정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth. Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

The LaTeX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from `http://www.thinkpython.com`.

Chapter 1

정규 표현식

지금까지 파일을 훑어서 패턴을 찾고, 관심있는 라인에서 다양한 비트(bits)를 뽑아냈다. `split`, `find` 같은 문자열 메쏘드를 사용하였고, 라인에서 일정 부분을 뽑아내기 위해서 리스트와 문자열 슬라이싱(slicing)을 사용했다.

검색하고 추출하는 작업은 너무 자주 있는 일이어서 파이썬은 상기와 같은 작업을 매우 우아하게 처리하는 **정규 표현식(regular expressions)**으로 불리는 매우 강력한 라이브러리를 제공한다. 정규 표현식을 책의 앞부분에 소개하지 않은 이유는 정규 표현식 라이브러리가 매우 강력하지만, 약간 복잡하고, 구문에 익숙해지는데 시간이 필요하기 때문이다.

정규표현식은 문자열을 검색하고 파싱하는데 그 자체가 작은 프로그래밍 언어다. 사실, 책 전체가 정규 표현식을 주제로 쓰여진 책이 몇권있다. 이번 장에서는 정규 표현식의 기초만을 다룰 것이다. 정규 표현식의 좀더 자세한 사항은 다음을 참조하라.

`http://en.wikipedia.org/wiki/Regular_expression`

`http://docs.python.org/library/re.html`

정규 표현식 라이브러리는 사용하기 전에 프로그램에 가져오기(import)하여야 한다. 정규 표현식 라이브러리의 가장 간단한 쓰임은 `search()` 검색 함수다. 다음 프로그램은 검색 함수의 사소한 사용례를 보여준다.

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

파일을 열고, 매 라인을 루프를 반복해서 정규 표현식 `search()` 메쏘드를 호출하여 문자열 "From"이 포함된 라인만 출력한다. 상기 프로그램은 정규 표현식의 진정 강력한 기능을 사용하지 않았다. 왜냐하면, `line.find()` 메쏘드를 가지고 동일한 결과를 쉽게 구현할 수 있기 때문이다.

정규 표현식의 강력한 기능은 문자열에 해당하는 라인을 좀더 정확하게 제어하기 위해서 검색 문자열에 특수문자를 추가할 때 확인할 수 있다. 매우 적은 코드를 작성해도 정규 표현식에 특수 문자를 추가하는 것 만으로도 정교한 일치(matching)와 추출이 가능하게 한다.

예를 들어, 탈자 기호(caret)는 라인의 "시작"과 일치하는 정규 표현식에 사용된다. 다음과 같이 "From:"으로 시작하는 라인만 일치하는 응용프로그램을 변경할 수 있다.

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

"From:" 문자열로 *시작*하는 라인만 일치할 수 있다. 여전히 매우 간단한 프로그램으로 문자열 라이브러리에서 `startswith()` 메쏘드로 동일하게 수행할 수 있다. 하지만, 무엇이 정규 표현식을 매칭하는가에 대해서 좀더 많은 제어를 할 수 있게 하는 특수 액션 문자를 담고 있는 정규 표현식의 개념을 소개하기에는 충분하다.

1.1 정규 표현식의 문자 매칭

좀더 강력한 정규 표현식을 작성할 수 있는 다른 특수문자 많이 있다. 가장 자주 사용되는 특수 문자는 임의의 문자를 매칭하는 마침표다.

다음 예제에서 정규 표현식 "F..m:"은 "From:", "Fxxm:", "F12m:", "F!@m:' 같은 임의의 문자열을 매칭한다. 왜냐하면 정규 표현식의 마침표 문자가 임의의 문자와 매칭되기 때문이다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line) :
        print line
```

정규 표현식에 "*", "+' ' 문자를 사용하여 문자가 원하는만큼 반복을 나타내는 기능과 결합되었을 때, 더욱 강력해진다. "*", "+' ' 특수 문자는 검색 문자열에 하나의 문자만을 매칭하는 대신에 별표 기호인 경우 0 혹은 그 이상의 매칭, 더하기 기호인 경우 1 혹은 그 이상의 문자의 매칭을 의미한다.

다음 예제에서 반복 **와일드 카드(wild card)** 문자를 사용하여 매칭하는 라인을 좀더 좁힐 수 있다.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
```

```
    if re.search('^From:.+@', line) :
        print line
```

검색 문자열 "^From:.+@" 은 "From:" 으로 시작하고, ".+" @ 기호로 되는 하나 혹은 그 이상의 문자들의 라인을 성공적으로 매칭한다. 그래서 다음 라인은 매칭이 될 것이다.

From: stephen.marquard @uct.ac.za

콜론(:)과 @ 기호 사이의 모든 문자들을 매칭하도록 확장하는 것으로 ".+" 와이드 카드를 간주할 수 있다.

From: .+ @

더하기와 별표 기호를 "밀어내기(pushy)" 문자로 생각하는 것이 좋다. 예를 들어, 다음 문자열은 ".+" 특수문자가 다음에 보여주듯이 밖으로 밀어내는 것 처럼 문자령릐 마지막 @ 기호를 매칭한다.

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

추가로 다른 특수문자를 추가함으로서 별표나 더하기 기호가 너무 "탐욕(greedy)"스럽지 않게 할 수 있다. 와일드 카드 특수문자의 탐욕스러운 기능을 끄는 것에 대해서는 자세한 정보를 참조바란다.

1.2 정규 표현식 사용 데이터 추출

파이썬에서 문자열에서 데이터를 추출하려면, findall() 메쏘드를 사용해서 정규 표현식과 매칭되는 모든 부속 문자열을 추출할 수 있다. 형식에 관계없이 임의의 라인에서 전자우편 주소 같은 문자열을 추출하는 예제를 사용하자. 예를 들어, 다음 각 라인에서 전자우편 주소를 뽑아내고자 한다.

```
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
```

각각의 라인에 대해서 다르게 쪼개고, 슬라이싱하면서 라인의 각각의 형식에 맞는 코드를 작성하고는 싶지 않다. 다음 프로그램은 findall() 메쏘드를 사용하여 전자우편 주소가 있는 라인을 찾아내고 하나 혹은 그 이상의 주소를 뽑아낸다.

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

findall() 메쏘드는 두번째 인수 문자열을 찾아서 전자우편 주소처럼 보이는 모든 문자열을 리스트로 반환한다. 공백이 아닌 문자 (\S)와 매칭되는 두 문자 스퀀스를 사용한다.

프로그램의 출력은 다음과 같다.

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

정규 표현식을 해석하면, 적어도 하나의 공백이 아닌 문자, @과 적어도 하나 이상의 공백이 아닌 문자를 가진 부속 문자열을 찾는다. 또한, "\S+" 특수 문자는 가능한 많이 공백이 아닌 문자를 매칭한다. (정규 표현식에서 **"탐욕(greedy)"** 매칭이라고 부른다.)

정규 표현식은 두번 매칭(csev@umich.edu, cwen@iupui.edu)하지만, 문자열 "@2PM"은 매칭을 하지 않는다. 왜냐하면, @ 기호 앞에 공백이 아닌 문자가 하나도 없기 때문이다. 프로그램의 정규 표현식을 사용해서 파일에 모든 라인을 읽고 다음과 같이 전자우편 주소처럼 보이는 모든 문자열을 출력한다.

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

각 라인을 읽어 들이고, 정규 표현식과 매칭되는 모든 부속 문자열을 추출한다. findall() 메쏘드는 리스트를 반환하기 때문에, 전자우편 처럼 보이는 부속 문자열을 적어도 하나 찾아서 출력하기 위해서 반환 리스트 요소 숫자가 0 보다 큰가를 만을 간단히 확인한다.

mbox.txt 파일에 프로그램을 실행하면, 다음 출력을 얻는다.

```
['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;']
```

전자우편 수조 몇몇은 "<", ";" 같은 잘못된 문자가 앞과 뒤에 붙어있다. 문자나 숫자로 시작하고 끝나는 문자열 부분만 관심있다고 하자.

그러기 위해서, 정규 표현식의 또 다른 기능을 사용한다. 매칭하려는 다중 허용 문자 집합을 표기하기 위해서 꺾쇠 괄호를 사용한다. 그런 의미에서 "\S"은 공백이 아닌 문자 집합을 매칭하게 한다. 이제 매칭하려는 문자에 관해서 좀더 명확해졌다.

여기 새로운 정규 표현식이 있다.

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

약간 복잡해졌다. 왜 정규 표현식이 자신만의 언어인가에 대해서 이해할 수 있다. 이 정규 표현식을 해석하면, 0 혹은 그 이상의 공백이 아닌 문자("\S*")로 하나의 소문자, 대문자 혹은 숫자("[a-zA-Z0-9]")를 가지며, @ 다음에 0 혹은 그

이상의 공백이 아닌 문자(“\S*”)로 하나의 소문자, 대문자 혹은 숫자(“[a-zA-Z0-9]”)로 된 부속 문자열을 찾는다. 0 혹은 그 이상의 공백이 아닌 문자를 나타내기 위해서 “+”에서 “*”으로 바꿨다. 왜냐하면 “[a-zA-Z0-9]” 자체가 이미 하나의 공백이 아닌 문자이기 때문이다. “*”, “+”는 단일 문자에 별표, 더하기 기호 왼편에 즉시 적용됨을 기억하세요.

프로그램에 정규 표현식을 사용하면, 데이터가 훨씬 깔끔해진다.

```
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

...
['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” 라인에서 문자열 끝에 “>;” 문자를 정규 표현식으로 제거한 것을 주목하세요. 정규 표현식 끝에 “[a-zA-Z]”을 추가하여서 정규 표현식 파서가 찾는 임의의 문자열은 문자로만 끝나야 되기 때문이다. 그래서, “sakaiproject.org>;”에서 “>”을 봤을 때, ”g”가 마지막 맞는 매칭이 되고, 거기서 마지막 매칭을 마치고 중단한다.

프로그램의 출력은 리스트의 단일 요소를 가진 문자열로 파이썬 리스트이다.

1.3 Combining searching and extracting

If we want to find numbers on lines that start with the string “X-” such as:

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

We don’t just want any floating point numbers from any lines. We only to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

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

Translating this, we are saying, we want lines that start with “X-” followed by zero or more characters “.*” followed by a colon (“:”) and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period

"[0-9.]+". Note that in between the square braces, the period matches an actual period (i.e. it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print line
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

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

But now we have to solve the problem of extracting the numbers using `split`. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression they are ignored when matching the string, but when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
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
```

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating point number to indicate we only want `findall()` to give us back the floating point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

The numbers are still in a list and need to be converted from strings to floating point but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

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

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
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
```

Translating our regular expression, we are looking for lines that start with "Details:', followed by any number of characters ".*" followed by "rev=" and then by one or more digits. We want lines that match the entire expression but we only want to extract the integer number at the end of the line so we surround "[0-9]+" with parentheses.

When we run the program, we get the following output:

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

Remember that the "[0-9]+" is "greedy" and it tries to make as large a string of digits as possible before extracting those digits. This "greedy" behavior is why we get all five digits for each number. The regular expression library expands in both directions until it counters a non-digit, the beginning, or the end of a line.

Now we can use regular expressions to re-do an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

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

And wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big

try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this far simpler with the following regular expression:

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

The translation of this regular expression is that we are looking for lines that start with "From " (note the space) followed by any number of characters ".*" followed by a space followed by two digits "[0-9][0-9]" followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

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

This results in the following program:

```
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
```

When the program runs, it produces the following output:

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

1.4 Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are "normal" and we want to match the actual character such as a dollar-sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar-sign with a backslash, it actually matches the dollar-sign in the input string instead of matching the "end of line" and the rest of the regular expression matches one or more digits or the period character. *Note:* In between square brackets, characters are not "special". So when we say "[0-9.]", it really means digits or a period. Outside of square brackets, a period is the "wild-card" character and matches any character. In between square brackets, the period is a period.

1.5 Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings that have special characters in them that communicate your wishes to the regular expression system as to what defines "matching" and what is extracted from the matched strings. Here are some of those special characters and character sequences:

^
Matches the beginning of the line.

$
Matches the end of the line.

.
Matches any character (a wildcard).

\s
Matches a whitespace character.

\S
Matches a non-whitespace character (opposite of \s).

*
Applies to the immediately preceding character and indicates to match zero or more of the preceding character.

*?
Applies to the immediately preceding character and indicates to match zero or more of the preceding character in "non-greedy mode".

+
Applies to the immediately preceding character and indicates to match zero or more of the preceding character.

+?
Applies to the immediately preceding character and indicates to match zero or more of the preceding character in "non-greedy mode".

[aeiou]
Matches a single character as long as that character is in the specified set. In this example, it would match “a”, “e”, “i”, “o” or “u” but no other characters.

[a-z0-9]
You can specify ranges of characters using the minus sign. This example is a single character that must be a lower case letter or a digit.

[ˆA-Za-z]
When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an upper or lower case character.

( )
When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

\b
Matches the empty string, but only at the start or end of a word.

\B
Matches the empty string, but not at the start or end of a word.

\d
Matches any decimal digit; equivalent to the set [0-9].

\D
Matches any non-digit character; equivalent to the set [ˆ0-9].

1.6 Bonus section for Unix users

Support for searching files using regular expressions was built into the Unix operating system since the 1960’s and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into Unix called **grep** (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command line window.

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

This tells `grep` to show you lines that start with the string “From:” in the file `mbox-short.txt`. If you experiment with the `grep` command a bit and read the

documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character "\S" so you will need to use the slightly more complex set notation "[^]"- which simply means - match a character that is anything other than a space.

1.7 Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the methods in the module as follows:

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

You can also get a small amount of documentation on a particular method using the dir command.

```
>>> 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.
>>>
```

The built in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

1.8 Glossary

brittle code: Code that works when the input data is in a particular format but prone to breakage if there is some deviation from the correct format. We call this "brittle code" because it is easily broken.

greedy matching: The notion that the "+" and "*" characters in a regular expression expand outward to match the largest possible string.

grep: A command available in most Unix systems that searches through text files looking for lines that match regular expressions. The command name stands for "Generalized Regular Expression Parser".

regular expression: A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.

wild card: A special character that matches any character. In regular expressions the wild card character is the period character.

1.9 Exercises

Exercise 1.1 Write a simple program to simulate the operation of the `grep` command on Unix. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Exercise 1.2 Write a program to look for lines of the form

```
New Revision: 39772
```

And extract the number from each of the lines using a regular expression and the `findall()` method. Compute the average of the numbers and print out the average.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```