L4-Python-and-Texts

2015 年 10 月 23 日

1 Regular Expression

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

1.1 Regular Expression Syntax

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'.

Some characters, like '|' or '(', are special. **Special characters** either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

- `[]`: Used to indicate a set of characters. In a set:
  - Characters can be listed individually, e.g. [amk] will match 'a', 'm', or 'k'.
  - Ranges of characters can be indicated by giving two characters and separating them by a '-', for example [a-z] will match any lowercase ASCII letter, [0-5][0-9] will match all the two-digits numbers from 00 to 59, and [0-9A-Fa-f] will match any hexadecimal digit. If - is escaped (e.g. [a-z]) or if it's placed as the first or last character (e.g. [a-]), it will match a literal '-'.
- `(...)`: Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals '(' or ')', use (or), or enclose them inside a character class: [(] [)].
- '`|`': A|B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the'|'in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by'|'are tried from left

to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the'|'operator is never greedy. To match a literal'|', use |, or enclose it inside a character class, as in [|].

The special sequences consist of '' and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, $ matches the character'$'

- `\d`: Matches any decimal digit; this is equivalent to the class `[0-9]`.
- `\D`: Matches any non-digit character; this is equivalent to the class `[^0-9]`.
- `\s`: Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.
- `\S`: Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.
- `\w`: Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.
- `\W`: Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For example, `[\s,.]` is a character class that will match any whitespace character, or ',' or '.'.

The final metacharacter in this section is .. It matches anything except a newline character, and there's an alternate mode (re.DOTALL) where it will match even a newline. '.' is often used where you want to match "any character".

For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax in the Standard Library reference.

1.2 Python `re` module

This module provides regular expression matching operations similar to those found in Perl.

1.2.1 Compiling Regular Expressions

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form. Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()` and `search()` methods

```
re.compile(pattern, flags=0)
```

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
In [42]: p = re.compile('o')
         p

Out[42]: re.compile(r'o', re.UNICODE)

In [43]: print(p.search('I love Python'))

<_sre.SRE_Match object; span=(3, 4), match='o'>
```

1.2.2 Backslash character (”)

Regular expressions use the backslash character (‘’) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write’\\’ as the pattern string, because the regular expression must be \, and each backslash must be expressed as \ inside a regular Python string literal.

```
In [1]: print('\\\\')
```

```
\\
```

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with ‘r’. So `r"\n"` is a two-character string containing ‘’ and ’n’, while `\n` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

```
In [2]: print('\n') # print a new line
```

```

```

```
In [3]: print(r'\n') # print '\n' string
```

```
\n
```

1.3 Matching Characters

1.4 `re.match()` and `re.search()`

Python offers two different primitive operations based on regular expressions: `re.match()` checks for a match only at the beginning of the string, while `re.search()` checks for a match anywhere in the string (this is what Perl does by default).

```
In [9]: import re
        out1 = re.match('c', "I love coding")
        out2 = re.search('c', "I love coding")

        print(out1)
        print(out2)
```

```
None
<_sre.SRE_Match object; span=(7, 8), match='c'>
```

Regular expressions beginning with ‘^’ can be used with `search()` to restrict the match at the beginning of the string:

```
In [12]: print(re.match("c", "abcdef"))  # No match
         print(re.search("^c", "abcdef")) # No match
         print(re.search("^a", "abcdef"))  # Match
```

```
None
None
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

In `MULTILINE` mode `match()` only matches at the beginning of the string, whereas using `search()` with a regular expression beginning with '^' will match at the beginning of each line.

```
In [17]: print(re.match('X', 'A\nB\nX', re.MULTILINE))  # No match
         print(re.search('^X', 'A\nB\nX', re.MULTILINE))  # Match
```

```
None
<_sre.SRE_Match object; span=(4, 5), match='X'>
```

1.5 match.group([group1, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If a groupN argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range `[1..99]`, it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is None. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
In [68]: m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
         print(m.group(0))       # The entire match

         print(m.group(1))       # The first parenthesized subgroup.

         print(m.group(2))       # The second parenthesized subgroup.

         print(m.group(1, 2))    # Multiple arguments give us a tuple.
```

```
Isaac Newton
Isaac
Newton
('Isaac', 'Newton')
```

1.6 match.start([group]) and match.end([group])

Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring). Return -1 if group exists but did not contribute to the match. For a match object m, and a group g that did contribute to the match, the substring matched by group g (equivalent to m.group(g)) is

```
In [46]: email = "tony@tiremove_thisger.net"
         m = re.search("remove_this", email)
         email[:m.start()] + email[m.end():]

Out[46]: 'tony@tiger.net'
```

1.7 Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; string `split()` only supports splitting by whitespace or by a fixed string.

```
In [44]: re.split('\W+', 'Words, words, words.')

Out[44]: ['Words', 'words', 'words', '']

In [47]: re.split('(\W+)', 'Words, words, words.')

Out[47]: ['Words', ', ', 'words', ', ', 'words', '.', '']

In [48]: re.split('\W+', 'Words, words, words.', 1)

Out[48]: ['Words', 'words, words.']

In [49]: re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)

Out[49]: ['0', '3', '9']
```

1.8 Substitution

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

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern isn't found, string is returned unchanged. repl can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\j` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
In [66]: re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
                r'static PyObject*\npy_\1(void)\n{',
                'def myfunc():')

Out[66]: 'static PyObject*\npy_myfunc(void)\n{'
```

2 Classes

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. Python classes provide all the standard features of Object Oriented Programming:

- the class inheritance mechanism allows multiple base classes,
- a derived class can override any methods of its base class or classes, and
- a method can call the method of a base class with the same name.

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods.

When a class definition is entered, a new **namespace** is created, and used as the *local scope* — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

Often, the first argument of a method is called self. This is nothing more than a convention: the name self has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a class browser program might be written that relies upon such a convention.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: *they are created at runtime, and can be modified further after creation.*

- Python tutorial: Classes

2.1 Class Objects

Class objects support two kinds of operations: **attribute references** and **instantiation**.

```
In [50]: class MyClass:
             """A simple example class"""
             i = 12345
             def f(self):
                 return 'hello world'
```

```
In [51]: MyClass()
```

```
Out[51]: <__main__.MyClass at 0x7f195922e898>
```

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created.

```
In [52]: MyClass.i
```

```
Out[52]: 12345
```

```
In [53]: MyClass.f
```

```
Out[53]: <function __main__.MyClass.f>
```

`MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment.

```
In [55]: MyClass.i = 3
```

```
In [56]: MyClass.i
```

```
Out[56]: 3
```

The **instantiation operation** ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
In [64]: def __init__(self):
             self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by calling

```
In [63]: x = MyClass()
         print(x)
```

```
<__main__.MyClass object at 0x7f19591b4da0>
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`.

```
In [59]: class Complex:
             def __init__(self, realpart, imagpart):
                 self.r = realpart
                 self.i = imagpart


         x = Complex(3.0, -4.5)
         x.r, x.i
```

```
Out[59]: (3.0, -4.5)
```

3 Web Scraping

One of the challenges of writing web crawlers is that you're often performing the same tasks again and again: find all links on a page, evaluate the difference between internal and external links, go to new pages. These basic patterns are useful to know about and to be able to write from scratch, but there are options if you want something else to handle the details for you.

Scrapy is a Python library that handles much of the complexity of finding and evaluating links on a website, crawling domains or lists of domains with ease. Unfortunately, Scrapy has not yet been released for Python 3.x, though it is compatible with Python 2.7.

3.1 Create a Scrapy project

Although writing Scrapy crawlers is relatively easy, there is a small amount of setup that needs to be done for each crawler. To create a new Scrapy project in the current directory, run from the command line:

```
scrapy startproject wikiSpider
```

In order to create a crawler, we will add a new file to `wikiSpider/wikiSpider/spiders/` called `items.py` . In addition, we will define a new item called `Article` inside the `items.py` file.

Your `items.py` file should be edited to look like this (with Scrapy-generated comments left in place, although you can feel free to remove them):

```
# -*- coding: utf-8 -*-
# Define here the models for your scraped items
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html
from scrapy import Item, Field

class Article(Item):
        # define the fields for your item here like:
        # name = scrapy.)
        title = Field()
```

Now you create a file `wikiSpider/wikiSpider/spiders/articleSpider.py`. In your newly created `articleSpider.py` file, write the following

```
# -*- coding: utf-8 -*-
from scrapy.selector import Selector
from scrapy import Spider
from wikiSpider.items import Article
class ArticleSpider(Spider):
    name="article"
    allowed_domains = ["en.wikipedia.org"]
    start_urls = ["http://en.wikipedia.org/wiki/Main_Page",
                  "http://en.wikipedia.org/wiki/Python_%28programming_language%29"]
```

```
    def parse(self, response):
        item = Article()
        title = response.xpath('//h1/text()')[0].extract()

        item['title'] = title
        yield item
```

Then go to your `wikiSpider` project home directory and run the project

```
scrapy crawl article -o wiki.csv
```

The information is now saved in `wiki.csv` file.

3.2 Retrieve a table from web

Still the `Python Programming Language` topic on Wikipedia, we'd like to extract the talbe entitled "Summary of Python 3's built-in types". First, let's create a new project called `wikiPythonTable`

```
scrapy startproject wikiPythonTable
```

And we add more items to the `items.py` file under `wikiPythonTable/wikiPythonTable/`

```
# -*- coding: utf-8 -*-
# Define here the models for your scraped items
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html
from scrapy import Item, Field

class Article(Item):
        # define the fields for your item here like:
        # name = scrapy.)
        datatype = Field()
        mutable = Field()
        description = Field()
        syntax = Field()
```

We now need a `tableSpider.py` under `wikiPythonTable/wikiPythonTable/spider`

```
# -*- coding: utf-8 -*-
from scrapy.selector import Selector
from scrapy import Spider
from wikiPythonTable.items import Article
class WikiPythonTable(Spider):
    name="table"
    allowed_domains = ["en.wikipedia.org"]
```

```
    start_urls = ["http://en.wikipedia.org/wiki/Python_%28programming_language%29"]
    def parse(self, response):
        item = Article()

        table_path = response.xpath('//table[re:test(@class,"wikitable")]/tr')
        for i in range(1, len(table_path)):
            item['datatype'] = table_path[i].xpath('.//td[1]/code/text()').extract()[0]
           item['mutable'] = table_path[i].xpath('.//td[2]/descendant::text()').extract()
        item['description'] = ''.join(table_path[i].xpath('.//td[3]/descendant::text()').extract())
        item['syntax'] = ' '.join(table_path[i].xpath('.//td[4]/descendant::text()').extract())
            yield item
```

Run your project with at the project root directory

```
scrapy crawl table -o table.csv  --logfile table.log
```

3.3 Scrapy Shell

To do the crawler interactively, just run

```
scrapy shell "http://en.wikipedia.org/wiki/Python_%28programming_language%29"
```