O'REILLY®

Wydanie II

# Ekstrakcja danych z językiem Python

POZYSKIWANIE DANYCH Z INTERNETU

Helion

Ryan Mitchell

Tytuł oryginału: Web Scraping with Python: Collecting More Data from the Modern Web, 2nd Edition

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-5635-1



Helion SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/ekspy2*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/ekspy2.zip*

Printed in Poland.

# Spis treści

ROZDZIAŁ 2.

# Zaawansowana analiza składniowa HTML

Michał Anioł, zapytany, jak udało mu się wykuć takie arcydzieło, jakim jest *Dawid*, udzielił słynnej odpowiedzi: „To proste. Usunąłem te elementy, które nie były Dawidem”.

W przeważającej liczbie aspektów ekstrakcja danych z internetu nie ma nic wspólnego z rzeźbiarstwem, jednak podczas wydobywania informacji ze skomplikowanych stron WWW wymaga podobnego podejścia. Możemy korzystać z wielu technik usuwających treści niebędące poszukiwaną przez nas zawartością aż do pozostawienia jedynie potrzebnych informacji. W tym rozdziale przyjrzymy się analizie składniowej skomplikowanych witryn internetowych w celu wyszukiwania jedynie interesujących nas informacji.

## Młotek nie zawsze jest potrzebny

Podczas mierzenia się ze znacznikowym węzłem gordyjskim kuszącą wizją jest zanurzenie się w ten chaos i stosowanie instrukcji wielowierszowych do wydobywania informacji. Pamiętaj jednak, że nieprzemyślane nawarstwianie omawianych w tym podrozdziale technik może prowadzić do powstania trudnego do poprawiania, delikatnego kodu. Zanim przejdziemy dalej, zastanówmy się nad pewnymi metodami pozwalającymi całkowicie uniknąć zaawansowanej analizy składniowej HTML.

Załóżmy, że masz jakąś docelową treść. Może to być nazwa, statystyka lub blok tekstowy. Być może treść ta jest zagrzebana pod 20 warstwami znaczników w bagnie HTML-owym pozbawionym znaczników pomocniczych lub atrybutów HTML. Powiedzmy, że zrezygnowałeś z ostrożności i próbujesz przeprowadzić ekstrakcję danych za pomocą poniższego wiersza:

```
bs.find_all('table')[4].find_all('tr')[2].find('td').find_all('div')[1].find('a')
```

Nie wygląda to zbyt dobrze. Pomijając aspekt estetyczny tego wiersza, każda najdrobniejsza nawet zmiana strony przez administratora może całkowicie rozregulować robota indeksującego. A jeżeli twórca strony postanowi dodać kolejną tabelę lub kolumnę z danymi? Co w przypadku, gdy wstawi nowy składnik (zawierający kilka znaczników `div`) na górze strony? Powyższy wiersz jest bardzo niestabilny i uzależniony od określonej struktury strony.

Jakie więc mamy możliwości?

- Poszukaj odnośnika „Wydrukuj tę stronę” lub wersji mobilnej tej strony, która może będzie miała lepiej sformatowaną strukturę HTML (więcej informacji o imitowaniu urządzeń mobilnych — i wyświetlaniu mobilnych wersji stron — znajdziesz w rozdziale 14.).
- Sprawdzaj informacje ukryte w pliku JavaScriptu. Pamiętaj, że w tym celu być może będziesz musiał analizować importowane pliki JavaScriptu. Na przykład kiedyś udało mi się pozyskać adresy (wraz ze współrzędnymi geograficznymi) w postaci elegancko sformatowanej tablicy poprzez zajrzenie do pliku JavaScriptu we wbudowanej aplikacji Google Maps wyświetlającej dokładną lokalizację każdego adresu.
- Częściej dotyczy to tytułów stron, ale informacje mogą występować w samym adresie URL strony.
- Jeżeli poszukiwane informacje są specyficzne dla danej strony, to masz pecha. W przeciwnym razie zastanów się nad innymi źródłami tych danych. Być może istnieje jeszcze jakaś strona zawierające te same informacje? Czy ta strona wyświetla dane pozyskane lub zgromadzone z innej witryny?

Ważne jest, aby nie zaczynać po prostu od wykopywania informacji i nie wskakiwać do dziury, z której już możesz się nie wydostać, zwłaszcza w przypadku głęboko zagrzebanych lub fatalnie sformatowanych danych. Weź głęboki oddech i pomyśl o alternatywach.

Jeżeli masz pewność, że nie istnieją takowe, w dalszej części rozdziału znajdziesz opis standardowych i kreatywnych sposobów dobierania znaczników na podstawie ich położenia, kontekstu, atrybutów i zawartości. Prezentowane tu techniki, jeżeli są prawidłowo wykorzystywane, pozwolą Ci zajść daleko w procesie pisania coraz stabilniejszych i pewniejszych pełzaczy.

# Kolejna porcja BeautifulSoup

W rozdziale 1. nauczyliśmy się instalować i uruchamiać bibliotekę *BeautifulSoup*, a także pojedynczo wybierać obiekty. Teraz dowiemy się, jak należy wyszukiwać znaczniki po ich atrybutach, pracować z listami znaczników i poruszać się po drzewach hierarchii analizy składniowej.

Niemal każda strona WWW zawiera arkusze stylów. Być może uważasz, że warstwa stylizacji strony internetowej została zaprojektowana przede wszystkim pod kątem przeglądarek, a jej interpretowanie przez człowieka jest niewłaściwe, jednak pojawienie się arkuszy CSS stanowi prawdziwe dobrodziejstwo dla robotów indeksujących. Style CSS bazują na różnicowaniu elementów HTML, które w przeciwnym wypadku musiałyby korzystać z tych samych znaczników, aby móc tworzyć ich różne style. Niektóre znaczniki mogą wyglądać następująco:

```
<span class="green"></span>
```

Inne wyglądają tak:

```
<span class="red"></span>
```

Pełzacze potrafią bez trudu rozróżnić te dwa znaczniki na podstawie klasy, na przykład mogą za pomocą biblioteki *BeautifulSoup* wyłapywać czerwony tekst, a całkowicie ignorować zielony. Szablony CSS dzięki tym atrybutom są w stanie we właściwy sposób stylizować wygląd strony, jest niemal pewne, że te klasy i atrybuty identyfikujące będą występować w większości współczesnych witryn internetowych.

Stwórzmy przykładowy pełzacz wydobywający dane ze strony umieszczonej pod adresem *http://www.pythonscraping.com/pages/warandpeace.html*.

Na tej stronie dialogi bohaterów powieści *Wojna i pokój* zostały oznaczone na czerwono, natomiast imiona — na zielono. Znaczniki `span` odnoszące się do tych dwóch klas CSS są widoczne w poniższym fragmencie kodu źródłowego strony:

```
<span class="red">Heavens! what a virulent attack!</span> replied
<span class="green">the prince</span>, not in the least disconcerted by this reception.
```

Możesz zająć się całą stroną i stworzyć obiekt `BeautifulSoup` za pomocą programu podobnego do użytego w rozdziale 1.:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
```

Za pomocą obiektu `BeautifulSoup` możemy wykorzystać funkcję `find_all` w celu wydobycia listy nazw własnych wyszukiwanych jedynie w tekście znajdującym się wewnątrz znaczników `<span class="green"></span>` (funkcja `find_all` jest niesamowicie wszechstronna i jeszcze wielokrotnie będziemy z niej korzystać):

```
nameList = bs.findAll('span', {'class':'green'})
for name in nameList:
    print(name.get_text())
```

Po uruchomieniu programu powinna zostać wyświetlona lista wszystkich nazw własnych w tekście, w kolejności pojawiania się w powieści. Co się tu więc dzieje? W poprzednim rozdziale wywołaliśmy funkcję `bs.tagName` do wyświetlenia pierwszego wystąpienia tego znacznika na stronie. Teraz wywołujemy funkcję `bs.find_all(tagName, tagAttributes)` po to, aby pobrać nie wyłącznie pierwszy znacznik, lecz listę wszystkich znaczników na stronie.

Po uzyskaniu listy imion program przegląda ją i realizuje funkcję `name.get_text()` w celu oddzielenia treści od znaczników.

### Kiedy należy stosować funkcję `get_text()`, a kiedy zachowywać znaczniki?

Funkcja `.get_text()` usuwa wszystkie znaczniki z danego dokumentu i zwraca ciąg znaków w systemie Unicode zawierający sam tekst. Przykładowo jeżeli pracujesz nad dużym blokiem tekstu zawierającym mnóstwo odnośników, akapitów i innych znaczników, znaczniki te zostaną usunięte i pozostanie sam tekst.

Nie zapominaj, że o wiele łatwiej znajdować poszukiwane elementy w obiekcie `BeautifulSoup` aniżeli w samym bloku tekstowym. Wywołanie funkcji `.get_text()` powinno zawsze stanowić ostateczność, tuż przed wydrukowaniem, zapisaniem lub zmodyfikowaniem danych końcowych. Ogółem warto jak najdłużej zachowywać strukturę znaczników dokumentu.

## Funkcje find() i find_all()

Prawdopodobnie będziesz najczęściej korzystać z funkcji `find()` i `find_all()` biblioteki *BeautifulSoup*. Za ich pomocą możesz z łatwością filtrować strony HTML w celu wyszukiwania list pożądanych znaczników lub pojedynczego znacznika, na podstawie różnych atrybutów tych znaczników.

Obydwie funkcje są do siebie niezwykle podobne, czego dowodem są ich definicje umieszczone w dokumentacji biblioteki *BeautifulSoup*:

```
find_all(tag, attributes, recursive, text, limit, keywords)
find(tag, attributes, recursive, text, keywords)
```

Najprawdopodobniej przez 95% czasu będziesz korzystać wyłącznie z dwóch pierwszych argumentów: `tag` i `attributes`. Przyjrzyjmy się jednak dokładniej wszystkim argumentom.

Z argumentem `tag` mieliśmy już wcześniej do czynienia; możemy w ten sposób przekazywać nazwę znacznika, a nawet listę nazw znaczników. Na przykład poniższy wiersz zwraca listę wszystkich znaczników nagłówków w dokumencie[1]:

```
.find_all(['h1','h2','h3','h4','h5','h6'])
```

Argument `attributes` przyjmuje słownik atrybutów i sprawdza, czy występują znaczniki zawierające jeden z tych atrybutów. Na przykład poniższa funkcja zwraca znaczniki `span` *zarówno* dla koloru czerwonego, jak i zielonego występujące w dokumencie HTML:

```
.find_all('span', {'class':{'green', 'red'}})
```

Argument `recursive` przyjmuje wartości logiczne. Jak głęboko chcesz zapuścić się wewnątrz dokumentu? Jeżeli wybierzesz wartość `True`, funkcja `find_all` będzie sprawdzać wszystkie elementy potomne w poszukiwaniu znaczników zawierających interesujące nas parametry. W przypadku wartości `False` sprawdzane będą tylko główne znaczniki dokumentu. Domyślnie funkcja `find_all` działa rekurencyjnie (wartość `True`); zazwyczaj warto pozostawić to ustawienie bez zmian, chyba że wiesz dokładnie, co robisz, a masz problem z wydajnością.

Argument `text` jest o tyle niezwykły, że przeprowadza dopasowanie na podstawie zawartości tekstowej znaczników, a nie ich własności. Jeżeli na przykład chcesz się dowiedzieć, ile razy wyrażenie *the prince* jest otoczone znacznikami na przykładowej stronie, moglibyśmy zastąpić funkcję `find_all()` z poprzedniego przykładu następującymi wierszami:

```
nameList = bs.find_all(text='the prince')
print(len(nameList))
```

W wyniku otrzymujemy cyfrę 7.

[1] Jeżeli chcesz uzyskać listę wszystkich znaczników `h<jakiś_poziom>` w dokumencie, istnieją bardziej zwięzłe metody pisania kodu, dające identyczne rezultaty. Przyjrzymy się im w podrozdziale „Wyrażenia regularne w bibliotece BeautifulSoup".

Argument `limit` jest oczywiście stosowany wyłącznie w metodzie `find_all()`; funkcja `find` jest równoważna wywołaniu funkcji `find_all` z ograniczeniem do jednego wyniku. Możemy wyznaczyć wartość tego argumentu, jeżeli interesuje nas tylko wyświetlenie pierwszych *x* elementów na danej stronie. Pamiętaj jednak, że elementy te są wyświetlane w kolejności występowania na stronie i niekoniecznie muszą to być te, których poszukujemy.

Argument `keyword` pozwala wybierać znaczniki zawierające określony atrybut lub zbiór atrybutów. Na przykład:

```
title = bs.find_all(id='title', class_='text')
```

Zostaje zwrócony pierwszy znacznik zawierający wyraz *text* w atrybucie `class_` i wyraz *title* w atrybucie `id`. Zwróć uwagę, że zgodnie z konwencją każda wartość atrybutu `id` powinna zostać użyta tylko raz na stronę. Zatem w praktyce powyższy wiersz może nie być zbyt przydatny i zasadniczo jest równoznaczny poniższemu wierszowi:

```
title = bs.find(id='title')
```

## Argument keyword i słowo kluczowe class

Argument `keyword` może przydać się w pewnych sytuacjach, okazuje się jednak funkcją nadmiarową w bibliotece *BeautifulSoup*. Pamiętaj, że wszystkie rezultaty uzyskiwane za pomocą argumentu `keyword` możemy również osiągnąć za pomocą technik opisanych w dalszej części rozdziału (w podrozdziałach „Wyrażenia regularne" i „Wyrażenia lambda").

Na przykład obydwa poniższe wiersze są sobie tożsame:

```
bs.find_all(id='text')
bs.find_all('', {'id':'text'})
```

Ponadto możemy niekiedy napotkać problem, korzystając z argumentu `keyword`, zwłaszcza podczas wyszukiwania elementów na podstawie ich atrybutu `class`, ponieważ jest to jedno z chronionych słów kluczowych środowiska Python. Oznacza to, że jest to zarezerwowany wyraz, którego nie możemy używać jako nazwy zmiennej lub argumentu (nie ma to związku z omówionym wcześniej argumentem `keyword` funkcji `BeautifulSoup.find_all()`)[2]. Jeśli na przykład spróbujesz zrealizować poniższe wywołanie, zostanie wyświetlony błąd składni z powodu niestandardowego użycia słowa kluczowego `class`:

```
bs.find_all(class='green')
```

Zamiast tego możemy skorzystać z dość niezgrabnego rozwiązania stosowanego w bibliotece *BeautifulSoup* i dodać podkreślnik:

```
bs.find_all(class_='green')
```

Ewentualnie możemy zamknąć słowo `class` w cudzysłowie pojedynczym:

```
bs.find_all('', {'class':'green'})
```

[2] Materiały referencyjne języka Python (*Python Language Reference*) zawierają pełną listę wszystkich chronionych słów kluczowych (*https://docs.python.org/3/reference/lexical_analysis.html#keywords*).

Pewnie myślisz sobie teraz: „Chwileczkę! Przecież potrafię już znaleźć znacznik zawierający listę atrybutów poprzez przekazanie atrybutów do funkcji w liście słownika!”.

Pamiętaj, że przekazanie listy znaczników do funkcji `.find_all()` za pomocą listy atrybutów działa niczym filtr logiczny „lub” (wybiera on listę wszystkich znaczników mających `tag1`, `tag2` lub `tag3`…). Jeżeli masz dużą listę znaczników, program może zwrócić mnóstwo niechcianych elementów. Argument `keyword` pozwala wprowadzić dodatkowy filtr „i”.

## Inne obiekty biblioteki BeautifulSoup

Do tej pory poznaliśmy dwa typy obiektów w bibliotece *BeautifulSoup*:

Obiekty `BeautifulSoup`
: Wystąpienia pojawiające się we wcześniejszych listingach, np. zmienna `bs`.

Obiekty `Tag`
: Otrzymywane w postaci list lub pojedynczo poprzez wywołanie funkcji `find` i `find_all` wobec obiektu `BeautifulSoup` lub schodząc zgodnie z hierarchią, np.:

```
bs.div.h1
```

Istnieją jednak jeszcze dwa obiekty, które choć są mniej popularne, również należy znać:

Obiekty `NavigableString`
: Służą do reprezentowania tekstu wewnątrz znaczników, a nie samych znaczników (niektóre funkcje działają na obiektach `NavigableString` i tworzą je).

Obiekt `Comment`
: Używany do wyszukiwania komentarzy HTML zawartych w znacznikach `<!--takich jak ten-->`.

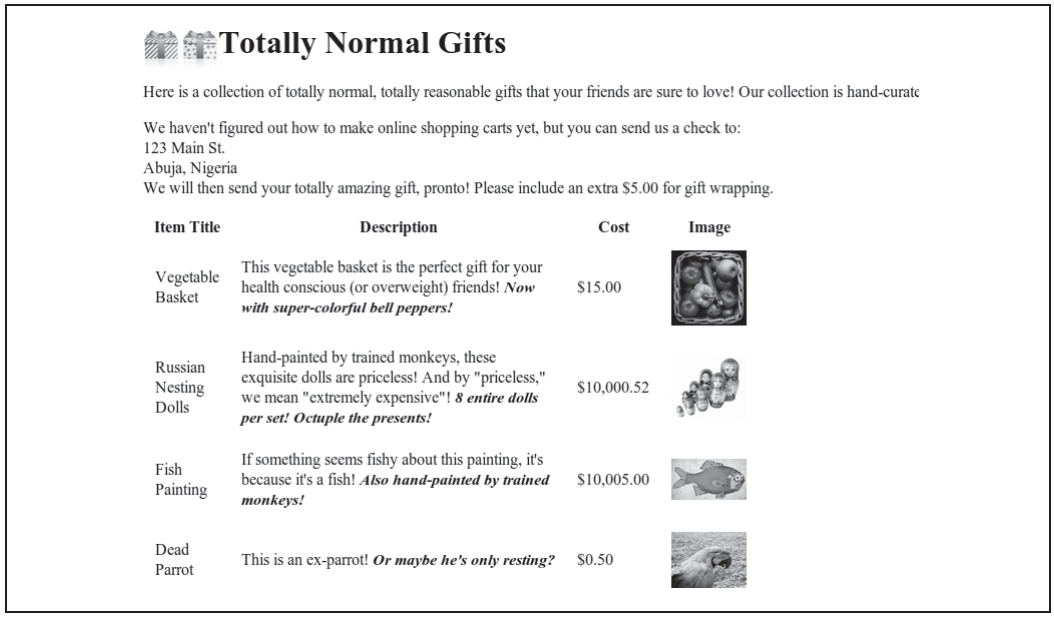
Są to jedyne cztery obiekty (przynajmniej w czasie pisania niniejszej książki), które napotkasz w bibliotece *BeautifulSoup*.

## Poruszanie się po drzewach hierarchii

Funkcja `find_all` odpowiedzialna jest za wyszukiwanie znaczników na podstawie ich nazw i zawartych w nich atrybutów. Jak jednak wyszukać znacznik bazując na jego położeniu w dokumencie? Do akcji wkracza tutaj nawigacja po drzewie hierarchii. W rozdziale 1. widzieliśmy sposób poruszania się po drzewie hierarchii obiektu `BeautifulSoup` w jednym kierunku:

```
bs.znacznik.znacznikPodrzędny.jeszczeBardziejPodrzędnyZnacznik
```

Nauczymy się teraz poruszać do góry, w poprzek oraz na skos po drzewach hierarchii HTML. Skorzystamy z naszego wysoce podejrzanego sklepu internetowego umieszczonego na stronie *http://www.pythonscraping.com/pages/page3.html* (rysunek 2.1) i zobaczymy, co nam się uda z niego wydobyć.



## Totally Normal Gifts

Here is a collection of totally normal, totally reasonable gifts that your friends are sure to love! Our collection is hand-curate

We haven't figured out how to make online shopping carts yet, but you can send us a check to:
123 Main St.
Abuja, Nigeria
We will then send your totally amazing gift, pronto! Please include an extra $5.00 for gift wrapping.

| Item Title | Description | Cost | Image |
|---|---|---|---|
| Vegetable Basket | This vegetable basket is the perfect gift for your health conscious (or overweight) friends! ***Now with super-colorful bell peppers!*** | $15.00 | |
| Russian Nesting Dolls | Hand-painted by trained monkeys, these exquisite dolls are priceless! And by "priceless," we mean "extremely expensive"! ***8 entire dolls per set! Octuple the presents!*** | $10,000.52 | |
| Fish Painting | If something seems fishy about this painting, it's because it's a fish! ***Also hand-painted by trained monkeys!*** | $10,005.00 | |
| Dead Parrot | This is an ex-parrot! ***Or maybe he's only resting?*** | $0.50 | |

*Rysunek 2.1. Zrzut ekranu strony dostępnej pod adresem* http://www.pythonscraping.com/pages/page3.html

Struktura HTML tej strony w postaci drzewa hierarchii (dla zachowania przejrzystości pominęłam niektóre znaczniki) przedstawia się następująco:

- HTML
  - `body`
    - `div.wrapper`
      - `h1`
      - `div.content`
      - `table#giftList`
        - `tr`
          - `th`
          - `th`
          - `th`
          - `th`
        - `tr.gift#gift1`
          - `td`
          - `td`
            - `span.excitingNote`
          - `td`
          - `td`
            - `img`
        - …kontynuacja wierszy tabeli…
      - `div.footer`

Będziemy korzystać z tej przykładowej struktury również w dalszej części rozdziału.

### Dzieci i pozostali potomkowie

W informatyce i niektórych działach matematyki czasami słyszymy o potwornościach, jakim są poddawane dzieci: ich przenoszeniu, przechowywaniu, usuwaniu, a nawet zabijaniu. Na szczęście tutaj będziemy zajmować się wyłącznie ich wybieraniem!

W bibliotece *BeautifulSoup* (i w wielu innych) istnieje różnica między **dziećmi** (ang. *children*) a **potomkami** (ang. *descendants*): podobnie jak w drzewie genealogicznym człowieka dzieci zawsze znajdują się dokładnie jeden poziom niżej od rodziców, natomiast potomkowie występują dowolnie niżej. Na przykład znaczniki `tr` są dziećmi znacznika `table`, a znaczniki `tr`, `th`, `td`, `img` i `span` są jego potomkami (przynajmniej w przypadku naszej przykładowej strony). Wszystkie dzieci są potomkami, ale nie wszyscy potomkowie są dziećmi.

Generalnie funkcje biblioteki *BeautifulSoup* zawsze przetwarzają potomków bieżącego wybranego znacznika. Przykładowo obiekt `bs.body.h1` wybiera pierwszy znacznik `h1` będący potomkiem znacznika `body`. Nie będzie poszukiwał znaczników występujących poza ciałem dokumentu.

Na drodze analogii funkcja `bs.div.find_all('img')` znajdzie pierwszy znacznik `div` w dokumencie, a następnie pobierze listę wszystkich znaczników `img` będących potomkami znacznika `div`.

Jeżeli interesują Cię wyłącznie potomkowie będący dziećmi, możesz użyć znacznika `.children`:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for child in bs.find('table',{'id':'giftList'}).children:
    print(child)
```

W wyniku powyższego listingu zostaje wyświetlona lista rzędów zawierających produkty w tabeli `giftList`, włącznie z pierwszym rzędem etykiet kolumn. Gdybyśmy zastąpili funkcję `children()` funkcją `descendants()`, zostałoby znalezionych i wyświetlonych ponad 20 znaczników, w tym znaczniki `img`, `span` i poszczególne znaczniki `td`. Zdecydowanie należy odróżniać dzieci od potomków!

### Rodzeństwo

Dzięki funkcji `next_siblings()` gromadzenie danych z tabel, zwłaszcza zawierających rzędy z nazwami kolumn, staje się banalne:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for sibling in bs.find('table', {'id':'giftList'}).tr.next_siblings:
    print(sibling)
```

Rezultatem powyższego kodu jest wyświetlenie wszystkich rzędów z produktami z tabeli `giftList` oprócz pierwszego rzędu (z nazwami kolumn). Dlaczego został pominięty ten wiersz? Obiekty nie mogą być **rodzeństwem** (ang. *siblings*) same dla siebie. Za każdym razem, gdy pobieramy rodzeństwo danego obiektu, sam ten obiekt nie może zostać umieszczony na tej liście. Jak sama nazwa funkcji sugeruje, wywoływane jest jedynie **następne** (ang. *next*) rodzeństwo. Gdybyśmy na przykład wybrali jakiś rząd ze środka listy i wywołali wobec niego funkcję `next_siblings`, zostałoby wyświetlone tylko rodzeństwo występujące po nim. Zatem wybierając rząd z nazwami kolumn i wywołując funkcję `next_siblings`, jesteśmy w stanie wybierać wszystkie rzędy w tabeli oprócz tego wskazanego.

### Precyzuj wyznaczane elementy

Powyższy kod będzie działał równie skutecznie, jeżeli do zaznaczenia pierwszego rzędu tabeli użyjesz wyrażenia `bs.table.tr`, a nawet `bs.tr`. Ja jednak zawsze fatyguję się z zapisywaniem tego typu instrukcji w rozwleklejszej postaci:

```
bs.find('table',{'id':'giftList'}).tr
```

Nawet jeśli wydaje nam się, że na stronie znajduje się tylko jedna tabela (lub inny znacznik docelowy), łatwo przeoczyć różne rzeczy. Poza tym szata graficzna strony często ulega zmianom. Znacznik, który kiedyś występował jako jedyny na stronie, może zostać jej drugim albo trzecim elementem tego typu. Aby zwiększyć wszechstronność robotów indeksujących, najlepiej zawsze w jak największym stopniu precyzować dobór znaczników. Korzystaj z atrybutów znaczników, jeśli tylko masz taką możliwość.

W ramach uzupełnienia funkcji `next_siblings` możemy również korzystać z funkcji `previous_siblings`, która często przydaje się w sytuacji występowania dobieranego znacznika na końcu listy znaczników, które chcesz pozyskać.

Dostępne są też analogiczne funkcje `next_sibling` i `previous_sibling`, których jedyną różnicą w stosunku do funkcji `next_siblings` i `previous_siblings` jest zwracanie pojedynczych znaczników, a nie całych ich list.

## Rodzice

Podczas pozyskiwania danych ze stron zauważysz, że znacznie rzadziej szuka się znaczników nadrzędnych niż dzieci czy rodzeństwa. Zazwyczaj podczas sprawdzania stron WWW pod względem ekstrakcji danych zaczynamy od górnej warstwy znaczników i poszukujemy sposobu zejścia aż do poziomu poszukiwanej informacji. Czasami jednak zdarzają się dziwne sytuacje wymagające skorzystania z funkcji wyszukiwania rodziców (ang. *parents*): `.parent` i `.parents`. Na przykład:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
print(bs.find('img',
              {'src':'../img/gifts/img1.jpg'})
    .parent.previous_sibling.get_text())
```

W wyniku tego kodu zostanie wyświetlona lista obiektu reprezentowanego przez obraz umieszczony w lokalizacji *../img/gifts/img1.jpg* (w tym przypadku cena wynosi 15 dolarów).

Jak wygląda użyty tu mechanizm działania? Poniżej zaprezentowałam hierarchię znaczników w interesującym nas fragmencie kodu HTML i zaznaczyłam na nim punktami poszczególne etapy:

- `<tr>`
  - `td`
  - `td`
  - `td` ❸
    - `"$15.00"` ❹
  - `td` ❷
    - `<img src="../img/gifts/img1.jpg">` ❶

❶ Najpierw zostaje wybrany znacznik obrazu, gdzie `src="../img/gifts/img1.jpg"`.

❷ Wybieramy rodzica tego znacznika (w tym przypadku jest to znacznik `td`).

❸ Wybieramy poprzednie rodzeństwo znacznika `td` (w tym przypadku jest to znacznik `td` zawierający cenę produktu).

❹ Wybieramy tekst znajdujący się wewnątrz tego znacznika (`"$15.00"`).

# Wyrażenia regularne

Oto stary dowcip informatyczny: Powiedzmy, że masz problem i postanawiasz go rozwiązać za pomocą wyrażeń regularnych. No to teraz masz dwa problemy.

Niestety **wyrażenia regularne** (ang. *regular expressions*; znane także pod skróconą nazwą **regex**) często prezentuje się studentom w postaci olbrzymich tabel zawierających losowe znaki, które są łączone ze sobą w pozornie bezsensowny sposób. Jest to doskonały sposób na zniechęcenie ludzi, którzy w trakcie swojej kariery nieraz tworzą niepotrzebnie skomplikowane funkcje wyszukiwania i filtrowania, podczas gdy najczęściej wystarczyłby jeden wiersz dobrze skonstruowanego wyrażenia regularnego!

Na szczęście dla nas koncepcja wyrażeń regularnych wcale nie jest tak skomplikowana i można łatwo się nauczyć korzystać z nich, jeśli przeanalizujemy kilka prostych przykładów i z nimi poeksperymentujemy.

Nazwa „wyrażenia regularne” bierze się stąd, że służą one do rozpoznawania regularnych ciągów znaków; informują nas one, że „tak, podany ciąg znaków jest zgodny z określonymi regułami i zwrócę go” albo „ten ciąg znaków nie jest zgodny z regułami i odrzucę go”. Rozwiązanie to jest wyjątkowo przydatne do szybkiego przeszukiwania dokumentów pod kątem ciągów znaków przypominających numery telefoniczne czy adresy e-mail.

Zwróć uwagę, że użyłam określenia **regularny ciąg znaków** (ang. *regular string*). Co to jest? Jest to dowolny ciąg znaków, który można wygenerować za pomocą szeregu reguł liniowych[3], na przykład:

1. Zapisz znak *a* przynajmniej jeden raz.
2. Dodaj znak *b* dokładnie pięć razy.
3. Dodaj znak *c* dowolną parzystą liczbę razy.
4. Zapisz na końcu znak *d* lub *e*.

Zgodne z tą regułą ciągi znaków wyglądają następująco: *aaaabbbbbccccd*, *aabbbbbcce* itd. (istnieje nieskończona liczba możliwości).

Wyrażenia regularne stanowią skróconą wersję zapisu tych reguł. Oto na przykład wyrażenie regularne symbolizujące dopiero co opisany zbiór reguł:

```
aa*bbbbb(cc)*(d|e)
```

Taki zapis może początkowo wydawać się nieco przytłaczający, ale po rozbiciu go na elementy składowe stanie się bardziej zrozumiały:

***aa****
: Zostaje zapisany znak *a*, po nim natomiast wyrażenie *a** (czyt. *a gwiazdka*), co oznacza „dowolna liczba znaków *a*, w tym również 0". W ten sposób mamy pewność, że znak *a* będzie występował przynajmniej raz.

***bbbbb***
: Tutaj nie ma niespodzianek — znak *b* ma po prostu występować pięć razy pod rząd.

***(cc)****
: Każda liczba parzysta elementów może zostać pogrupowana w pary, zatem w celu wymuszenia reguły parzystości możesz umieścić dwa znaki *c* w nawiasie, a poza nawiasem wstawić gwiazdkę, co oznacza wyszukiwanie dowolnej liczby **par** znaku *c* (równie dobrze żadna para może nie występować w wyrażeniu).

***(d|e)***
: Kreska pionowa wstawiona pomiędzy znakami oznacza „ten znak *lub* tamten znak". W tym przypadku rozumiemy powyższy zapis jako „dodaj znak *d* lub znak *e*". W ten sposób mamy pewność, że będzie występował dokładnie jeden z tych znaków.

[3] Być może zastanawiasz się teraz, czy istnieją wyrażenia „nieregularne". Ich opis wykracza poza zakres niniejszej książki, ogółem jednak opisywane są w ten sposób ciągi znaków zgodne z regułami typu „zapisz znaki *a* przez liczbę pierwszą powtórzeń, a następnie dokładnie dwukrotnie więcej znaków *b*" lub „zapisz palindrom". Nie jest możliwe rozpoznawanie tego typu konstrukcji za pomocą wyrażeń regularnych. Na szczęście nigdy nie znalazłam się w sytuacji, w której mój pełzacz musiałby wykrywać takie struktury.

### Eksperymentowanie z wyrażeniami regularnymi

Podczas nauki zapisywania wyrażeń regularnych najważniejsze jest ich testowanie, dzięki czemu łatwiej zrozumieć mechanizm ich działania. Jeżeli nie chce Ci się uruchamiać edytora kodu, zapisywać wyrażenia regularnego i realizować stworzonego programu w celu sprawdzenia, czy dane wyrażenie działa zgodnie z oczekiwaniami, możesz odwiedzić taką stronę jak np. Regex Pal (*https://www.regexpal.com/*) i na bieżąco testować wyrażenia regularne.

Tabela 2.1 zawiera spis popularnych symboli wyrażeń regularnych wraz z krótkimi opisami i przykładami. Lista ta jest niepełna i, jak już wspomniałam, mogą występować drobne różnice pomiędzy poszczególnymi językami programowania. Jednak 12 poniższych symboli jest najczęściej spotykanych w środowisku Python i możemy używać ich do wyszukiwania i gromadzenia niemal każdego typu ciągów znaków.

*Tabela 2.1. Popularne symbole wyrażeń regularnych*

| Symbol(e) | Znaczenie | Przykład | Przykładowe ciągi znaków |
|---|---|---|---|
| `*` | Dopasowuje poprzedzający znak, podwyrażenie lub znaki umieszczone w nawiasach 0 lub więcej razy. | `a*b*` | *aaaaaaaa, aaabbbbb, bbbbbb* |
| `+` | Dopasowuje poprzedzający znak, podwyrażenie lub znaki umieszczone w nawiasach 1 lub więcej razy. | `a+b+` | *aaaaaaaab, aaabbbbb, abbbbbb* |
| `[]` | Dopasowuje dowolny znak znajdujący się wewnątrz nawiasu (tj. „dobierz dowolny z tych elementów"). | `[A-Z]*` | *PASSA, WIELKIE, QWERTY* |
| `()` | Podwyrażenie grupowane (jest ono przetwarzane jako pierwsze w „kolejności operacji" wyrażeń regularnych). | `(a*b)*` | *aaabaab, abaaab, ababaaaaab* |
| `{m, n}` | Dopasowuje poprzedzający znak, podwyrażenie lub znaki umieszczone w nawiasach w zakresie od m do n razy (włącznie). | `a{2,3}b{2,3}` | *aabbb, aaabbb, aabb* |
| `[^]` | Dopasowuje każdy znak nieumieszczony wewnątrz nawiasu. | `[^A-Z]*` | *passa, małe, qwerty* |
| `\|` | Dopasowuje każdy znak, podwyrażenie lub ciąg znaków rozdzielone za pomocą symbolu \| (jest to kreska pionowa, a nie duża litera i). | `k(a\|i\|o)t` | *kat, kit, kot* |
| `.` | Dopasowuje dowolny pojedynczy znak (w tym symbole, cyfry, odstępy itd.). | `k.t` | *kat, kzt, k$t, k t* |
| `^` | Wskazuje, że dany znak lub podwyrażenie znajduje się na początku ciągu znaków. | `^a` | *aktor, asdf, a* |
| `\` | Symbol ucieczki (znaki specjalne mogą być używane w znaczeniu dosłownym). | `\. \| \\` | *.\|* |
| `$` | Symbol często stawiany na końcu wyrażenia regularnego. Oznacza on „dopasuj to do końca ciągu znaków". Bez tego symbolu każde wyrażenie regularne kończy się zasadniczo członem `.*`, co sprawia, że akceptowane są jedynie ciągi znaków, których pierwsza część jest dopasowana. Symbol ten można uznać za analog znaku `^`. | `[A-Z]*` ↪`[a-z]*$` | *ABCabc, zzzyx, Robert* |
| `?!` | „Nie zawiera". Takie dziwne połączenie symboli, umieszczane przed znakiem (lub wyrażeniem regularnym) oznacza, że znak ten nie powinien występować w danym obszarze dłuższego ciągu znaków. Stosowanie tego rozwiązania nie jest takie proste — dany znak może przecież występować w innym rejonie ciągu znaków. Jeżeli chcesz zupełnie wyeliminować dany znak, skorzystaj z kombinacji tych symboli ze znakami `^` i $ na obydwu końcach. | `^((?![A-` ↪`Z]).)*$` | *same-małe-litery, $ymb0le m0gą 8yć* |

Klasycznym przykładem stosowania wyrażeń regularnych jest rozpoznawanie adresów e-mail. Dokładne reguły zarządzania adresami e-mail różnią się nieznacznie pomiędzy poszczególnymi serwerami pocztowymi, jesteśmy jednak w stanie określić kilka ogólnych zasad. W drugiej kolumnie prezentuję wyrażenie regularne dla każdej z tych reguł:

| | |
|---|---|
| *Reguła 1.*<br>Pierwsza część adresu e-mail zawiera co najmniej jedno z następujących: duże litery, małe litery, cyfry 0-9, kropki (.), znaki plus (+) lub podkreślniki (_). | *[A-Za-z0-9\._+]+*<br>To wyrażenie regularne jest dość sprytnie zaplanowane. Na przykład „wie" ono, że fragment *A-Z* oznacza „dowolna duża litera od *A* do *Z*". Umieszczając wszystkie możliwe sekwencje w nawiasach kwadratowych (a nie okrągłych), mówimy „symbol ten może być dowolny z umieszczonych w nawiasie". Zwróć również uwagę, że znak + oznacza „znaki te mogą występować dowolną liczbę razy, ale muszą się pojawić przynajmniej raz". |
| *Reguła 2.*<br>Następnym elementem adresu e-mail jest znak @. | *@*<br>Jest to całkiem zrozumiałe: symbol @ musi znajdować się w środku i występować tylko raz. |
| *Reguła 3.*<br>Adres e-mail musi następnie zawierać przynajmniej jedną dużą lub małą literę. | *[A-Za-z]+*<br>W pierwszej części nazwy domenowej (po znaku @) mogą być używane wyłącznie znaki literowe. Do tego musi występować przynajmniej jeden znak. |
| *Reguła 4.*<br>Teraz pojawia się kropka. | *\.*<br>W środku nazwy domenowej musi występować kropka. Ukośnik pełni tu funkcję znaku ucieczki. |
| *Reguła 5.*<br>Adres e-mail musi kończyć się nazwą *com*, *org*, *edu* lub *net* (w rzeczywistości istnieje wiele głównych nazw domenowych, ale cztery wspomniane wystarczą na potrzeby naszego przykładu). | *(com\|org\|edu\|net)*<br>Umieszczamy tu listę możliwych sekwencji liter występujących na końcu adresu e-mail. |

Łącząc wszystkie te reguły, otrzymujemy następujące wyrażenie regularne:

```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

Podczas próby stworzenia wyrażenia regularnego od podstaw najlepiej najpierw rozpisać sobie listę reguł określających strukturę docelowego ciągu znaków. Zwracaj szczególną uwagę na elementy krańcowe. Na przykład czy podczas rozpoznawania numerów telefonów bierzesz pod uwagę kody krajów i numery wewnętrzne?

### Wyrażenia regularne nie zawsze regularne!

Standardowa wersja wyrażeń regularnych (opisana w tej książce i obsługiwana przez środowisko Python i bibliotekę *BeautifulSoup*) bazuje na składni stosowanej w języku Perl. Jest ona wykorzystywana w większości współczesnych języków programowania. Pamiętaj jednak, że wyrażenia regularne w innych językach mogą wyglądać nieco odmiennie. Nawet takie języki jak Java mogą inaczej obsługiwać niektóre wyrażenia regularne. W razie jakichkolwiek wątpliwości nie bój się korzystać z dokumentacji!

# Wyrażenia regularne w bibliotece BeautifulSoup

Jeżeli uważasz, że poprzedni podrozdział poświęcony wyrażeniom regularnym nie był powiązany z treścią niniejszej książki, to teraz przekonasz się, że jest inaczej. Biblioteka *BeautifulSoup* i wyrażenia regularne idą ze sobą w parze, gdy chodzi o ekstrakcję danych. W rzeczywistości większość funkcji przyjmujących argumenty w postaci ciągów znaków (np. `find(id="identyfikatorZnacznika")`) „rozumieją" również wyrażenia regularne.

Przyjrzyjmy się niektórym przykładom i zobaczmy, co uda nam się wydobyć ze strony *http://www.pythonscraping.com/pages/page3.html*.

Zwróć uwagę, że strona ta zawiera kilka zdjęć produktów w następującej postaci:

```
<img src="../img/gifts/img3.jpg">
```

Gdybyś chciał uzyskać adresy URL do wszystkich obrazów, pozornie wydawałoby się to bardzo proste: wystarczyłoby znaleźć wszystkie znaczniki obrazów za pomocą funkcji `.find_all("img")`, prawda? Pojawia się tu jednak pewien problem. Współczesne witryny WWW, oprócz oczywistych „dodatkowych" obrazów (takich jak logo), zawierają również często „ukryte" obrazy, puste elementy graficzne ułatwiające rozmieszczanie elementów strony, a także inne losowe znaczniki obrazów, o których możemy nie mieć pojęcia. Z pewnością nie możemy być przekonani o tym, że jedynymi obrazami na stronie są zdjęcia produktów.

Załóżmy także, że szata graficzna strony może ulec zmianie lub z jakiegoś innego powodu nie chcesz polegać na *pozycji* obrazu na stronie w celu znalezienia prawidłowego znacznika. Może tak być w sytuacji, gdy próbujesz wydobyć określone elementy lub dane rozrzucone losowo po całej stronie. Na przykład oferta specjalna może być umieszczona w osobnym układzie graficznym występującym tylko na niektórych stronach sklepu.

Rozwiązaniem okazuje się poszukiwanie jakiejś cechy pozwalającej rozpoznać dany znacznik. W tym przypadku możemy szukać ścieżki do pliku obrazu:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
images = bs.find_all('img',
    {'src':re.compile('\.\.\/img\/gifts/img.*\.jpg')})
for image in images:
    print(image['src'])
```

Zostaną wyświetlone jedynie względne ścieżki do obrazów rozpoczynające się od wyrażenia *../img/gifts/img* i zakończone rozszerzeniem *.jpg*:

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

Wyrażenie regularne może zostać wstawione jako dowolny argument w bibliotece *BeautifulSoup*, dzięki czemu zyskujemy olbrzymią swobodę w wyszukiwaniu elementów docelowych.

# Uzyskiwanie dostępu do atrybutów

Do tej pory dowiedzieliśmy się, w jaki sposób uzyskiwać dostęp do znaczników i jak je filtrować w celu uzyskania dostępu do umieszczonej w nich treści. Często jednak w procesie ekstrakcji danych poszukujemy nie tyle zawartości znacznika, co jego atrybutów. Jest to szczególnie przydatne w przypadku takich znaczników jak `a`, w którym adres URL jest przechowywany za pomocą atrybutu `href`; innym przykładem jest znacznik `img`, gdzie docelowy obraz stanowi wartość atrybutu `src`.

W przypadku obiektów `tag` możemy uzyskać automatycznie listę atrybutów za pomocą wywołania:

```
myTag.attrs
```

Pamiętaj, że zostaje dosłownie zwrócony obiekt słownika Python, dzięki czemu odczytywanie i modyfikowanie tych atrybutów staje się banalne. Na przykład lokalizacja źródłowa obrazu może zostać wyszukana w następujący sposób:

```
myImgTag.attrs['src']
```

# Wyrażenia lambda

Jeżeli masz wykształcenie informatyczne, to prawdopodobnie kiedyś uczyłeś się o wyrażeniach lambda, po czym więcej już do nich nie wróciłeś. Jeśli nie, to być może nawet o nich nie słyszałeś (lub kojarzysz je jako „te bzdurki, które miałem kiedyś wykuwać"). Nie będziemy w tym podrozdziale zbytnio wnikać w ten typ funkcji, ale udowodnię, że mogą być one przydatne w procesie ekstrakcji danych.

W istocie **wyrażenie lambda** (ang. *lambda expression*) stanowi funkcję, która jest przekazywana do innej funkcji jako zmienna; zamiast definiować funkcję jako *f(x, y)*, możemy ją zdefiniować jako *f(g(x), y)*, a nawet *f(g(x), h(x))*.

Biblioteka *BeautifulSoup* umożliwia przekazywanie pewnych typów funkcji jako parametrów funkcji `find_all`.

Jedynym ograniczeniem jest konieczność przyjmowania przez te funkcje obiektu `tag` jako argumentu, przez co zwracają wartości logiczne. Każdy obiekt `tag` napotykany przez bibliotekę *BeautifulSoup* jest oceniany wewnątrz tej funkcji — zwracane są znaczniki dające w wyniku wartość `True`, pozostałe zaś zostają odrzucone.

Na przykład ten wiersz zwraca wszystkie znaczniki zawierające dokładnie dwa atrybuty:

```
bs.find_all(lambda tag: len(tag.attrs) == 2)
```

Tutaj funkcją przekazywaną w postaci argumentu jest `len(tag.attrs) == 2`. Jeżeli dany znacznik zwraca wartość `True`, zostanie on zwrócony przez funkcję `find_all`. Oznacza to, że zostaną wyszukane wszystkie znaczniki zawierające dwa atrybuty, na przykład:

```
<div class="body" id="content"></div>
<span style="color:red" class="title"></span>
```

Funkcje lambda okazują się tak użyteczne, że możemy za ich pomocą zastępować nawet istniejące funkcje biblioteki *BeautifulSoup*:

```
bs.find_all(lambda tag: tag.get_text() ==
    'Or maybe he\'s only resting?')
```

To samo możemy osiągnąć bez pomocy wyrażenia lambda:

```
bs.find_all('', text='Or maybe he\'s only resting?')
```

Jeżeli jednak zapamiętasz składnię funkcji lambda i sposób uzyskiwania dostępu do własności znacznika, nie będzie Ci potrzebna znajomość żadnej innej składni biblioteki *BeautifulSoup*!

Funkcja lambda może być dowolną funkcją zwracającą wartości `True` lub `False`, dlatego możesz ją łączyć nawet z wyrażeniami regularnymi w celu wyszukiwania znaczników, których atrybuty są zgodne z określonym wzorcem ciągu znaków.

# Skorowidz

## A



## B

## C



## D



## E



## F



## G



## I

## J



## K



## L



## M



## N



## O



## P

## R



## S



## S



## T



## U



## V



## W

## Z

**Notatki**

O'REILLY®

# Wyszukuj dane, gromadź je i korzystaj z nich do woli!

Ekstrakcję danych (ang. *web scraping*), zwaną też wydobywaniem danych z zasobów internetu, wiele osób postrzega jako wyższy stopień wtajemniczenia, nie dziwi więc, że wokół tej dziedziny narosło mnóstwo mitów. Wątpliwości jest wiele, począwszy od legalności tego rodzaju praktyk, skończywszy na właściwościach różnych narzędzi. W praktyce na ekstrakcję danych składa się cały szereg zróżnicowanych technik i technologii, takich jak analiza danych, analiza składniowa języka naturalnego, a także zabezpieczenie informacji. Aby w pełni wykorzystać ich zalety, konieczne jest zrozumienie sposobu, w jaki funkcjonują.

W tej książce znajdziesz zasady gromadzenia, przekształcania i wykorzystywania danych z różnych zasobów. W kontekście ekstrakcji danych omówiono zagadnienia związane z bazami danych, serwerami sieciowymi, protokołem HTTP, językiem HTML, bezpieczeństwem sieciowym, przetwarzaniem obrazów, analizą danych i wieloma innymi kwestiami. Zaprezentowane tu rozwiązania programistyczne zostały napisane w Pythonie. Nie zabrakło też omówienia bibliotek przydatnych w pracy osób tworzących roboty indeksujące. Poznaj rozwiązania stosowane w prognozowaniu rynkowym, tłumaczeniu maszynowym, a nawet w diagnostyce medycznej!

Najważniejsze zagadnienia:

- korzystanie z platformy Scrapy do tworzenia robotów
- metody odczytu, wydobywania i przechowywania pozyskiwanych danych
- oczyszczanie i normalizacja danych
- interfejsy API
- przetwarzanie obrazów na tekst
- testowanie witryn za pomocą robotów

**Ryan Mitchell** — inżynier oprogramowania i analityk danych. Zajmuje się tworzeniem interfejsów API i narzędzi do analizy danych. Jest również doradczynią w zakresie ekstrakcji danych w branży handlu detalicznego, w sektorze finansowym i farmaceutycznym, ponadto była konsultantką ds. programu nauczania oraz wykładowczynią kontraktową na Uniwersytecie Północno-Zachodnim i w Olin College of Engineering.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl



ISBN 978-83-283-5635-1
9 788328 356351

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł