start_requests` се користе за дефинисање почетних веб-адреса које паук треба да посети, али постоји неколико разлика

између њих. Атрибут `start_urls` је једноставнији начин за дефинисање фиксних почетних веб-адреса. Овај атрибут је листа веб-адреса која се поставља директно у пауку. Када је паук покренут, аутоматски ће посетити све веб-адресе из листе `start_urls` и обрадити одговоре. Са друге стране, атрибут `start_requests` је метода која се може имплементирати у класи паука како би се генерисали почетни захтеви. Уместо да се користи листа фиксних веб-адреса, могуће је динамички генерисати захтеве користећи метод `start_requests`. Ова метода треба да врати итератор објеката `Request`, који садржи информације о веб-адреси и *callback* методи која ће се позвати за сваки од тих захтева. Ово омогућава флексибилност у генерисању почетних захтева, на пример, могуће је прочитати веб-адресе из датотеке, базе података и других извора.

Шаблон *CrawlSpider*

Шаблон *CrawlSpider* је напреднији шаблон паука у библиотеци *Scrapy* који је посебно дизајниран за скрејпинг веб-страница. Када се користи овај шаблон, прво се креира класа паука која наслеђује класу `CrawlSpider`. У оквиру паука, дефинишу се правила за праћење веб-адреса на веб-сајту користећи атрибут `rules`. Ова правила омогућавају пауку да аутоматски обиђе више страница на веб-сајту. Када паук добије одговор са веб-странице, аутоматски се примењују правила за праћење веб-адреса и позива се метод `parse` за обраду одговора.

Атрибут `rules` се користи за дефинисање правила за аутоматско праћење и претрагу веб-страница са више нивоа линкова. Атрибут `rules` је листа објеката правила, где свако правило дефинише како треба поступати са одређеним типом линкова на страницама. Свако правило се састоји од неколико делова:

1. `Rule.link_extractor` — Овде се дефинише како треба издвојити линкове са странице. Може се користити *CSS* селектор или *XPath* израз да би се лоцирали линкови на страници.

2. `Rule.callback` — Ово је *callback* метода која се позива када се пронађе одговарајући линк на страници и користи се за обраду одговора и издвајање података са странице.

3. `Rule.follow` — Овај параметар дефинише да ли треба пратити линкове пронађене на тренутној страници и аутоматски прећи на њих. Ако је параметар постављен на вредност *True* паук ће аутоматски прећи на те линкове и применити правила на њима.

Креирање паука

За креирање паука, неопходно је извршити команду

```
scrapy genspider ime_spajdera url_veb_stranice
```

замењујући *ime_spajdera* са именом паука и *url_veb_stranice* са веб-адресом странице са које је потребно преузети податке. Ова команда креира нову датотеку која садржи шаблон Пајтон скрипте за паука са задатим именом и почетном веб-адресом са које ће се преузети подаци [11].

Фајл који се добија написаном изнад наредбом је могуће видети у коду на листингу 3.12. Овај фајл почиње са импортом потребних модула. Затим се креира класа паука која наслеђује класу `scrapy.Spider`. У овом делу фајла потребно је допунити класу паука са основним својствима и функционалностима, као што су име паука, дозвољени домени и почетна веб-адреса. Могуће је додати функције за парсирање веб-страница, издвајање података, слање нових захтева и обраду добијених резултата. Такође, могуће је конфигурисати паука у складу са потребама пројекта, на пример, лимитирањем броја захтева или постављањем других параметара.

```
1 import scrapy
2
3 class AudibleSearchSinglePageSpider(scrapy.Spider):
4     name = 'audible_search_single_page'
5     allowed_domains = ['www.audible.com']
6     start_urls = ['http://www.audible.com/']
7
8     def parse(self, response):
9         pass
```

Listing 3.12: Шаблон паука

.

Покретање паука

За покретање паука у библиотеци *Scrapy* неопходно је позиционирати се у оквиру терминала у директоријум где се налази паук. Затим, извршити команду

```
scrapy crawl ime_pauka
```

замењујући *ime_pauka* са именом паука које дефинисано у оквиру променљиве *name* унутар компоненте паука.

Додатно, постоји могућност да се при покретању паука додатно конфигуришу његови аргументи и опције. На пример, могуће је додати опцију *-o* за чување излазних података у жељеном формату, као што су *CSV* или *JSON*. Покретање паука са конкретно подешеном опцијом *-o rezultati.csv* би значило да ће се резултати сачувати у формату *CSV* у датотеци са називом *rezultati.csv* [11].

```
scrapy crawl ime_pauka -o rezultati.csv
```

Покретање паука омогућава да се започне процес прикупљања података са одабране веб-странице.

Лоцирање елемената

У библиотеци *Scrapy*, лоцирање елемената на веб-страници се најчешће врши коришћењем селектора *CSS* и *XPath*. Ови селектори омогућавају прецизно проналажење одређених елемената на основу њихових атрибута, тагова, класа и других карактеристика.

Селектори *CSS* се користе за проналажење и прикупљање података са веб-страница. Могуће је лоцирати појединачне елементе, више елемената или приступити елементима угњежденим унутар других елеманата. Библиотека *Scrapy* такође пружа подршку за напредне псеудо-класе *CSS* и псеудо-елементе што омогућава већу контролу при лоцирању елемената.

Псеудо-класа представља додатну ознаку која се може додати селектору за циљање елемената који имају одређена својства. На пример, псеудо-класа *:hover* циља елемент када је миш преко њега, док псеудо-класа *:active* циља елемент када је активан (кликнут) итд. Псеудо-елемент представља додатни

део селектора који омогућава приступ и стилизовање одређеног дела елемента, који не постоји у самом *HTML* коду. На пример, псеудо-елемент *::before* се користи за додавање садржаја испред елемента, а псеудо-елемент *::after* за додавање садржаја иза елемента.

У коду на листингу 3.13 је приказан пример лоцирања елемента селектором *CSS* у библиотеци *Scrapy*. У овом случају, елемент са класом *„adbl-impression-container"* унутар *div* елемента биће лоциран на страници и смештен у променљиву *container*. Овај пример показује како се селектори *CSS* могу користити за прецизно лоцирање жељених елемената на веб-страницама.

```
1 def parse(self, response):
2     container = response.css('div.adbl-impression-container')
```

Listing 3.13: Лоцирање елемента селектором *CSS*

.

Могуће је користити различите изразе и функције *XPath* како би циљано приступили елементима на основу њихових атрибута, тагова, текста или њихове позиције у документу.

У коду на листингу 3.14 је приказан пример лоцирања елемента селектором *XPath* у библиотеци *Scrapy*. У овом примеру, променљива *heading* ће садржати све елементе *h3* који садрже класу *„bc-heading"*.

```
1 def parse(self, response):
2     heading = book.xpath('.//h3[contains(@class, "bc-heading")]')
```

Listing 3.14: Лоцирање елемента селектором *XPath*

Након што се лоцирају жељени елементи, могуће је користити методе као што су `get()` или `getall()` за издвајање текста или атрибута тих елемената. Метод `get()` се користи када се очекује један елемент, а враћа текст или вредност атрибута тог елемента. Метод `getall()` се користи када се очекује више елемената и враћа листу са свим текстовима или вредностима атрибута тих елемената.

У коду на листингу 3.15 је приказан пример лоцирања елемента помоћу селектора *XPath* и *CSS* у библиотеци *Scrapy*. У овом примеру, променљива `book_authors` се допуњава информацијом о ауторима. Прво се помоћу селектора *XPath* лоцирају сви елементи *li* са класом *„authorLabel"*. Затим се примењује селектор *CSS a::text* на добијену листу елемената, што омогућава издвајање самог текста унутар елемената *a*. На крају, са помоћу метода `get()`

и `strip()` се издваја и чисти текст. Ово илуструје комбинацију различитих селектора, *XPath* и *CSS*, за лоцирање и издвајање потребних информација из веб-страница у оквиру библиотеке *Scrapy*.

```
1 def parse(self, response):
2     book_authors.append(book.xpath('.//li[contains(@class, "authorLabel
      ")]').css('a::text').get().strip())
```

Listing 3.15: Лоцирање елемента селекторима *XPath* и *CSS*

Прикупљање података са више веб-страница

Скрејповање података са више страница користећи библиотеку *Scrapy* је уобичајена потреба приликом обраде веб-страница. Библиотека *Scrapy* пружа флексибилност за навигацију кроз различите странице и прикупљање података са сваке од њих.

Прво, потребно је одредити почетну веб-страницу са које ће се прикупљати подаци. Затим, унутар паук компоненте треба имплементирати методу `start_requests()` која генерише захтеве за почетну веб-страницу. Ова метода враћа објекат типа *Request* са одговорајућим веб-страницама које треба посетити. Даље, треба дефинисати методу `parse()` која се бави обрадом одговора са веб-странице. У овој методи се лоцирају жељени подаци и може се имплементирати логика за навигацију на следећу страницу, ако је потребно. Могуће је генерисати нове захтеве за следеће странице које треба посетити у методи `parse()` додавањем нових веб-страница на листу захтева `yield Request(url)` или променом веб-странице на основу информација са тренутне странице.

У коду на листингу 3.16 је приказан пример лоцирања елемента селекторима *CSS* и *XPath* у библиотеци *Scrapy*. У овом примеру, после извршавања лоцирања елемента и сакупљања информација, користи се *XPath* селектор *//span[contains(@class, „nextButton")]//a/@href* да би се добио *URL* следеће странице. Ова *URL* вредност се користи за генерисање новог захтева и поновно извршавање методе `parse()`. Ако постоји следећа страница, захтев се шаље користећи `scrapy.Request`, а примењена је повратна позивна функција `self.parse`. Овај процес се понавља све док постоји следећа страница за обраду.

```
1     def parse(self, response):
```

```
2         container = response.css('div.adbl-impression-container')
3         titles, authors, release_dates, prices = audible_shared.
    collect_books_info(container)
4         self.book_titles.extend(titles)
5         self.book_authors.extend(authors)
6         self.book_release_dates.extend(release_dates)
7         self.book_prices.extend(prices)
8
9         next_page_url = 'http://www.audible.com' + response.xpath('//
    span[contains(@class, "nextButton")]//a/@href').get()
10        if next_page_url:
11            yield scrapy.Request(next_page_url, callback=self.parse)
```

Listing 3.16: Прикупљање података са више страница

Прикупљање података са више веб-страница помоћу дефинисаних правила

Правила(енг. *Rules*) је класа у оквиру библиотеке *Scrapy* која се користи за дефинисање правила за праћење линкова и управљање понашањем паука приликом претраживања веб-страница. Она омогућава аутоматизовану навигацију и сакупљање података са различитих страница унутар исте веб-локације.

Када се користе правила, неопходно је дефинисати одређене обрасце линкова које је потребно пратити, као и одговарајуће методе које ће се позвати за обраду сваке пронађене странице. Ова функционалност је посебно корисна када је потребно прећи кроз више страница, на пример, странице са пагинацијом или категоријама.

Приликом коришћења правила, обично се користи у комбинацији са класом `LinkExtractor`, која екстрахује линкове који одговарају задатом обрасцу. Ови линкови се затим прате и обрађују према дефинисаним правилима [6].

Коришћење правила омогућава ефикасније и једноставније управљање навигацијом и сакупљањем података на веб-страницама, елиминишући потребу за ручним праћењем линкова и пагинацијом.

У коду на листингу 3.17 је приказан пример употребе правила за прикупљање података са више страница на веб-локацији. У методи `start_requests()` се дефинише почетна веб-страница са које се започиње процес прикупљања. Затим, у променљивој `rules` се дефинише правило за праћење линкова унутар

објекта `LinkExtractor`, који ће бити обрађени у `parse_item()` методи. Поставља се и аргумент `follow` како би се пронађени линкови наставили пратити и скреповање на следећим страницама. У методи `parse_item()` се обрађује одговор са веб-странице.

```
1 class AudibleSearchAllPages(CrawlSpider):
2     ...
3     def start_requests(self):
4         yield scrapy.Request(url='http://www.audible.com/search')
5 
6     rules = (
7         Rule(LinkExtractor(restrict_xpaths=("//div[contains(@class, '
      categories')]//a")), callback='parse_item', follow=True),
8     )
9 
10    def parse_item(self, response):
11        ...
```

Listing 3.17: Прикупљање података са више страница коришћењем правила

Прикупљање података са *API*

Библиотека *Scrapy* је првенствено намењена за прикупљање података са веб-страница. Међутим, библиотека такође пружа могућност прикупљања података директно са *API* ендпоинта(енгл. *endpoint*).

Прикупљање података директно са *API* користећи *Scrapy* има следеће предности: омогућава аутоматизацију процеса без ручне интервенције, ефикасан је за обраду великог броја података, лако се скалира за рад са више ендпоинта или за велики број захтева и има уграђену подршку за аутентификацију и пагинацију, што је корисно за *API* са сложеном структуром или захтевима за аутентификацију.

За прикупљање података директно са *API* ендпоинта, неопходно је дефинисати класу паука са променљивом `start_urls` која садрже *URL* адресе *API* ендпоинта. У `parse` методи се имплементира логика за обраду одговора, издвајање података и додатна обрада, што је и приказано у коду на листингу 3.18.

```
1 class MySpider(scrapy.Spider):
2     ...
3     start_urls = ['http://api_endpoint_url']
```

```
4
5       def parse(self, response):
6           # Ovde implementirati obradu odgovora sa API endpointa
7           pass
```

Listing 3.18: Прикупљање података директно са *API*

Изазови аутентикације и аутоматизације

Претходно поменуте изазове аутентикације и аутоматизације у секцији 3.2 је такође могуће превазићи у оквиру библиотеке *Scrapy*.

У коду на листингу 3.19 је приказан пример како се користи библиотека *Scrapy* за аутентикацију на веб-страницу. У методи *parse* се креира објекат `form_data` који садржи корисничко име и лозинку. Затим се користи наредба `yield` за слање *POST* захтева на одређену веб-страницу са подацима `form_data` и прослеђује се позив методе `self.after_login` за обраду одговора. Овај пример показује како се користи метода `FormRequest` и `yield` да би се аутоматизовала аутентикација на веб-страницу и обрадио одговор сервера.

```
1 def parse(self, response):
2     form_data = {
3         'username': 'my_username',
4         'password': 'my_password',
5     }
6
7     yield scrapy.FormRequest(url='http://www.example.com/login',
    formdata=form_data, callback=self.after_login)
8
9     def after_login(self, response):
10        if response.status == 200:
11            self.logger.info('Ulogovani ste!')
12        else:
13            self.logger.error('Neuspesan login!')
```

Listing 3.19: Аутентикација у библиотеци *Scrapy*

Ограничења библиотеке *Scrapy* за прикупљање података са динамичких веб-страница

Библиотека *Scrapy* није намењена за директно прикупљање података са динамичких веб-страница. Разлог за то је што библиотека *Scrapy* ради на основу *HTTP* захтева и одговора, а не поседује уграђену подршку за рендеровање и извршавање Јаваскрипт кода.

Динамичке веб-странице често користе програмски језик Јаваскрипт како би генерисале или модификовале садржај након што се страница учита. Ово може укључивати динамичко учитавање података и интерактивне елементе. Будући да библиотека *Scrapy* не извршава Јаваскрипт кôд приликом прикупљања података, већ само преузима *HTML* садржај веб-странице, неће бити у могућности да прикупи податке који се генеришу или модификују помоћу Јаваскрипт-а.

Да бисте прикупили податке са динамичких веб-страница, потребно је користити додатне алате или библиотеке за рендеровање Јаваскрипт кода, као што је *SPLASH* или *Selenium*. Ови алати омогућавају симулирање Веб-претраживача и извршавање Јаваскрипт кода, што омогућава прикупљање података са динамичких веб-страница.

Алат *SPLASH* и програмски језик *LUA*

Алат *SPLASH* се користи за рендеровање Јаваскрипт кода на веб-страницама, а користи програмски језик *LUA* за дефинисање скрипти. Алат *SPLASH* омогућава извршавање Јаваскрипт кода приликом прикупљања података, што је посебно корисно за веб-странице које се динамички генеришу помоћу Јаваскрипт-а. Алат *SPLASH* се може користити као додатак библиотеци *Scrapy* како би се омогућило ефикасно прикупљање података са веб-страница које зависе од Јаваскрипт-а и имају комплекснију логику приказа садржаја.

Програмски језик *LUA* је програмски језик отвореног кода који је дизајниран да буде једноставан за интеграцију и користи се у различитим областима, једна од којих је веб-скрејпинг. Програмски језик *LUA* је интерпретирани језик, који подржава процедурално, функционално и објектно-оријентисано програмирање. Такође је флексибилан и пружа богат скуп библиотека и модула за различите намене. Захваљујући свим овим карактеристикама, програмски језик *LUA* је постао популаран избор за различите пројекте у свету

програмирања.

У контексту *SPLASH*, програмски језик *LUA* се користи за писање скрипти које дефинишу како ће се извршавати Јаваскрипт кôд на веб-страницама. Скрипте написане у програмском језику *LUA* омогућавају контролу над рендеровањем странице, приступ *DOM* елементима, манипулацију подацима и друге интеракције са веб-страницом. Алат *SPLASH* користи програмски језик *LUA* да би извршио ове скрипте и добио резултате које је могуће даље обрадити. Тако алат *SPLASH* и програмски језик *LUA* заједно омогућавају ефикасно прикупљање података са веб-страница.

Интеграција Докера и алата *SPLASH*

Докер (енг. *Docker*) је платформа дизајнирана да помогне програмерима да граде, деле и покрећу модерне апликације [10]. Докер представља платформу која омогућава паковање, дистрибуцију и извршавање апликација у контејнерима (енг. *container*). Контејнери су изолована окружења која укључују све потребне зависности за покретање апликације.

Докер омогућава лако конфигурисање и покретање алата *SPLASH* без потребе за инсталацијом свих додатних зависности и конфигурација на локалном систему. Он такође пружа могућност да алат *SPLASH* ради у изолованом окружењу, без утицаја на друге апликације и системске ресурсе.

Коришћењем наредби

```
docker pull scrapinghub/splash
docker run -it -p 8050:8050 scrapinghub/splash
```

је могуће покренути алат *SPLASH* за веб-скрејпинг. Прва наредба преузима Докер слику која садржи *SPLASH*, док друга наредба покреће *SPLASH* контејнер на локалном систему и омогућава приступ алату преко одређеног порта.

Увод у алат *SPLASH*

SPLASH је алат за прикупљање података са веб-страница који се базира на Веб-прегледачу, специфичној Веб-прегледачкој машини (енг. *engine*) и програмском језику *LUA*. Омогућава веб-скрејпинг веб-страница које користе Јаваскрипт и друге динамичке елементе, што га чини веома корисним у прикупљању података.

У коду на листингу 3.20 приказан је пример употребе алата *SPLASH* за дохватање *HTML* кода веб-странице. Функција `main` прихвата два аргумента: `splash` и `args`. Унутар функције, прво се позива метод `splash:go` са прослеђеним *URL* као аргументом, што отвара жељену веб-страницу. Затим се позива метод `splash:wait` који чека 2 секунди како би се осигурало да је сав садржај на страници у потпуности учитан. На крају, враћа се објекат са пољем `html` које садржи *HTML* садржај учитане странице. Овај пример илуструје основну структуру функције у алату *SPLASH* и коришћење неких од основних метода за контролу Веб-прегледача и добијање *HTML* садржаја.

```
1 function main(splash, args)
2   assert(splash:go(args.url))
3   assert(splash:wait(2))
4   return {
5     html = splash:html(),
6   }
7 end
```

Listing 3.20: Пример употребе алата *SPLASH*

Лоцирање елемената у алату *SPLASH*

Проналажење и манипулација елемената на веб-страницама су кључне функционалности при веб-скрејпингу и аутоматизацији. За ове сврхе, *SPLASH* је моћан алат који омогућава извршавање Јаваскрипт кода на веб-страницама и приступање *DOM* елементима.

Приликом употребе алата *SPLASH* постоји неколико основних метода и приступа који се користе за проналажење и манипулацију елемената на веб-страницама. Могуће је користити селекторе *CSS* или изразе *XPath* за идентификацију жељених елемената. Алат *SPLASH* пружа методе као што су `splash:select` и `splash:select_all` за проналажење елемената на основу селектора *CSS*, као и методу `splash:select` за проналажење елемената путем израза *XPath*. Такође, алат *SPLASH* омогућава приступ и манипулацију са *DOM* елементима веб-странице. Метода `splash:evaljs` се користи за извршавање Јаваскрипт кода који манипулише са *DOM* елементима [2].

Неколико конкретних примера лоцирања елемената на веб-страницама користећи алат *SPLASH*:

Лоцирање по идентификатору

```
local element = splash:select(,,#element_id'')
```

Лоцирање по селектору *CSS*

```
local element = splash:select(,,div.container > p'')
```

Лоцирање по изразу *XPath*

```
local element = splash:select(,,//div[@class='container']/p[2]'')
```

Манипулација *DOM* елементима

```
splash:evaljs(,,document.getElementById('ID HERE').innerHTML = 'TEXT
HERE''')
```

У коду на листингу 3.21 приказан је пример скрипте у алату *SPLASH* написане у програмском језику *LUA*. Ова скрипта се користи за аутоматизацију интеракције са веб-страницом. Прво се *SPLASH* прегледач навигира на *URL* који је прослеђен као аргумент. Након што се страница учита, скрипта чека 3 секунде како би се осигурало да је страница потпуно приказана. Затим се користи селектор *CSS „btn.login-btn"* да би пронашао елемент са задатом класом, који се затим кликне помоћу функције `mouse_click()`, а затим се чека још 3 секунде како би се страница поново приказала. На крају, мења се величина приказа да би сав садржај био видљив, и враћају се резултати у облику *HTML* кода `splash:html()`.

```
1 function main(splash, args)
2     assert(splash:go(args.url))
3     assert(splash:wait(3))
4     login_btn = assert(splash:select("btn.login-btn"))
5     login_btn: mouse_click()
6     assert (splash:wait(3))
7     return {splash: html()}
8 end
```

Listing 3.21: Пример кода програмског језика *LUA*

Интеграција библиотеке *Scrapy* и алата *SPLASH*

За интеграцију библиотеке *Scrapy* и алата *SPLASH* могуће је користити библиотеку *scrapy-splash* која пружа подршку за алат *SPLASH* у оквиру библиотеке *Scrapy*. За инсталацију поменуте библиотеке неопходно је покренути следећу наредбу из командне линије:

```
pip install scrapy-splash
```

У коду на листингу 3.22 приказан је пример интеграције библиотеке *Scrapy* и алата *SPLASH*. У овом примеру, кроз функцију `start_requests` је дефинисана почетна тачка за комуникацију између библиотеке *Scrapy* и алата *SPLASH*. Користећи објекат класе `SplashRequest`, шаље се захтев за прикупљање података одређене веб-странице. Захтев садржи *URL* странице, функцију `self.parse` која се позива по добијању одговора, аргумент `endpoint` и аргумент `lua_source` који садржи *Lua* скрипту која ће се извршити у алату *SPLASH*. Када се постави аргумент `endpoint='execute'`, то значи да је *Lua* скрипту неопходно извршити унутар сервера *SPLASH*. Резултат извршавања скрипте се затим враћа као одговор. У функцији `parse`, обрада се врши над добијеним одговором, у овом случају исписивањем одговора.

```
script = '''
    function main(splash, args)
        assert(splash:go(args.url))
        assert(splash:wait(3))
        login_btn = assert(splash:select("btn.login-btn"))
        login_btn: mouse_click()
        assert (splash:wait(3))
        return {splash: html()}
    end
'''
def start_requests(self):
    yield SplashRequest(url='https://www.example.com', callback=self.
    parse, endpoint='execute', args {'lua_source':self.script})

def parse(self, response):
    print(response.body)
```

Listing 3.22: Пример интеграције библиотеке *Scrapy* и алата *SPLASH*

Глава 4

Поређење особина библиотека

Поређење карактеристика и особина и перформанси прикупљања података са библиотекама *BeautifulSoup*, *Selenium* и *Selenium* може бити корисно како бисте донели информисане одлуке о избору библиотеке у складу са својим потребама.

Библиотека *BeautifulSoup* је једноставна за коришћење и има интуитивни интерфејс. Омогућава лако парсирање и манипулацију *HTML* кодом, што олакшава извлачење података са веб-страница. Међутим, она није директно намењена за аутоматизацију интеракције са динамичким веб-страницама или за прикупљање великих количина података.

Библиотека *Selenium* је посебно дизајнирана за аутоматизацију Веб-прегледача. Омогућава контролу и симулацију корисничких интеракција на веб-страницама, укључујући кликтање дугмића, попуњавање форми и скроловање. Ово је корисно када је потребно прикупити податке са динамичких веб-страница или када је неопходно извршити акције које захтевају интеракцију са страницом. Међутим, библиотека *Selenium* може бити сложенија за подешавање и захтева додатне инсталације и конфигурације.

Библиотека *Scrapy* пружа целокупно решење за прикупљање података са веб-страница, укључујући аутоматизацију, управљање сесијама, извлачење и чување података. Библиотека *Scrapy* има подршку за паралелно прикупљање података и могућност дефинисања сложених правила за извлачење података са различитих страница. Ова библиотека је посебно погодна за велике и комплексне пројекте прикупљања података. Међутим, библиотека *Scrapy* захтева одређено време за учење и упознавање са њеном архитектуром и концептима.

4.1 Поређење имплементације прикупљања података са једне веб-странице

Када упоредимо имплементације кодова који користе библиотеке *BeautifulSoup*, *Selenium* и *Scrapy* за веб-скрејпинг једне странице са веб-сајта, могу се приметити неке разлике у приступу и коду:

1. Имплементација уз помоћ библиотеке *BeautifulSoup*:
   a) Библиотека *requests* се користи за преузимање *HTML* кода веб-страница.
   b) Затим се страница прослеђује у конструктор `BeautifulSoup` и креира објекат класе `BeautifulSoup`. Након тога, користи се метод `find` за проналажење блока са подацима. Функција `collect_books_info` обилази све елементе на страници и издваја наслове, ауторе, датуме издавања и цене књига.
   c) Ова имплементација је једноставна и приступачна, али не подржава интеракцију са динамичким садржајем.
2. Имплементација уз помоћ библиотеке *Selenium*:
   a) Неопходно иницијализовати веб-драјвер, у овом случају драјвер *Google Chrome*.
   b) Затим се користе методе из библиотеке *Selenium* за приступање веб-страницама и проналажење елемената. У функцији `collect_books_info`, користе се методе библиотеке *Selenium* за проналажење и издвајање података.
   c) Ова имплементација омогућава интеракцију са динамичким садржајем, али може бити спорија у извршавању у поређењу са осталим решењима.
3. Имплементација уз помоћ библиотеке *Scrapy*:
   a) Имплементација се базира на посебној класи `scrapy.Spider` и користи функцију `parse` за издвајање података из одговора.
   b) Имплементација је део ширег оквира који омогућава асинхрони приступ, скалабилност и аутоматско праћење линкова.

c) У функцији `collect_books_info`, користе се селектори из библиотеке *Scrapy* за проналажење и издвајање података.

d) Ова имплементација може бити најснажнија и најфлексибилнија за скрејпинг, али захтева неколико додатних конфигурационих корака.

4.2 Поређење времена извршавања

Време извршавања варира у зависности од различитих фактора, као што су брзина интернет везе, величина веб-страница, број података који се прикупљају и ефикасност саме имплементације. Ево неколико информација о временском извршавању на основу кода:

1. Имплементација уз помоћ библиотеке *BeautifulSoup*:

   a) Време извршавања ове имплементације може бити релативно брже у односу на *Selenium* и *Scrapy* имплементације.

   b) Библиотека *BeautifulSoup* је специјализована за парсирање *HTML* и омогућава ефикасно претраживање и екстракцију података.

   c) Ова имплементација се извршава директно на локалном рачунару, без потребе за покретањем прегледача, што може резултовати бржим временом извршавања.

2. Имплементација уз помоћ библиотеке *Selenium*:

   a) Време извршавања ове имплементације може бити спорије у односу на *BeautifulSoup* и *Scrapy* имплементације.

   b) Будући да библиотека *Selenium* користи стварни прегледач за извршавање радњи, укључујући приказивање страница, кликове и чекања, постоји додатни временски трошак.

3. Имплементација уз помоћ библиотеке *Scrapy*:

   a) Време извршавања ове имплементације може бити конкурентно у односу на *BeautifulSoup* и *Selenium* имплементације.

   b) Библиотека *Scrapy* је дизајнирана да буде ефикасна и брза у прикупљању података са веб-страница.

c) Ова имплементација користи асинхроно програмирање и оптимизовану обраду података како би смањила време извршавања.

Важно је напоменути да време извршавања може значајно варирати у зависности од специфичних услова скрејпинг пројекта, као и од перформанси рачунара и ресурса. Препоручује се тестирање различитих имплементација са стварним подацима како би се утврдила најбоља опција у конкретном случају.

4.3 Поређење потрошње меморије

Када се упоређује потрошња меморије између ових три имплементације, такође постоје разлике које треба узети у обзир. Ево неколико информација о потрошњи меморије на основу наведеног кода:

1. Имплементација уз помоћ библиотеке *BeautifulSoup*:

   a) Ова имплементација користи Пајтон објекте за парсирање и обраду *HTML* кода.

   b) Меморија коју ова имплементација користи зависи од величине *HTML* кода веб-странице и броја објеката које ствара током процеса парсирања.

   c) Будући да је парсирање и обрада *HTML* кода извршена у меморији, могуће је да ова имплементација користи више меморије у односу на друге имплементације.

2. Имплементација уз помоћ библиотеке *Selenium*:

   a) Ова имплементација користи библиотеку *Selenium* која комуницира са прегледачем и извршава радње као што су кликови на дугмад и прикупљање података.

   b) Меморија коју ова имплементација користи зависи од величине прегледача и броја отворених табова приликом извршавања скрејпинга.

   c) Ова имплементација може користити додатну меморију због покретања прегледача и отварања веб-страница.

3. Имплементација уз помоћ библиотеке *Scrapy*:

a) Библиотека *Scrapy* је оптимизована за ефикасну потрошњу меморије.

b) Библиотека *Scrapy* има уграђени механизам управљања меморијом који омогућава ефикасно прикупљање, обраду и чување података.

c) У зависности од конфигурације и величине прикупљених података, потрошња меморије може бити мања у односу на претходне имплементације.

Укратко, *BeautifulSoup* имплементација може користити више меморије због парсирања и обраде *HTML* кода, док *Selenium* имплементација може користити додатну меморију због покретања прегледача. *Scrapy* имплементација је оптимизована за ефикасну потрошњу меморије. Избор имплементације треба да се заснива на величини и комплексности скрејпинг пројекта, као и на доступној меморији и ресурсима.

4.4 Поређење скалабилности

Када се упоређује скалабилност између ових три имплементације, такође постоје разлике које треба узети у обзир. Ево неколико информација о скалабилности на основу наведеног кода:

1. Имплементација уз помоћ библиотеке *BeautifulSoup*:

   a) Ова имплементација није инхерентно скалабилна и може се сусрести са изазовима при обради веома великих *HTML* страница или прикупљању података са великог броја веб-страница.

   b) Парсирање и обрада *HTML* кода се врше секвенцијално, што може ограничити брзину обраде и утицати на време извршавања скрејпинга.

2. Имплементација уз помоћ библиотеке *Selenium*:

   a) *Selenium* имплементација може бити скалабилна у одређеној мери, посебно када се користи за паралелно извршавање скрејпинга на више прегледача или отворених табова.

b) Помоћу одговарајућих техника и алата, могуће је повећати скалабилност ове имплементације, омогућавајући брже прикупљање података са већег броја веб-страница.

3. Имплементација уз помоћ библиотеке *Scrapy*:

a) *Scrapy* је дизајниран са скалабилношћу у виду.

b) *Scrapy* користи асинхрони модел обраде који омогућава паралелно извршавање скрејпинга на више веб-страница.

c) Поред тога, *Scrapy* има уграђену подршку за расподелу и управљање ресурсима, као што су време чекања, брзина прикупљања података и управљање захтевима. Ово омогућава високу скалабилност при обради великог броја веб-страница и прикупљању података у кратком временском периоду.

Укратко, *BeautifulSoup* имплементација има ограничену скалабилност, док се скалабилност *Selenium* имплементације може побољшати коришћењем одговарајућих техника. *Scrapy* имплементација је дизајнирана са скалабилношћу у виду и пружа могућности за паралелно и ефикасно прикупљање података са великог броја веб-страница. При одабиру имплементације, треба узети у обзир величину пројекта, број веб страница које треба обрадити и потребу за скалабилношћу.

4.5 Поређење промене циљног веб-сајта

Промена циљног веб-сајта у све три имплементације, *BeautifulSoup*, *Selenium* и *Scrapy*, може захтевати различите нивое комплексности. Ево описа колико је комплексно променити циљни веб-сајт у свакој од ових имплементација:

1. Имплементација уз помоћ библиотеке *BeautifulSoup*: Промена циљног веб-сајта у *BeautifulSoup* имплементацији може бити умерено комплексна. Овде је неопходно променити парсирање и обраду *HTML* кода у складу са структуром и особинама новог веб-сајта. Потребно је анализирати нову *HTML* структуру, променити селекторе елемената и прилагодити постојећу логику за прикупљање података.

2. Имплементација уз помоћ библиотеке *Selenium*: Промена циљног веб-сајта у *Selenium* имплементацији може бити захтевнија у односу на *BeautifulSoup*. Уз промену парсирања и обраде *HTML* кода, потребно је ажурирати скрипте које комуницирају са Веб-прегледачем. Ово може укључивати измене у кликовима на дугмад, уносу података, преласку на различите странице итд.

3. Имплементација уз помоћ библиотеке *Scrapy*: Промена циљног веб-сајта у *Scrapy* имплементацији је најзахтевнија од све три. Како *Scrapy* имплементација користи фрејмворк за прикупљање, обраду и складиштење података, промена циљног веб-сајта може захтевати веће промене у структури и понашању спајдера у односу на претходни веб-сајт.

У свим случајевима, промена циљног веб-сајта захтева претходну анализу нове HTML структуре, селектора елемената и логике прикупљања података. Комплексност измена ће зависити од структуре и сложености новог веб-сајта.

4.6 Прикупљени подаци

Приликом прикупљања података са веб-сајта *Audible* уз помоћ све три имплементације креиран је фајл у формату *csv*. Ови подаци представљају информације о наслову, аутору, датуму објављивања и цени књига са различитих веб-страница. Прикупљање података је извршено за једну страницу, више страница и читав веб-сајт уз помоћ три библиотеке: *BeautifulSoup*, *Selenium* и *Scrapy*.

Ови подаци могу бити корисни за различите сврхе. На пример, могу се користити за анализу популарности одређених аутора, праћење трендова у издавању књига или упоређивање цена књига. Такође, ови подаци могу бити основа за даљу обраду, као што је генерисање извештаја, визуализација или машинско учење.

Глава 5

Закључак

У овом раду анализирани су различити алати и технике за прикупљање података са веб-страница. Разматрани су изазови са којима се може суочити у процесу прикупљања података и идентификација елемената унутар *HTML* кода. Такође, прегледани су библиотека *BeautifulSoup*, библиотека *Selenium* и библиотека *Scrapy*, као и алат *SPLASH*.

TODO

Библиографија

[1] pip documentation v23.1.1. on-line at: https://pip.pypa.io/en/stable/, author = The pip developers, Made with Sphinx and @pradyunsg's Furo.

[2] Splash - A javascript rendering service, 2019. on-line at: https://splash.readthedocs.io/en/stable/, author = © Copyright 2019, Scrapinghub Revision 6dc78f59.

[3] Keio) 1999 W3C® (MIT, INRIA. XPath. on-line at: https://www.w3.org/TR/1999/REC-xpath-19991116/.

[4] Python Software Foundation 2001-2023. html.parser — Simple HTML and XHTML parser. on-line at: https://docs.python.org/3/library/html.parser.html.

[5] Stefan Behnel and Martijn Faassen. Parsing XML and HTML with lxml. on-line at: https://lxml.de/parsing.html.

[6] Maintained by Zyte (formerly Scrapinghub) and many other contributors. Scrapy. on-line at: https://scrapy.org/.

[7] Osmar Castrillo-Fernández. Web scraping: Applications and tools.

[8] 2023 Software Freedom Conservancy. Selenium. on-line at: https://www.selenium.dev/documentation/.

[9] Sam Sneddon Copyright 2006 2013, James Graham and contributors Revision 3e500bb6. html5lib. on-line at: https://html5lib.readthedocs.io/en/latest/.

[10] 2023 Docker Inc. Docker.

[11] Ryan Mitchell. Web scraping with python. In *Web Scraping with Python*, 2015.

[12] Emil Persson. Evaluating tools and techniques for web scraping.

[13] Leonard Richardson. Beautiful Soup Documentation, 2004-2023. on-line at: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`.

[14] 2000-2010 Carnegie Mellon University. CAPTCHA. on-line at: `http://www.captcha.net/`.

Биографија аутора

Зорана Гајић, рођена је 04.11.1997. у Москви, где је завршила први разред основне школе, због чега је по повратку у Београд наставила и завршила основно и средње образовање у Руској школи при Амбасади Руске Федерације са одликованом златном медаљом од стране Руске Федерације за посебна достигнућа у настави. Смер Информатика на Математичком факултету Универзитета у Београду уписала је 2015. године, а завршила у јулу 2019. године са просечном оценом 8.8. Након завршених основних студија, уписала је мастер студије информатике на истом факултету.

Од септембра 2020. године је запослена у компанији *Smart Apartment Data* где ради у фронт-енд тиму на изради апликације која нуди поуздан извор тржишне интелигенције за стамбену индустрију. Нуде свеобухватне платформе података за власнике, брокере, компаније, тимове и добављаче којима су потребне тачне детаљне информације за пословне одлуке и информисана улагања. Тренутно ради на позицији вође фронт-енд тима у истој фирми.