Application of Python

Introduction to Python

Wang Tianyu[1]

Fall, 2018

Outline

What is regular expression?

A regular expression, regex or regexp, is a sequence of characters that define a search pattern.

Usually this pattern is used by **string searching algorithms** for "find" or "find and replace" operations on strings, or for **input validation**. It is a technique that developed in theoretical computer science and formal language theory.

Why regular expression?

1. *Search engines*
2. *Text editors*
3. *Lexical analysis*

Search engines: need to quickly figure out the results matching the user-specified pattern.

Text editors: need to take care of the user-input text and maintain the style (such as cursor).

Lexical analysis: need to convert a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).

Pattern

An expression used to specify a set of strings required for a particular purpose.

A simple way to specify a finite set of strings is to list its elements or members. However, there are often more concise ways to specify the desired set of strings.

Note: "pattern" could be the alias of "regular expression", but the former is more concrete and specific in terms of functionality than the latter.

Formalism

If there exists at least one regular expression that matches a particular set then there exists an infinite number of other regular expression that also match it.

Most formalisms provide the following operations to construct regular expressions:

- **Boolean "or"**: A vertical bar separates alternatives. For example, "hay|hey" could match "hay" or "hey".
- **Grouping**: Parentheses are used to define the scope and precedence of the operators among other uses. For example, "h(a|e)y" could match "hay" or "hey", which is equivalent to "hay|hey".
- **Quantification**: Details later.
- **Wildcard**: The wildcard "." matches any character.

Quantification: A quantifier after a token or group specifies how often that a preceding element is allowed to occur.

- "?" indicates *zero or one* occurrences of the preceding element. For example, "tak?e" matches "take" and "tae".
- "$\star$" indicates *zero or more* occurrences of the preceding element. For example, "hap$\star$y" matches "hay", "hapy", "happy", "happpy", ...
- "+" indicates *one or more* occurrences of the preceding element. For example, "hap+y" matches "hapy", "happy", "happpy", ...
- "{n}" indicates the preceding element is matched exactly *n* times. For example, "hap{2}y" matches only "happy".
- "$\wedge$" indicates matching the element at the beginning of the string. For example, "$\wedge$www" matches those beginning with "www".
- "$" indicates matching the element at the end of the string. For example, "com$".

More on quantification ...

- "[ ]" matches one of the characters listed inside the bracket. For example, "h[ae]y" matches "hay" and "hey".
- "[∧]" matches any character other than the characters listed inside the bracket. For example, "h[∧ae]y" matches "hqy", "hwy", ..., but except for "hay" and "hey".
- "[0-9]" matches any number, which is equivalent to "[0123456789]".
- "[a-z]" matches any lower-case letter, while "[A-Z]" matches any upper-case letter.
- "\d" is equivalent to "[0-9]", while "\D" is equivalent to "[∧0-9]".
- "\s" matches any blank character, like '\t' and '\n', while "\S" vice versa.
- "\w" is equivalent to "[A-Za-z0-9_]", while "\W" vice versa.

Some small exercises ...

1. How to match the string that is composed of digit, letter and underline?

   Solution: $\wedge\backslash$w+$

2. How to check the validity of 18-digit personal ID card?

   Solution: $\wedge$[1-9]$\backslash$d{5}[1-9]$\backslash$d{3}((0$\backslash$d)|(1[0-2]))(([0|1|2]$\backslash$d)|3[0-1])$\backslash$d{3}([0-9]|X)$

3. How to match the set of binary numbers that are multiples of 3?

   Solution: $(0|(1|(01\star 0)\star 1))\star$

re.search

re.search(pattern, string, flags=0)

Parameters:

1. *pattern*: the pattern specified for matching
2. *string*: the original string to be matched with
3. *flags*: the indicator of the matching way

Return: On success, return a matched object; None otherwise.

flags:

- re.I: make the matching case-sensitive free
- re.M: render multi-line matching, which would affect $\wedge$ and \$
- re.X: ignore the comments after \$, which would add to readability
- ...

Examples...

```
import re
search_obj = re.search(r'www', 'www.baidu.com')
print(search_obj.span()) # (0,3)
my_string = "I am Happy Now and Then"
nongreedy_obj =
       re.search(r'(.*) am (.*?) .*', my_string, re.I)
# I am Happy Now and Then
print(nongreedy_obj.group())
# I
print(nongreedy_obj.group(1))
# Happy
print(nongreedy_obj.group(2))
greedy_obj =
       re.search(r'(.*) am (.*) .*', my_string, re.I)
# Happy Now and
print(greedy_obj.group(2))
```

re.sub

re.sub(pattern, repl, string, count=0)

Parameters:

1. *pattern*: the pattern specified for matching
2. *repl*: the **string** or **function** to do substitution
3. *string*: the original string to be matched with
4. *count*: the maximum number of substitution on matching, with 0 on default indicating all substitution

Return: The string after substitution.

Examples...

```python
import re
work_string = "66-66-66 # you are strong"
de_commenting = re.sub(r'#.*$', "", work_string)
number = re.sub(r'\D', "", work_string)
# 66-66-66
print(de_commenting)
# 666666
print(number)
```

re.compile

re.compile(pattern[, flags])

Parameters:

1. *pattern*: the pattern specified for matching
2. *flags*: optional, the indicator of the matching way

Return: The regular expression after compilation

Comment: This is usually used along with *search*, *sub*, ...

Examples...

```
import re
pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)
result = pattern.match("I Am Happy Now")
# I
print(result.group(1))
# (0, 1)
print(result.span(1))
```

re.split

re.split(pattern, string[, maxsplit=0, flags=0])

Parameters:

1. *pattern*: the pattern specified for matching
2. *string*: the original string to be matched with
3. *maxsplit*: optional, indicating the maximum number of splitting
4. *flags*: optional

Return: A list containing each part split. In particular, if not matched, there will be the list containing the single original string.

Examples...

```python
import re
# ['apple', 'banana', 'orange']
print(re.split(r'\W+', "apple, banana, orange"))
# ['baidu.com']
print(re.split(r'^www', "baidu.com"))
# ['apple', 'banana, orange']
print(re.split(r'\W+', "apple, banana, orange"), 1)
```

Internet Worm

A program that can automatically grab the valuable information from a specific platform (e.g., website)

The general architecture could be summarized as:

1. **Dispatcher**: coordinate the whole job, just like a CPU in a computer.
2. **URL manager**: manage those URLs crawled and to be crawled, and prevent repeated and circular crawling.
3. **Web page downloader**: download a specific web page via URL and transform it to a string.
4. **Web page parser**: parse the string and extract desired information.
5. **Application**: data-driven entity.

Web page downloader

```python
# urllib2 could not be used in Python 3
import urllib.request
url = "http://www.baidu.com"
# <http.client.HTTPResponse object at 0x104d00fd0>
response = urllib.request.urlopen(url)
# 200 indicates success
print(response.getcode())
# return a HTML string
print(response.read())
```

Comment: when we try to download a web page with URL specified as "https", like "`https://github.com`", we fail. The reason is that we locally lack the certificate to access it.

A somewhat dirty solution ...

```python
import ssl
import urllib.request
# manually create a fake certificate context
context = ssl._create_unverified_context()
url = "https://www.github.com"
urllib.request.urlopen(url, context=context)
```

You can search by yourself for cleaner solution.

Web page parser

– A user-friendly tool: *Beautiful Soup* (official website: `https://www.crummy.com/software/BeautifulSoup/`)

– To install: `pip3 install beautifulsoup4`

– To parse a HTML web page:

```
# if install does not work, can try this
# import pip
# pip.main(["install","bs4"])
import re
import urllib.request
from bs4 import BeautifulSoup
url = "http://www.baidu.com"
html_doc = urllib.request.urlopen(url).read()
soup = BeautifulSoup(html_doc, "html.parser",
                                  from_encoding="utf-8")
link_node = soup.find('a', href=re.compile(r'news'))
print(link_node.name)
print(link_node.get_text())
```

What is .csv file?

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases.

While the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

Elements:

- *Header*: specifies the column names
- *Row*: each row corresponds to a record
- *Column*: each column corresponds to a feature

Table: grade.csv

Student No.	Name	Grade
0	Peter	89
1	John	93
2	Ben	61
3	Alice	95

Read:

```
import csv
with open("grade.csv", "r") as in_csv:
    # read csv file
    # return a list containing each row
    input = csv.reader(in_csv)
    # get the header list
    header = next(input)
    # get index
    grade_index = header.index("Grade")
    for row in input:
        print(row)
        print(row[grade_index])
```

Write:

```python
import csv
with open("grade_new.csv", "w") as out_csv:
    output = csv.writer(out_csv)
    output.writerow(["4", "Hill", "78"])
```

Store list:

```
my_list = [1, 2, 3]
file_out = open('list_store.txt', 'w')
for item in my_list:
    file_out.write(str(item))
    file_out.write('\n')
file_out.close()
```

Read list:

```
f = open('list_store.txt','r')
in_list = f.read()
f.close()
```

Store dictionary:

```
my_dict = {1:{1:2,3:4},2:{3:4,4:5}}
f = open('dict_store.txt','w')
f.write(str(my_dict))
f.close()
```

Read dictionary:

```
f = open('dict_store.txt','r')
a = f.read()
my_dict = eval(a)
f.close()
```

More interesting libraries ...

- `Sklearn`: including the API of those classic algorithms widely used in Machine Learning.
- `OpenCV`: including the API of those classic algorithms in Computer Vision, Image Processing, ...
- `MongoDB`: including the API of a distributive file storage database.
- `Matplotlib`: supporting a variety of techniques for plotting (visualization).
- `Pandas`: supporting a lot of techniques for data analysis and statistics.

Thanks for your listening!

Follow us on github: `https://github.com/gunnerwang/UMJI-Career-and-Learning-Advising-Center`

Wechat: `UMJIAdvisingCenter`