Poniższy rozdział pochodzi z książki:

Charles R. Severance - "Python dla wszystkich: Odkrywanie danych z Python 3"

Pełna wersja podręcznika znajduje się na stronie https://py4e.pl/book

Rozdział 11

Wyrażenia regularne

Do tej pory odczytywaliśmy dane poprzez pliki, szukaliśmy wzorców i wyodrębnialiśmy różne fragmenty linii, które wydawały się nam interesujące. Używaliśmy metod związanych z napisami, takich jak `split()` i `find()`, a także używaliśmy list i wycinania fragmentów napisów do wyodrębnienia części linii.

Zadanie wyszukiwania i wyodrębniania tekstu jest w praktyce wykonywane tak często, że Python ma bardzo wszechstronną i wydajną bibliotekę o nazwie *wyrażenia regularne*, która elegancko radzi sobie z tymi zadaniami. Powodem, dla którego nie wprowadziliśmy wcześniej w książce wyrażeń regularnych jest to, że choć mają dużo możliwości i są bardzo przydatne, to niestety są trochę skomplikowane, a przyzwyczajenie się do ich składni wymaga trochę czasu.

Wyrażenia regularne (w skrócie z ang. *regex*) są prawie samodzielnym, małym językiem programowania do wyszukiwania i parsowania napisów. O wyrażeniach regularnych napisano już wiele książek; w tym rozdziale zajmiemy się tylko ich podstawami. Więcej szczegółów na temat wyrażeń regularnych znajdziesz na stronach:

- https://pl.wikipedia.org/wiki/Wyra%C5%BCenie_regularne
- https://docs.python.org/library/re.html

Zanim będziesz mógł korzystać w swoim programie z wyrażeń regularnych, musisz zaimportować bibliotekę `re`. Najprostszym wykorzystaniem tej biblioteki jest funkcja `search()`. Poniższy program demonstruje jej podstawowe użycie.

```
# Szukaj linii, które zawierają 'From:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/re01.py
```

Otwieramy plik, przechodzimy przez każdą linię i używamy w `search()` wyrażenia regularnego do wypisania tylko tych linii, które zawierają napis "From:". Powyższy program nie wykorzystuje prawdziwej mocy wyrażeń regularnych, ponieważ równie łatwo mogliśmy użyć `line.find()` do uzyskania tego samego wyniku.

Moc wyrażeń regularnych ujawnia się wtedy, gdy do naszego tekstowego wzorca wyszukiwania dodajemy specjalne znaki, które pozwalają nam dokładniej kontrolować to, które linie do niego pasują. Dodanie tych specjalnych znaków do naszego wzorca pozwala nam na wyrafinowane dopasowanie i ekstrakcję danych przy jednoczesnym użyciu bardzo małej ilości kodu.

Na przykład znak karety ^ (tzw. daszek) jest używany w wyrażeniach regularnych, by dopasować "początek" linii. Moglibyśmy zmienić nasz program na dopasowywanie tylko tych linii, w których "From:" było na początku, w następujący sposób:

```
# Szukaj linii, które zaczynają się od 'From:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/re02.py
```

Teraz będziemy dopasowywać tylko te linie, które *zaczynają się od* napisu "From:". Jest to wciąż bardzo prosty przykład, który mogliśmy zrealizować równoważnie przy pomocy metody `startswith()` związanej z tekstowym typem danych. Powyższy przykład wskazuje jednak, że wyrażenia regularne zawierają specjalne znaki akcji, które dają nam większą kontrolę nad dopasowaniami.

11.1. Dopasowywanie znaków w wyrażeniach regularnych

Istnieje wiele innych specjalnych znaków, które pozwalają nam budować jeszcze bardziej wyrafinowane wyrażenia regularne. Najczęściej używanym znakiem specjalnym jest kropka `.`, która dopasowuje każdy znak.

W poniższym przykładzie wyrażenie regularne `F..m:` pasuje do któregokolwiek z napisów "From:", "Fxxm:", "F12m:" lub "F!@m:", ponieważ kropka w wyrażeniu regularnym oznacza dopasowanie dowolnego znaku.

```
# Wyszukaj linie zaczynające się od 'F',
# po których następują 2 dowolne znaki,
# a następnie 'm:'.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/re03.py
```

Takie wyrażenie regularne ma szczególnie dużą moc, gdy przy pomocy `*` lub `+` poszerzy się je o wskazanie, że dany znak może być powtórzony dowolną liczbę razy. Te znaki specjalne oznaczają, że zamiast dopasowywać pojedynczy znak we wzorcu wyszukiwania, dopasowują zero lub więcej znaków (w przypadku asterysku/gwiazdki `*`) lub jeden lub więcej znaków (w przypadku plusa `+`).

W poniższym przykładzie możemy jeszcze bardziej zawęzić dopasowywane linie używając powtarzającego się *wieloznacznika* (symbolu wieloznacznego, symbolu maski):

```
# Szukaj linii, które zaczynają się od 'From:' i zawierają znak małpy
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/re04.py
```

Wzorzec wyszukiwania `^From:.+@` pomyślnie dopasuje te linie, które zaczynają się od "From:", po których następuje jeden lub więcej znaków (`.+`) i po których następuje znak małpy "@". Zatem będzie on pasował do następującej linii:

```
From: stephen.marquard@uct.ac.za
```

Możesz myśleć o symbolu wieloznacznym .+ jako o rozszerzeniu, które dopasowuje wszystkie znaki między znakiem dwukropka a znakiem małpy.

```
From:.+@
```

Dobrze jest też myśleć o plusie i gwiazdce jako o znakach "ekspansywnych". Na przykład następujący wzorzec dopasowuje tekst aż do ostatniego znaku małpy w napisie, jako że .+ "wychodzi do zewnątrz":

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

Poprzez dodanie kolejnego znaku można też wskazać, że znak gwiazdki lub plusa nie ma być aż tak "zachłanny". Więcej informacji na temat wyłączania zachłannego zachowania tych znaków znajduje się w dokumentacji.

11.2. Wyciąganie danych przy użyciu wyrażeń regularnych

Jeśli w Pythonie chcemy wyodrębnić dane z tekstu, to możemy użyć metody `findall()` do wyodrębnienia wszystkich podciągów, które pasują do wyrażenia regularnego. Użyjemy przykładu, w którym w dowolnej linii (niezależnie od jej formatu) chcemy wyodrębnić wszystko to, co wygląda jak adres email. Na przykład chcemy wyodrębnić adresy email z każdej z poniższych linii:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

Nie chcemy pisać specjalnego kodu dla każdego z typów linii oraz różnego dzielenia i wycinania napisów dla poszczególnych typów linii. Następujący program używa `findall()` do znalezienia linii z adresami mailowymi i do wyciągnięcia z nich jednego lub więcej adresów.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Kod źródłowy: https://py4e.pl/code3/re05.py
```

Metoda `findall()` przeszukuje napis podany w drugim argumencie i zwraca listę wszystkich ciągów znaków, które wyglądają jak adresy e-mail. Używamy sekwencji dwuznakowej, która oznacza dopasowanie dowolnego znaku, który nie jest tzw. białym znakiem[1] (\S).

Wynik programu jest następujący:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Tłumacząc wyrażenie regularne na ludzki język: szukamy podciągów, które zawierają co najmniej jeden *niebiały* znak, po którym następuje znak małpy, po którym następuje co najmniej jeszcze jeden niebiały znak. \S+ dopasowuje tyle niebiałych znaków, ile tylko się da.

Wyrażenie regularne pasuje dwukrotnie ("csev@umich.edu" i "cwen@iupui.edu"), ale nie pasuje do napisu "@2PM", ponieważ nie ma w nim żadnych niebiałych znaków *przed* znakiem małpy. Możemy w następujący sposób użyć tego wyrażenia regularnego, by odczytać wszystkie linie w pliku i wypisać wszystko to, co wygląda jak adres e-mail:

```
# Szukaj linii, które zawierają znak małpy pomiędzy innymi znakami
import re
```

[1] Przypomnijmy, że jest to zbiorcze określenie takich znaków jak spacja, znak tabulacji, znak końca linii lub dowolny inny znak niemający kształtu na ekranie.

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://py4e.pl/code3/re06.py
```

Czytamy każdą linię, a następnie wyodrębniamy wszystkie podciągi, które pasują do naszego wyrażenia regularnego. Ponieważ `findall()` zwraca listę, po prostu sprawdzamy, czy liczba elementów na tej liście jest większa od zera, tak by wypisać tylko te linie, w których znaleźliśmy przynajmniej jeden podciąg wyglądający jak adres e-mail.

Jeśli uruchomimy program na pliku `mbox.txt`, otrzymamy następujący wynik:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Niektóre z tych adresów e-mail mają nieprawidłowe znaki, takie jak "<" lub ";", na początku lub na końcu. Zaznaczymy w programie, że interesuje nas tylko ta część napisu, która zaczyna i kończy się literą lub cyfrą.

Aby to zrobić, używamy innej cechy wyrażeń regularnych. Nawiasy kwadratowe `[]` służą do wskazania zbioru akceptowalnych znaków, które jesteśmy skłonni rozważyć jako pasujące. W pewnym sensie `\S` prosi o dopasowanie dowolnych "niebiałych znaków". Teraz będziemy nieco bardziej precyzyjni, jeśli chodzi o znaki, które chcemy dopasowywać.

Oto nasze nowe wyrażenie regularne:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Robi się to już trochę skomplikowane i być może zaczynasz rozumieć, dlaczego wyrażenia regularne są samodzielnym, małym językiem. Tłumacząc to wyrażenie regularne na ludzki język: szukamy podciągów, które rozpoczynają się *pojedynczą* małą literą, dużą literą lub liczbą (`[a-zA-Z0-9]`), po których następuje zero lub więcej niebiałych znaków (`\S*`), po których następuje znak małpy (`@`), po których następuje zero lub więcej niebiałych znaków (`\S*`), po których następuje duża lub mała litera (`[a-zA-Z]`). Zauważ, że zmieniliśmy znak `+` na `*`, tak by wskazać zero lub więcej niebiałych znaków, ponieważ `[a-zA-Z0-9]` jest już jednym niebiałym znakiem. Pamiętaj, że `*` lub `+` odnosi się do pojedynczego znaku znajdującego się bezpośrednio po lewej stronie plusa lub gwiazdki.

Jeśli użyjemy tego wyrażenia regularnego w naszym programie, nasze dane będą wyglądać porządniej:

```
# Szukaj linii, które zawierają znak małpy pomiędzy innymi znakami
# Znak początkowy musi być literą lub cyfrą
# Znak końcowy musi być literą
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://py4e.pl/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Zauważ, że w liniach `source@collab.sakaiproject.org` nasze wyrażenie regularne wyeliminowało dwa znaki na końcu napisu (“>;”). Dzieje się tak dlatego, że kiedy dołączamy `[a-zA-Z]` na końcu wyrażenia regularnego, wymagamy, by jakikolwiek ciąg znaków, który znajdzie parser wyrażenia, musiał kończyć się literą. Więc kiedy parser widzi “>” na końcu “sakaiproject.org>;”, to po prostu zatrzymuje się na ostatniej znalezionej “pasującej” literze (w tym przypadku “g” było ostatnim dobrym dopasowaniem).

Zauważ również, że wynik programu jest listą, której jedynym elementem jest napis.

11.3. Łączenie wyszukiwania i wyodrębniania tekstu

Załóżmy, że chcemy znaleźć liczby w liniach, które zaczynają się od ciągu znaków “X-”, np.:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

Nie chcemy dowolnych liczb zmiennoprzecinkowych z dowolnych linii. Chcemy wyciągnąć z tekstu tylko liczby i tylko z tych linii, które mają powyższą składnię, czyli zaczynają się od “X-”.

Możemy skonstruować następujące wyrażenie regularne, tak aby wybrać interesujące nas linie:

```
^X-.*: [0-9.]+
```

Tłumacząc to na ludzki, mówimy, że chcemy te linie, które zaczynają się od `X-`, po których następuje zero lub więcej znaków (`.*`), po których następuje dwukropek (`:`), a następnie spacja. Po spacji szukamy jednego lub więcej znaków, które są albo cyfrą, albo kropką (`[0-9.]+`). Zauważ, że wewnątrz nawiasów kwadratowych kropka odpowiada rzeczywistej kropce (tzn. pomiędzy nawiasami kwadratowymi nie jest ona traktowana jako wieloznacznik).

Jest to bardzo zwięzłe wyrażenie, które będzie pasowało tylko do interesujących nas linii, jak widać poniżej:

```
# Szukaj linii, które zaczynają się od 'X-',
# po których występują dowolne niebiałe znaki oraz ':',
# po których występuje spacja i dowolna liczba.
# Liczba może być całkowita.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X-\S*: [0-9.]+', line):
        print(line)

# Kod źródłowy: https://py4e.pl/code3/re10.py
```

Kiedy uruchomimy program, zobaczymy ładnie przefiltrowane dane, które zawierają tylko szukane przez nas linie.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
```

```
X-DSPAM-Probability: 0.0000
...
```

Teraz musimy rozwiązać problem wyodrębniania liczb. Chociaż da się to zrobić bardzo prosto przy użyciu `split()`, to jednak możemy użyć kolejnej cechy wyrażeń regularnych, tak by w tym samym czasie zarówno wyszukać, jak i przetworzyć linię.

Nawiasy okrągłe `()` są kolejną szczególną cechą wyrażeń regularnych. Kiedy dodajemy nawiasy okrągłe do wyrażenia regularnego, są one ignorowane podczas dopasowywania ciągu znaków. Natomiast kiedy używasz `findall()`, nawiasy okrągłe wskazują, że w momencie gdy chcesz, by całe wyrażenie pasowało, to jesteś zainteresowany wyciągnięciem tylko części podciągu (wewnątrz nawiasów okrągłych) z dopasowania.

Tak więc dokonujemy następującej zmiany w naszym programie:

```
# Szukaj linii, które zaczynają się od 'X-',
# po których występują dowolne niebiałe znaki oraz ':',
# po których występuje spacja i dowolna liczba.
# Liczba może być całkowita.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X-\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://py4e.pl/code3/re11.py
```

Zamiast wywołania `search()`, dodajemy nawiasy okrągłe wokół tej części wyrażenia regularnego, która reprezentuje liczbę zmiennoprzecinkową, tak by wskazać `findall()`, że chcemy pozyskać z pasującego napisu tylko część dotyczącą liczby zmiennoprzecinkowej.

Wynik programu jest następujący:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
...
```

Liczby wciąż są elementami list i muszą zostać przekonwertowane z tekstowego typu danych na typ zmiennoprzecinkowy. Niemniej jednak użyliśmy wachlarza możliwości wyrażeń regularnych zarówno do wyszukiwania, jak i wydobywania informacji, które uznaliśmy za interesujące.

A oto inny przykład użycia powyższej techniki. Jeśli spojrzymy na zawartość pliku, to znajdziemy tam wiele linii w następującej postaci:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Gdybyśmy chcieli wyciągnąć wszystkie numery zmian (liczba całkowita na końcu tego typu wierszy) przy użyciu tej samej techniki, co chwilę wcześniej, moglibyśmy napisać następujący program:

```
# Szukaj linii, które mają postać 'Details:...rev='
# i są zakończone liczbą
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://py4e.pl/code3/re12.py
```

Tłumacząc nasze wyrażenie regularne po ludzku: szukamy linii, które rozpoczynają się od `Details:`, po których następuje dowolna liczba znaków (`.*`), po których następuje `rev=`, a następnie jedna lub więcej cyfr. Chcemy znaleźć te wiersze, które pasują do całego wyrażenia, ale wyciągnąć z nich tylko liczbę całkowitą znajdującą się na końcu, więc otaczamy `[0-9]+` nawiasami okrągłymi.

Kiedy uruchamiamy program, otrzymujemy następujący wynik:

```
['39772']
['39771']
['39770']
['39769']
...
```

Pamiętaj, że `[0-9]+` jest "zachłanne", więc zanim wydobędziemy te cyfry, to stara się on uzyskać jak najdłuższy ciąg cyfr. To "zachłanne" zachowanie jest powodem, dla którego otrzymujemy wszystkie pięć cyfr dla każdej liczby. Biblioteka wyrażeń regularnych przeszukuje w obu kierunkach tak długo, aż napotka coś innego niż cyfra albo na początek lub koniec wiersza.

Teraz możemy użyć wyrażeń regularnych, aby powtórzyć ćwiczenie z wcześniejszej części książki, w której byliśmy zainteresowani porą dnia każdej wiadomości. Szukaliśmy linii w postaci:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

i dla każdej linii chcieliśmy wyodrębnić godzinę. Wcześniej robiliśmy to za pomocą dwóch wywołań `split()`. Najpierw linia była dzielona na słowa, a następnie wyciągaliśmy piąte słowo i ponownie dzieliliśmy je po znaku dwukropka, tak aby wyciągnąć pierwsze dwa znaki, które nas interesowały.

Pomimo działania, w rzeczywistości skutkowało to dość *kruchym kodem*, który zakłada, że linie są ładnie sformatowane. Jeśli dodałbyś wystarczająco dużo sekcji sprawdzających błędy (lub duży blok `try/except`), by upewnić się, że Twój program nigdy nie zawiedzie podczas analizy źle sformatowanych linii, kod rozrósłby się do 10-15 linii, które byłyby dość ciężkie do odczytania.

Na szczęście możemy osiągnąć nasz cel w znacznie prostszy sposób, używając następującego wyrażenia regularnego:

```
^From .* [0-9][0-9]:
```

Tłumaczenie tego wyrażenia regularnego jest takie, że szukamy linii, które zaczynają się od `From` (zwróć uwagę na spację), po których następuje dowolna liczba znaków (`.*`), po których następuje spacja, po której następują dwie cyfry `[0-9][0-9]`, po których następuje znak dwukropka `:`. Jest to definicja typów linii, których szukamy.

Aby wyciągnąć tylko godzinę przy użyciu `findall()`, dodajemy nawiasy okrągłe wokół dwóch cyfr:

```
^From .* ([0-9][0-9]):
```

W ten sposób powstaje następujący program:

```
# Szukaj linii, które zaczynają się od 'From '
# i dowolnych znaków, po których następuje spacja
# i dwie cyfry od 00 do 99, po których występuje ':'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://py4e.pl/code3/re13.py
```

Gdy uruchomimy program, uzyskamy taki wynik:

```
['09']
['18']
['16']
['15']
...
```

11.4. Znak ucieczki

W wyrażeniach regularnych używamy znaków specjalnych aby dopasować początek lub koniec wiersza lub określić wieloznaczniki, zatem potrzebujemy jakiegoś sposobu na wskazanie, że te same znaki mogą być "normalne" i chcemy dopasować rzeczywisty znak, taki jak znak dolara lub daszka.

Możemy wskazać, że chcemy po prostu dopasować znak, poprzedzając go odwrotnym ukośnikiem \ – jest to tzw. *znak ucieczki*[2]. Na przykład przy pomocy poniższego wyrażenia regularnego możemy znaleźć kwoty pieniędzy.

```
import re
x = 'Uzyskaliśmy $10.00 za ciasteczka.'
y = re.findall('\$[0-9.]+', x)
print(y)
```

Ponieważ poprzedzamy znak dolara odwrotnym ukośnikiem, w rzeczywistości w wejściowym napisie odpowiada on znakowi dolara (zamiast oznaczać "koniec linii"), a reszta wyrażenia regularnego odpowiada jednej lub kilku cyfrom lub znakowi kropki.

Przypomnijmy jeszcze raz: w nawiasach kwadratowych znaki nie są traktowane jako "specjalne". Jeśli napiszemy `[0-9.]`, to w rzeczywistości oznacza to cyfry lub kropkę. Poza nawiasami kwadratowymi kropka jest znakiem symbolu wieloznacznego i dopasowuje każdy znak. W nawiasach kwadratowych kropka jest kropką.

11.5. Podsumowanie

Podczas gdy tylko zarysowaliśmy niewielki zakres wyrażeń regularnych, to jednak nauczyliśmy się trochę o ich języku. Są to ciągi specjalnych znaków przeznaczone do wyszukiwania, które przekazują Twoje życzenia do systemu wyrażeń regularnych, który z kolei definiuje "dopasowanie" i wyodrębnia dopasowany tekst z przetwarzanych napisów. Oto niektóre z tych specjalnych znaków i sekwencji znaków:

`^` Dopasowuje początek linii.

`$` Dopasowuje koniec linii.

`.` Dopasowuje dowolny znak (wieloznacznik).

`\s` Dopasowuje biały znak.

`\S` Dopasowuje niebiały znak (przeciwieństwo `\s`).

`*` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je zero lub więcej razy.

`*?` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je zero lub więcej razy, "wyłączając tryb zachłanny".

`+` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je jeden lub więcej razy.

`+?` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je jeden lub więcej razy, "wyłączając tryb zachłanny".

`?` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je zero lub jeden raz.

[2] Inne nazwy to znak modyfikacji lub znak uwalniania; w praktyce programistycznej proces nadawania normalnego znaczenia znakom specjalnym czasem określa się *eskejpowaniem*, które pochodzi od angielskiego określenia znaku ucieczki, tj. *escape character*.

`??` Dotyczy bezpośrednio poprzedzającego znaku lub znaków i wskazuje, by dopasować je zero lub jeden raz, "wyłączając tryb zachłanny".

`[aeiou]` Dopasowuje pojedynczy znak, o ile ten znak jest w określonym zestawie. W tym przykładzie dopasowanie dotyczy "a", "e", "i", "o" lub "u", ale nie dopasowuje żadnych innych znaków.

`[a-z0-9]` Możesz określić zakresy znaków, używając znaku minus. Ten przykład to pojedynczy znak, który musi być małą literą lub cyfrą.

`[^A-Za-z]` Gdy pierwszym znakiem w zapisie zestawu znaków jest kareta `^` (daszek), to oznacza on zaprzeczenie. Ten przykład odpowiada pojedynczemu znakowi, który jest *czymś innym* niż duże lub małe litery.

`( )` Gdy nawiasy okrągłe są dodawane do wyrażenia regularnego, są one ignorowane podczas dopasowania, ale pozwalają na wyodrębnienie konkretnego podciągu dopasowanego napisu, a nie całego napisu tak jak podczas użycia `findall()`.

`\b` Dopasowuje pusty napis, ale tylko na początku lub na końcu słowa.

`\B` Dopasowuje pusty napis, ale ani na początku, ani na końcu słowa.

`\d` Dopasowuje dowolną cyfrę dziesiętną; odpowiednik `[0-9]`.

`\D` Dopasowuje dowolny znak niebędący cyfrą; odpowiednik zestawu `[^0-9]`.

11.6. Dodatek dla użytkowników systemu Unix / Linux

Wsparcie wyszukiwania plików za pomocą wyrażeń regularnych zostało wbudowane w system operacyjny Unix w latach 60. i jest dostępne w prawie wszystkich językach programowania w takiej czy innej formie.

W rzeczywistości Unix posiada wbudowany program wiersza poleceń o nazwie *grep* (Generalized Regular Expression Parser, czyli uogólniony parser wyrażeń regularnych), który robi prawie to samo, co przykłady z `search()` umieszczone w tym rozdziale. Zatem jeśli masz system macOS lub Linux, możesz spróbować następujących komend w oknie terminala:

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Powyższe polecenie mówi programowi `grep`, żeby pokazał Ci linie, które zaczynają się od napisu "From:" w pliku `mbox-short.txt`. Jeśli trochę poeksperymentujesz z komendą `grep` i przeczytasz dokumentację tego programu, znajdziesz kilka subtelnych różnic między obsługą wyrażeń regularnych w Pythonie a obsługą wyrażeń regularnych w `grep`. Na przykład `grep` w macOS nie obsługuje `\S` oznaczającego niebiały znak, więc będziesz musiał użyć nieco bardziej złożonego zapisu (w uproszczeniu) `[^ \n\r\t]`, co po prostu oznacza dopasowanie znaku, który jest czymś innym niż spacja, znaki końca linii lub tabulacja.

11.7. Debugowanie

Python ma wbudowaną prostą i podstawową dokumentację, która może być bardzo pomocna, jeśli musisz szybko przypomnieć sobie nazwę danej metody. Dokumentacja ta może być przeglądana w trybie interaktywnym w interpreterze Pythona.

Możesz wywołać interaktywny system pomocy, używając `help()`.

```
>>> help()

help> modules
```

Aby wyjść z systemu pomocy, wpisz `quit`.

Jeśli wiesz którego modułu chcesz użyć, możesz użyć polecenia `dir()` w następujący sposób, tak aby znaleźć metody dostępne w tym module:

```
>>> import re
>>> dir(re)
[..., 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

Za pomocą polecenia `help()` możesz również uzyskać niewielką część dokumentacji na temat określonej metody.

```
>>> help(re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

Wbudowana dokumentacja nie jest zbyt obszerna, ale może być pomocna, gdy potrzebujesz coś szybko znaleźć lub nie masz dostępu do przeglądarki internetowej lub wyszukiwarki.

11.8. Słowniczek

dopasowanie zachłanne Pojęcie oznaczające, że znaki + i * w wyrażeniu regularnym rozszerzają się na zewnątrz, tak aby dopasować jak najdłuższy możliwy ciąg znaków.

grep Polecenie dostępne w większości systemów unixowych, które przeszukuje pliki tekstowe w poszukiwaniu linii pasujących do wyrażeń regularnych. Nazwa komendy oznacza "Generalized Regular Expression Parser", czyli uogólniony parser wyrażeń regularnych.

kruchy kod Kod programu, który działa tylko wtedy, gdy dane wejściowe są w określonym formacie, ale jest podatny na wysypanie całego programu, jeśli wystąpią jakieś odchylenia od właściwego formatu. Nazywamy to "kruchym kodem", ponieważ można go łatwo zepsuć.

wieloznacznik Znak specjalny, który pasuje do dowolnego znaku. W wyrażeniach regularnych znakiem wieloznacznika jest kropka.

wyrażenie regularne Język służący do tworzenia bardziej złożonych ciągów wyszukiwania. Wyrażenie regularne może zawierać znaki specjalne, które wskazują, że szukane wyrażenie pasuje tylko do początku lub końca wiersza lub wielu innych podobnych możliwości.

11.9. Ćwiczenia

Ćwiczenie 1. Napisz prosty program symulujący działanie unixowej komendy `grep`. Poproś użytkownika o wpisanie wyrażenia regularnego i zlicz linie, które do niego pasują:

```
Podaj wyrażenie regularne: ^Author
mbox.txt ma 1798 linii, które pasują do ^Author
```

```
Podaj wyrażenie regularne: ^X-
mbox.txt ma 14368 linii, które pasują do ^X-
```

```
Podaj wyrażenie regularne: java$
mbox.txt ma 4175 linii, które pasują do java$
```

Wskazówka: zauważ, że w przypadku wzorca `java$` otrzymujemy 4175 linii. Jeśli otrzymujesz 4218 linii, to prawdopodobnie podczas przetwarzania danego wiersza ucinasz w nim z prawej strony zbyt dużo białych znaków.

Ćwiczenie 2. Napisz program znajdujący linie w postaci:

```
New Revision: 39772
```

Wyodrębnij liczbę z każdej pasującej linii za pomocą wyrażenia regularnego i metody `findall()`. Oblicz średnią z tych liczb i wypisz ją w postaci liczby całkowitej.

```
Podaj nazwę pliku: mbox.txt
38549

Podaj nazwę pliku: mbox-short.txt
39756
```