Rozdział 12

Programy sieciowe

Mimo że wiele przykładów w tej książce koncentruje się na odczytywaniu plików i wyszukiwaniu w nich danych, istnieje wiele innych, zróżnicowanych źródeł informacji, jeśli wziąć pod uwagę internet.

W tym rozdziale będziemy udawać, że jesteśmy przeglądarką internetową, i wyszukiwać strony internetowe za pomocą HTTP (Hypertext Transfer Protocol, czyli protokołu przesyłania dokumentów hipertekstowych). Następnie będziemy odczytywać dane ze stron internetowych oraz je przetwarzać.

12.1. Hypertext Transfer Protocol – HTTP

Protokół sieciowy, który napędza internet, jest w rzeczywistości dość prosty. Python wspiera go poprzez wbudowaną funkcję `socket()`, która ułatwia nawiązywanie połączeń sieciowych i pobieranie danych przez tzw. gniazda (lub z ang. sockety).

Gniazdo jest bardzo podobne do pliku, z tą różnicą, że jedno gniazdo zapewnia dwukierunkowe połączenie między dwoma programami. Przy pomocy tego samego gniazda możesz zarówno odczytywać dane, jak i wysyłać/zapisywać dane. Jeżeli wyślesz jakieś dane do gniazda, to zostaną one wysłane do aplikacji znajdującej się po drugiej stronie tego gniazda. Jeżeli czytasz z gniazda, to otrzymujesz dane, które wysłała do Ciebie aplikacja z drugiej strony.

Ale jeżeli próbujesz odczytać z gniazda dane wtedy, gdy program na drugim końcu jeszcze żadnych nie wysłał, to po prostu czekasz. Jeżeli programy po obu stronach gniazda po prostu czekają na jakieś dane, nie wysyłając niczego, to będą czekać bardzo długo, więc ważną częścią programów komunikujących się przez internet jest posiadanie jakiegoś protokołu.

Protokół jest zestawem precyzyjnych reguł, które określają, kto robi pierwszy krok, jak to zrobić, a następnie jakie są możliwe odpowiedzi, kto wysyła kolejną wiadomość itd. W pewnym sensie te dwie aplikacje na obu końcach gniazda wykonują taniec i pilnują, żeby się nawzajem nie nadepnąć.

Istnieje wiele dokumentów, które opisują protokoły sieciowe. Hypertext Transfer Protocol jest opisany w poniższym dokumencie:

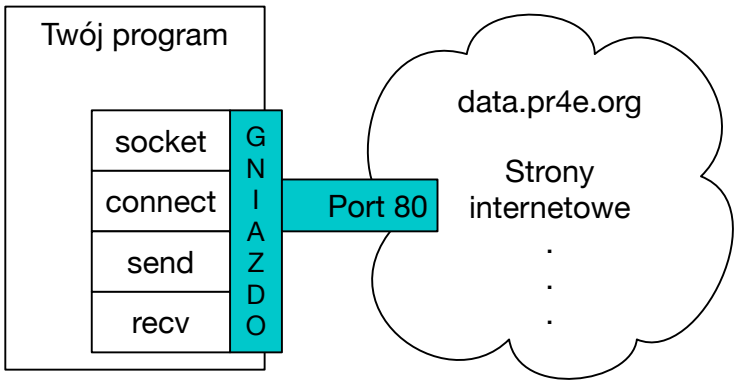
https://www.w3.org/Protocols/rfc2616/rfc2616.txt

Jest to długi i skomplikowany 176-stronicowy dokument z dużą ilością szczegółów. Jeśli uważasz, że jest interesujący, przeczytaj go w całości. Ale jeśli przyjrzysz się stronie 36 dokumentu RFC2616, to znajdziesz składnię dla żądania GET. Aby poprosić o dokument z serwera WWW, wykonujemy połączenie z serwerem data.pr4e.org na porcie 80, a następnie wysyłamy linię w postaci

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

gdzie drugim parametrem jest strona internetowa, o którą prosimy, a następnie wysyłamy pustą linię. Serwer internetowy odpowie na zapytanie pewnym nagłówkiem i pustym wierszem, po którym wyśle treść dokumentu.

Czym natomiast jest port? Ogólnie rzeczy biorąc, jest to liczba wskazująca, z którą aplikacją kontaktujesz się podczas połączenia z serwerem. Dla przykładu, ruch internetowy zazwyczaj korzysta z portu 80 (HTTP) lub 443 (szyfrowany HTTP), podczas gdy ruch mailowy korzysta z portu 25.



Rysunek 12.1. Połączenie przez gniazdo

12.2. Najprostsza przeglądarka internetowa na świecie

Być może najprostszym sposobem na pokazanie, jak działa protokół HTTP, jest napisanie bardzo prostego programu, który nawiązuje połączenie z serwerem WWW i postępuje zgodnie z zasadami protokołu HTTP, żeby zażądać dokumentu i wyświetlić to, co serwer wysyłał z powrotem.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()

# Kod źródłowy: https://py4e.pl/code3/socket1.py
```

Najpierw program nawiązuje połączenie z serwerem data.pr4e.org na porcie 80. Ponieważ nasz program pełni rolę „przeglądarki internetowej”, protokół HTTP mówi, że musimy wysłać polecenie GET, a następnie pustą linię. `\r\n` oznacza EOL (ang. *end of line*, czyli koniec linii), a więc `\r\n\r\n` oznacza pustkę pomiędzy dwiema sekwencjami EOL. Jest to odpowiednik pustego wiersza.

Gdy już wyślemy pusty wiersz, piszemy pętlę, która odbiera z gniazda dane w 512-znakowych kawałkach i drukuje je aż do momentu, gdy nie będzie więcej danych do odczytania (czyli `recv()` zwróci pusty ciąg znaków).

Program generuje następujące wynik:

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
```

```
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Wyjście zaczyna się od nagłówków, które serwer WWW wysyła w celu opisania dokumentu. Na przykład nagłówek `Content-Type` wskazuje, że dokument jest zwykłym dokumentem tekstowym (`text/plain`).

Po wysłaniu przez serwer nagłówków dodaje on pustą linię wskazującą koniec nagłówków, a następnie wysyła dane rzeczywiście zawarte w pliku `romeo.txt`.

Powyższy przykład pokazuje, jak przy pomocy gniazd wykonać niskopoziomowe połączenie sieciowe. Gniazda mogą być używane do komunikacji z serwerem WWW, z serwerem pocztowym albo z różnymi innymi rodzajami serwerów. Wystarczy znaleźć dokument, który opisuje protokół, i napisać kod do wysłania i odbioru danych zgodnie z tym protokołem.

Ponieważ jednak najczęściej używanym przez nas protokołem jest protokół internetowy HTTP, Python posiada specjalną bibliotekę zaprojektowaną specjalnie do jego obsługi i pobierania dokumentów i danych umieszczonych w sieci.

Jednym z wymogów korzystania z protokołu HTTP jest konieczność wysyłania i odbierania danych w postaci obiektów bajtowych a nie ciągów znaków. W poprzednim przykładzie metody `encode()` i `decode()` przekształcają odpowiednio ciągi znaków na obiekty bajtowe i obiekty bajtowe na ciągi znaków.

Następny przykład używa notacji `b''` do określenia, że zmienna powinna być przechowywana jako obiekt bajtowy. `encode()` i `b''` są równoważne, o ile nie używamy np. znaków diakrytycznych (generalnie użycie `encode()` jest bardziej uniwersalne).

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

12.3. Pobieranie obrazu przez HTTP

W poprzednim przykładzie pobraliśmy zwykły plik tekstowy, który zawierał znaki końca linii i po prostu wyświetliliśmy te dane na ekranie w trakcie działania programu. Podobnego programu możemy użyć do odtworzenia obrazu za pomocą HTTP. Zamiast wyświetlać dane na ekranie w trakcie działania programu, zgromadzimy je w obiekcie bajtowym, usuniemy nagłówki, a następnie zapiszemy dane obrazu do pliku:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
```

```
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Znajdź koniec nagłówka (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Długość nagłówka:', pos)
print(picture[:pos].decode())

# Pomiń nagłówek i zapisz dane obrazu
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Kod źródłowy: https://py4e.pl/code3/urljpeg.py
```

Po uruchomieniu programu generuje on następujące wyjście:

```
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Długość nagłówka: 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Widać, że dla tego adresu URL nagłówek `Content-Type` wskazuje, że treść dokumentu jest obrazem (`image/jpeg`). Gdy program zakończy pracę, możesz obejrzeć dane obrazu, otwierając plik `stuff.jpg` w programie do przeglądania obrazów.

W trakcie działania programu widać, że nie otrzymujemy 5120 znaków za każdym razem, gdy wywołujemy metodę `recv()`. Otrzymujemy tyle znaków, ile zostało wysłanych do nas przez serwer WWW w momencie wywołania `recv()`. W tym przykładzie możemy otrzymać od 3200 do 5120 znaków za każdym razem, gdy żądamy danych.

W zależności od szybkości Twojej sieci te wartości mogą być jeszcze inne. Zauważ również, że przy ostatnim wywołaniu `recv()` otrzymujemy 3167 bajtów, czyli końcówkę strumienia, a przy następnym wywołaniu `recv()` otrzymujemy ciąg o zerowej długości, który mówi nam, że serwer wywołał `close()` po swojej stronie gniazda i nie przyjdzie już więcej danych.

Możemy spowolnić nasze kolejne wywołania `recv()` przez odkomentowanie wywołania `time.sleep()`. W ten sposób czekamy ćwierć sekundy po każdym wywołaniu, tak aby serwer mógł „wyprzedzić" nas i przesłać

nam więcej danych, zanim ponownie wywołamy recv(). Z włączonym opóźnieniem program wykonuje się w następujący sposób:

```
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Długość nagłówka: 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Teraz, poza pierwszym i ostatnim wywołaniem recv(), otrzymujemy 5120 znaków za każdym razem, gdy prosimy o nowe dane.

Pomiędzy serwerem wykonującym żądania send() a naszą aplikacją wykonującą żądania recv() jest bufor. Kiedy uruchamiamy program z założonym opóźnieniem, w pewnym momencie serwer może zapełnić bufor w gnieździe i być zmuszony zaczekać, aż nasz program zacznie opróżniać bufor. Wstrzymanie aplikacji wysyłającej lub odbierającej w takiej sytuacji nazywane jest „kontrolą przepływu".

12.4. Pobieranie stron internetowych za pomocą urllib

Choć możemy ręcznie wysyłać i odbierać dane za pomocą protokołu HTTP przy użyciu biblioteki gniazd, istnieje w Pythonie znacznie prostszy sposób na wykonanie tego zadania przy użyciu biblioteki urllib.

Kiedy używasz urllib, możesz traktować stronę internetową jak plik. Po prostu wskazujesz stronę internetową do pobrania, a urllib obsługuje wszystkie szczegóły dotyczące protokołu HTTP i nagłówka.

Równoważny kod odczytujący z sieci plik romeo.txt przy użyciu urllib jest następujący:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Kod źródłowy: https://py4e.pl/code3/urllib1.py
```

Po otwarciu strony internetowej za pomocą urllib.urlopen() możemy potraktować ją jak plik i odczytać za pomocą pętli for.

Po uruchomieniu programu widzimy tylko zawartość pliku. Nagłówki są nadal wysyłane przez serwer, ale kod urllib pochłania nagłówki i zwraca nam tylko dane.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Jako kolejny przykład możemy napisać program, który pobierze dane z `romeo.txt` i obliczy liczbę wystąpień każdego słowa:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Kod źródłowy: https://py4e.pl/code3/urlwords.py
```

Po otwarciu strony internetowej możemy odczytać jej zawartość jak z lokalnego pliku.

12.5. Odczytywanie plików binarnych za pomocą `urllib`

Czasami chcesz pobrać nietekstowy (lub binarny) plik, taki jak zdjęcie lub plik wideo. Dane w tych plikach na ogół nie są przydatne do wypisywania na ekranie, jednak za pomocą `urllib` możesz łatwo skopiować dane z pliku pod adresem URL do lokalnego pliku na dysku twardym.

Schemat postępowania polega na otwarciu adresu URL i użyciu `read()` do pobrania całej zawartości dokumentu do zmiennej zawierającej ciąg znaków (`img`), a następnie zapisaniu tej informacji do pliku lokalnego:

```
import urllib.request

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Kod źródłowy: https://py4e.pl/code3/curl1.py
```

Program ten odczytuje przez sieć wszystkie dane na raz i przechowuje je w zmiennej `img` w pamięci głównej komputera, a później otwiera plik `cover.jpg` i zapisuje dane na dysku. Argument `wb` w wywołaniu funkcji `open()` otwiera plik binarny tylko do zapisu. Program ten będzie działał, jeśli rozmiar pliku jest mniejszy niż rozmiar pamięci Twojego komputera.

Jeśli jednak jest to duży plik audio lub wideo, powyższy program może się zawiesić lub przynajmniej działać bardzo wolno, kiedy na Twoim komputerze zabraknie pamięci. Aby uniknąć wyczerpania pamięci, pobieramy dane w blokach (lub buforach), a następnie zapisujemy każdy blok na dysku przed pobraniem kolejnego bloku. W ten sposób program może odczytać plik o dowolnym rozmiarze bez zajmowania całej pamięci, którą posiadasz w komputerze.

```
import urllib.request

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    size = size + len(info)
```

```
    fhand.write(info)

print('Skopiowano', size, 'znaków.')
fhand.close()

# Kod źródłowy: https://py4e.pl/code3/curl2.py
```

W powyższym przykładzie odczytujemy jednocześnie tylko 100 000 znaków, a następnie zapisujemy te znaki w pliku `cover.jpg` przed pobraniem z sieci kolejnych 100 000 znaków danych.

Program działa następująco:

```
Skopiowano 230210 znaków.
```

12.6. Parsowanie HTMLa, roboty internetowe i web scraping

Jednym z częstych zastosowań `urllib` jest tzw. *web scraping* (dosł. wyłuskiwanie danych z sieci). Mamy z nim do czynienia wtedy, gdy piszemy program, który udaje przeglądarkę internetową i pobiera strony, a następnie analizujemy dane zawarte na tych stronach w poszukiwaniu interesujących nas wzorców.

Na przykład wyszukiwarka taka jak Google sprawdza kod źródłowy jednej strony internetowej, wyodrębnia linki do innych stron, a następnie pobiera te strony, wyodrębnia z nich linki itd. Używając tej techniki, Google poprzez swoje *roboty internetowe* indeksuje/przechodzi przez prawie wszystkie strony znajdujące się w sieci internetowej.

Google wykorzystuje również informacje o liczbie linków z innych stron do danej strony jako jedną z miar tego, jak „ważna" jest dana strona i jak wysoko powinna się ona znajdować w wynikach wyszukiwania.

12.7. Parsowanie HTMLa przy użyciu wyrażeń regularnych

Jednym z prostych sposobów parsowania HTMLa jest użycie wyrażeń regularnych do wielokrotnego wyszukiwania i wyodrębniania podciągów znaków, które pasują do konkretnego wzorca.

Poniżej mamy prostą stronę internetową:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

Możemy skonstruować wyrażenie regularne, tak aby dopasować i wyodrębnić z powyższego tekstu linki:

```
href="http[s]?://.+?"
```

Nasze wyrażenie regularne szuka napisów, które zaczynają się od „href="http://" lub „href="https://", po których następuje jeden lub więcej znaków (`.+?`), po których następuje kolejny cudzysłów. Znak zapytania `?` za `[s]` wskazuje na szukanie łańcucha „http", po którym następuje zero lub jedno „s".

Znak zapytania dodany do `.+?` wskazuje, że dopasowanie nie może być wykonane w trybie „zachłannym". Takie dopasowanie próbuje znaleźć *najmniejszy* możliwy pasujący ciąg znaków, a z kolei zachłanne dopasowanie próbuje znaleźć *największy* możliwy pasujący ciąg znaków.

Do naszego wyrażenia regularnego dodajemy nawiasy, aby wskazać, którą część pasującego napisu chcielibyśmy wydobyć. Program jest następujący:

```
# Wyszukaj linki w danych z URL
import urllib.request
import re
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b'href="(http[s]?://.*?)"', html)
for link in links:
    print(link.decode())

# Kod źródłowy: https://py4e.pl/code3/urlregex.py
```

Moduł `ssl` pozwala naszemu programowi na dostęp do stron internetowych, które ściśle wymuszają HTTPS. Metoda `read()` zwraca kod źródłowy HTML jako obiekt bajtowy, zamiast zwracać obiekt typu `HTTPResponse`. Metoda `findall()` daje nam listę wszystkich ciągów znaków, które pasują do naszego wyrażenia regularnego, zwracając tylko tekst linku zawarty pomiędzy cudzysłowami.

Kiedy uruchomimy program i wprowadzimy adres URL, otrzymamy mniej więcej następujący wynik:

```
Podaj link - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://www.python.org/
https://www.python.org/psf/donations/
http://sphinx.pocoo.org/
```

Wyrażenia regularne działają bardzo dobrze wtedy, gdy analizowany kod HTML jest dobrze sformatowany i przewidywalny. Niestety istnieje wiele „zepsutych" stron HTML, więc poleganie na rozwiązaniu opierającym się tylko na wyrażeniach regularnych może albo pominąć niektóre poprawne linki, albo skończyć się pozyskaniem niepoprawnych danych.

Problem ten można rozwiązać za pomocą solidnej i stabilnej biblioteki do parsowania HTMLa.

12.8. Parsowanie HTMLa przy użyciu `BeautifulSoup`

Nawet jeśli HTML wygląda jak XML[1] i niektóre strony faktycznie są starannie skonstruowane jako XML, to niestety większość stron HTML jest zazwyczaj popsuta do tego stopnia, że parser XMLa odrzuca całą stronę HTML jako nieprawidłowo sformatowany dokument.

Istnieje wiele bibliotek Pythona, które mogą pomóc Ci w parsowaniu HTMLa i wyodrębnieniu danych ze stron. Każda z tych bibliotek ma swoje mocne i słabe strony, a Ty możesz wybrać jedną z nich w zależności od swoich potrzeb.

[1] Format XML jest opisany w następnym rozdziale.

Jako przykład przetworzymy trochę danych HTML i wyodrębnimy linki za pomocą biblioteki BeautifulSoup. Ta biblioteka toleruje bardzo wadliwy kod HTML i nadal pozwala na łatwe wyodrębnienie potrzebnych danych. Możesz pobrać i zainstalować kod BeautifulSoup ze strony:

https://pypi.org/project/beautifulsoup4/

Bibliotekę możemy zainstalować poprzez wiersz linii poleceń, wydając komendę:

```
$ pip3 install beautifulsoup4
```

Bardziej szczegółowe informacje na temat instalacji bibliotek za pomocą narzędzia *Python Package Index* pip są dostępne na stronie internetowej:

https://packaging.python.org/tutorials/installing-packages/

Użyjemy urllib do odczytania strony, a następnie użyjemy BeautifulSoup do wyodrębnienia atrybutów href ze znacznika hiperłącza (a).

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

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Kod źródłowy: https://py4e.pl/code3/urllinks.py
```

Program pyta o adres internetowy, po czym otwiera stronę, odczytuje dane i przekazuje je do parsera BeautifulSoup, a następnie pobiera wszystkie znaczniki hiperłączy i dla każdego z nich wypisuje atrybut href.

Po uruchomieniu programu uzyskamy mniej więcej następujący wynik:

```
Podaj link - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
```

```
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

Tym razem lista jest o wiele dłuższa, ponieważ niektóre znaczniki hiperłączy są ścieżkami względnymi (np. „tutorial/index.html") lub odniesieniami wewnątrz strony (np. „#"), które nie zawierają „http://" lub „https://", co było wymogiem w naszym podejściu z wyrażeniem regularnym.

Możesz również użyć BeautifulSoup do wyciągnięcia różnych elementów związanych ze znacznikiem HTML:

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

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    # Przejrzyj elementy związane ze znacznikiem
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Kod źródłowy: https://py4e.pl/code3/urllink2.py
```

```
Podaj link - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
```

```
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Contents:
Second Page
Attrs: {'href': 'http://www.dr-chuck.com/page2.htm'}
```

`html.parser` to parser HTMLa zawarty w standardowej bibliotece Pythona 3. Informacje o innych parserach HTMLa są dostępne na stronie internetowej:

https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

Powyższe przykłady są tylko namiastką możliwości, jakie daje `BeautifulSoup`, jeśli chodzi o parsowanie HTMLa.

12.9. Dodatek dla użytkowników systemu Unix / Linux

Jeśli posiadasz komputer z systemem Linux, Unix lub macOS, prawdopodobnie masz wbudowane w system komendy, które pobierają zarówno zwykły tekst, jak i pliki binarne za pomocą protokołów HTTP lub FTP (File Transfer Protocol, czyli protokół transferu plików). Jedną z tych komend jest `curl`:

```
$ curl -O https://www.py4e.com/cover.jpg
```

Polecenie `curl` jest skrótem od „copy URL" (z ang. skopiuj URL) i dwa wymienione wcześniej na https://py4e.pl/code3 przykłady pobierania plików binarnych z `urllib` mają nazwy `curl1.py` i `curl2.py`, ponieważ implementują podobną funkcjonalność jak polecenie `curl`. Istnieje również przykładowy program `curl3.py`, który wykonuje to zadanie nieco efektywniej, na wypadek gdybyś rzeczywiście chciał użyć tego schematu w swoim programie.

Drugim poleceniem, które działa bardzo podobnie, jest `wget`:

```
$ wget http://www.py4e.com/cover.jpg
```

Obie te komendy sprawiają, że pobieranie stron internetowych i plików jest prostą czynnością.

12.10. Słowniczek

BeautifulSoup Biblioteka Pythona do przetwarzania dokumentów HTML i wyodrębniania danych z dokumentów HTML, która kompensuje większość niedoskonałości w kodzie HTML, zazwyczaj ignorowanych przez przeglądarki internetowe. Biblioteka `BeautifulSoup` ma swoją stronę pod adresem https://www.crummy.com/software/BeautifulSoup.

gniazdo Połączenie sieciowe pomiędzy dwiema aplikacjami, w którym aplikacje mogą wysyłać i odbierać dane w obu kierunkach.

port Liczba ogólnie wskazująca, z którą aplikacją kontaktujesz się podczas połączenia z serwerem. Dla przykładu, ruch internetowy zazwyczaj korzysta z portu 80 (HTTP) lub 443 (szyfrowany HTTP), podczas gdy ruch mailowy korzysta z portu 25.

robot internetowy Działanie wyszukiwarek internetowych polegające na pobieraniu strony, a następnie wszystkich stron, do których prowadzą z niej linki itd., aż do momentu, gdy zgromadzą prawie wszystkie strony w internecie, których używają do zbudowania swojego indeksu wyszukiwania.

web scraping Sytuacja w której program podszywa się pod przeglądarkę internetową, pobiera stronę i przegląda jej zawartość. Często programy podążają za linkami na jednej stronie, by znaleźć następną, dzięki czemu mogą przemierzyć duże zbiory stron lub serwisy społecznościowe.

12.11. Ćwiczenia

Ćwiczenie 1. Zmień program używający gniazda `socket1.py` tak, aby poprosić użytkownika o podanie adresu URL i by mógł on przeczytać dowolną stronę internetową. Możesz użyć `split('/')`, aby rozbić URL na części składowe, dzięki czemu łatwo wyodrębnisz nazwę hosta dla metody `connect()`. Dodaj sprawdzanie błędów przy użyciu `try` i `except`, aby poradzić sobie z przypadkiem, w którym użytkownik wprowadzi nieprawidłowo sformatowany lub nieistniejący adres URL. Przetestuj działanie na http://data.pr4e.org/romeo.txt.

Ćwiczenie 2. Zmień swój program używający gniazda tak, by zliczał liczbę otrzymanych znaków i przestawał wyświetlać jakikolwiek tekst po wyświetleniu 3000 znaków. Program powinien pobrać cały dokument, zliczyć całkowitą liczbę znaków oraz na końcu ją wyświetlić. Przetestuj działanie na http://data.pr4e.org/romeo-full.txt.

Ćwiczenie 3. Użyj `urllib`, aby powtórzyć poprzednie ćwiczenie dotyczące (1) pobrania dokumentu z adresu URL, (2) wyświetlenia do 3000 znaków, (3) zliczenia całkowitej liczby znaków w dokumencie. Nie martw się o nagłówki, po prostu pokaż pierwsze 3000 znaków dokumentu. Przetestuj działanie na http://data.pr4e.org/romeo-full.txt.

Ćwiczenie 4. Zmień program `urllinks.py` tak, by wyodrębnić i policzyć znaczniki akapitu (`p`) z pobranego dokumentu HTML i wyświetlić liczbę akapitów jako wynik programu. Nie wyświetlaj tekstu akapitów, a jedynie je zliczaj. Przetestuj swój program na kilku niewielkich stronach internetowych, jak również na kilku większych.

Ćwiczenie 5. (Zaawansowane) Zmień program używający gniazda tak, by wyświetlał dane dopiero po otrzymaniu nagłówków i pustej linii. Pamiętaj, że `recv()` otrzymuje znaki (wszystkie, włączając w to znaki końca linii), a nie linie. Przetestuj działanie na http://data.pr4e.org/romeo-full.txt.