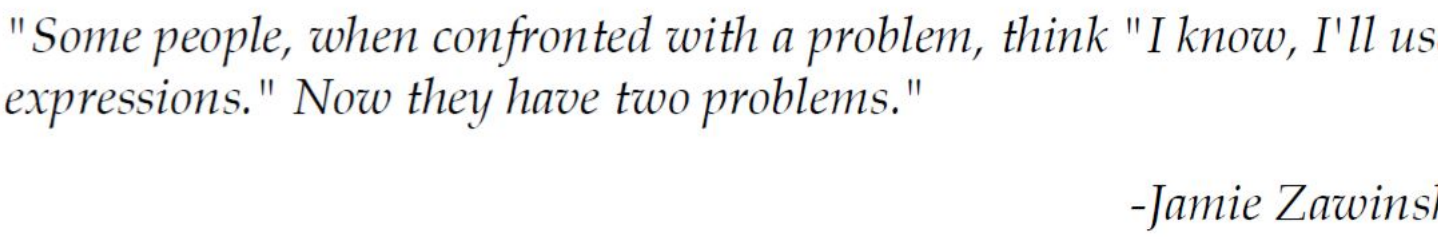
Beginners Tutorial for Regular Expression in Python

INTERMEDIATE PYTHON REGEX

This article was published as a part of the Data Science Blogathon.

Introduction

Like every other person, I've faced quite some difficulties in using a regular expressions, and I am sure still there is a lot to learn. But, I've reached a point where I can use them in my day-to-day work. In my process of learning regular expression, I came across a saying which I feel isn't 100% true in my case now.



So I am writing this article that serves as a beginner's guide for learning Regular expressions in the Python programming language. This article illustrates the importance and the use cases of regular expressions. And by the end, you'll be able to use them efficiently.

What is a Regular Expression?

Regular Expression (short name RegEx) is nothing but a sequence of characters that specifies a search pattern to extract or replace a part of the text.

Some of the Applications of Regular Expression Are:

1. Pre-process the text data as part of textual analysis tasks
2. Define validation rules to validate usernames/email ids, phone numbers, etc.
3. Parsing data in web scraping or data extraction for Machine learning tasks

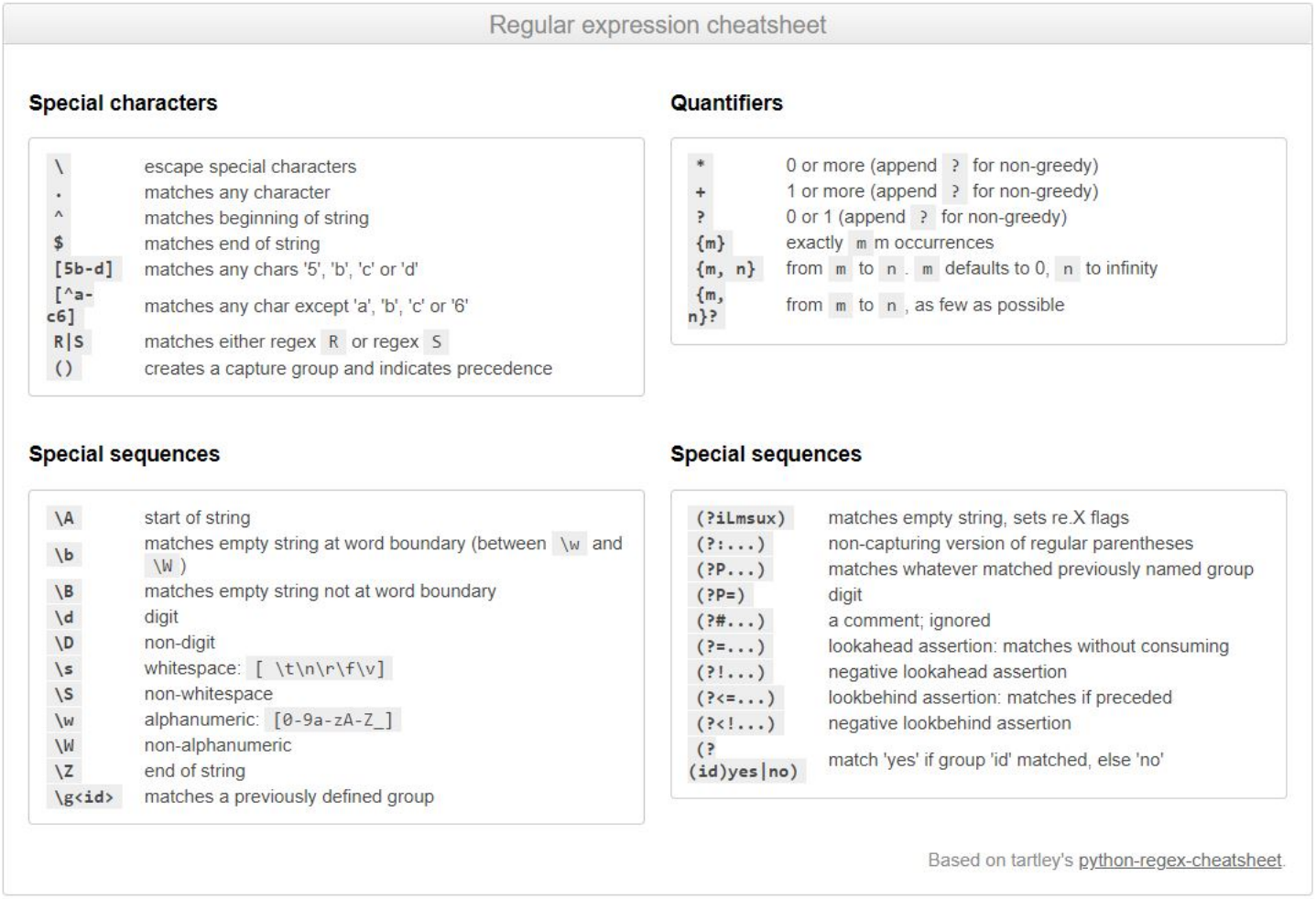
Python Module for Regular Expression

The Python module that supports Regular Expression (RegEx) is **re**. The **re** module comes with Python installation, so we need not install it separately.

Here is a link to the documentation of the Python re module.

Regular Expression Cheatsheet

The cheat sheet attached here is from pythex.org(a quick Regular Expression editor and tester for Python language). Take your time and go through it thoroughly before moving forward to the next section.



Regular expression cheatsheet

Special characters

`\`	escape special characters
`.`	matches any character
`^`	matches beginning of string
`$`	matches end of string
`[5b-d]`	matches any chars '5', 'b', 'c' or 'd'
`[^a-c6]`	matches any char except 'a', 'b', 'c' or '6'
`R|S`	matches either regex `R` or regex `S`
`()`	creates a capture group and indicates precedence

Quantifiers

`*`	0 or more (append `?` for non-greedy)
`+`	1 or more (append `?` for non-greedy)
`?`	0 or 1 (append `?` for non-greedy)
`{m}`	exactly `m` m occurrences
`{m, n}`	from `m` to `n`. `m` defaults to 0, `n` to infinity
`{m, n}?`	from `m` to `n`, as few as possible

Special sequences

`\A`	start of string
`\b`	matches empty string at word boundary (between `\w` and `\W`)
`\B`	matches empty string not at word boundary
`\d`	digit
`\D`	non-digit
`\s`	whitespace: `[ \t\n\r\f\v]`
`\S`	non-whitespace
`\w`	alphanumeric: `[0-9a-zA-Z_]`
`\W`	non-alphanumeric
`\Z`	end of string
`\g<id>`	matches a previously defined group

Special sequences

`(?iLmsux)`	matches empty string, sets re.X flags
`(?:...)`	non-capturing version of regular parentheses
`(?P...)`	matches whatever matched previously named group
`(?P=)`	digit
`(?#...)`	a comment; ignored
`(?=...)`	lookahead assertion: matches without consuming
`(?!...)`	negative lookahead assertion
`(?<=...)`	lookbehind assertion: matches if preceded
`(?<!...)`	negative lookbehind assertion
`(?(id)yes|no)`	match 'yes' if group 'id' matched, else 'no'

Based on tartley's python-regex-cheatsheet.

Basic Operations and Functions

The three basic operations that we can perform using regular expressions are as below:

1. Search – to see if the given pattern is present in the string
2. Match – to see if the input string matches the given pattern
3. Transform – to substitute/replace the matched text.

1. Search

The search operation is similar to finding the required text in a word document or a web page. Returns the match object in case search is a success, and None otherwise.

search(), findall() can be used to perform search operations. search() method stops with the first occurrence whereas findall() returns all the occurrences as a list.

2. Match

The match operation is similar to the search, but it starts to search from the beginning of the text, whereas the search operation searches the whole string even if the substring is present in the middle of the input string.

Follow the below steps to perform match or search operations:

1. Import re module and define the regex ***pattern***
2. Create a ***regex object*** using *re.compile()* method. Do not forget to pass the pattern as a raw string
   - regex_obj = re.compile(pattern)
3. Call the *search(), findall(),* or *match()* method on the regex object and pass the address string as an argument. The resultant would be a *match object*.
   - match_obj = regex_obj.search(input_string) *or*
   - match_obj = regex_obj.findall(input_string)
   - match_obj = regex_obj.match(input_string)
4. call the respective methods on the ***match object*** to see the output

Example for Search Operation using search(): Check if the word 'Order' is present in the string 'Harry Potter and the Order of the Phoenix.'

```
# 1. import re module and define the regex pattern import re pattern = r'Order' # 2. create a regex object regex_obj = re.compile(pattern, flags=re.I) # 3. call the search() method on the regex object match_obj = regex_obj.search('Harry Potter and the Order of the Phoenix') # 4. call the applicable methods on match_obj print(match_obj.start(), match_obj.end(), match_obj.group(0))
```

Output: 21 26 Order

As we haven't done the grouping yet, we called .group(0) to obtain the matched word. We will learn more about grouping in the use cases section.

We can implement the same without re.compile() by directly calling *re.search()*

Syntax: *match_obj = re.search(pattern, input_string)*

match_obj = re.search(r'Order', 'Harry Potter and the Order of the Phoenix')

```
print(match_obj.start(), match_obj.end(), match_obj.group(0))
```

Output: 21 26 Order

Example of search operation using findall(): obtain all occurrences of 'the' in 'Harry Potter and the Order of the Phoenix'

```
# import re module and define the regex pattern import re pattern = r'the' # create a regex object regex_obj = re.compile(pattern, flags=re.I) # call the findall() method on the regex object match_obj = regex_obj.findall('Harry Potter and the Order of the Phoenix') match_obj
```

Output: ['the', 'the']

findall() without re.compile()

```
match_obj = re.findall('the', 'Harry Potter and the Order of the Phoenix') match_obj
```

Output: ['the', 'the']

Example for Match Operation: Check if the input word of any length starts with a and ends with s

```
import re pattern = r'^aw*s$' regex_obj = re.compile(pattern) match_obj = regex_obj.match('analytics') print(match_obj.start(), match_obj.end(), match_obj.group(0))
```

Output: 0 9 analytics

'^' is to indicate the start, and '$' is to indicate the end.

match() without re.compile()

```
match_obj = re.match(r'^aw*s$', 'analytics') print(match_obj.start(), match_obj.end(), match_obj.group(0))
```

Output: 0 9 analytics

3. Transform

1. replace parts of a string

Manipulation of a string is a basic example of regex transform operation. re.sub() method is used to replace parts of a string.

syntax: re.sub(pattern, replacement, input_string, count, flags)

'flags=re.I' is to make the pattern case insensitive.

Example: Replace all occurrences of 'cat' or 'dog' with 'pet'.

```
string = 'Cats are cute. I play with my dog all time.' replaced_str = re.sub('cat|dog', 'pet', string, flags=re.I) replaced_str
```

Output: pets are cute. I play with my pet all time.

Example: Mask phone number

string = 'My name is Harry, and you can contact me at (805) 588 8745'
masked_str = re.sub('d', '*', string)
masked_str

Output: My name is Harry, and you can contact me at (***) *** ****

Example: Mask only the first six digits

string = 'My name is Harry, and you can contact me at (805) 588 8745'
masked_str = re.sub('d', '*', string, count=6)
masked_str

Output: My name is Harry, and you can contact me at (***) *** 8745

2. Converting string to list

re.split() method splits the string and returns a list

Splitting the string by spaces.

```
string = 'This is a regex tutorial' lst = re.split('s', string) print(last)
```

Output: [‘This’, ‘is’, ‘a’, ‘regex’, ‘tutorial’]

Use Cases of Regular Expressions

Identify the Patterns to Get the Name and Age

The hint here is every word that starts with a capital letter is a name, and the numbers are ages.

```
NameAge = '''Janice is 22 and Kacy is 33 Gabriel is 44 and Joey is 101''' ages = re.findall(r'd{1,3}', NameAge) names = re.findall(r'[A-Z][a-z]*', NameAge) person = {} x = 0 for name in names: person[name] = ages[x] x+=1 print(person)
```

Output: {‘Janice’: ’22’, ‘Kacy’: ’33’, ‘Gabriel’: ’44’, ‘Joey’: ‘101’}

Email Validation

Assuming the username has to be 6 to 30 characters long.

```
emails = ['harika96_02%@gmail.com', 'hari029yahoo.com', 'har+uhs@.COMmm'] for email in emails: if(re.findall("[wW]{6,30}@[w]{2,20}.[A-Z]{2,3}", email, flags=re.I)): print(email, ': valid') else: print(email, ': invalid')
```

Output:

harika96_02%@gmail.com :valid
hari029yahoo.com : invalid
har+uhs@.COMmm : invalid

Obtaining Details From the Address Text

I want to obtain the apartment number, street, City, State, and zip code from a given address string as five groups.

Source: Manifold.net

Above are some of the sample addresses for practice purposes. Let’s create a list of addresses(I picked only a few).

```
addr_lst = [ '555 Wille Stargell Ave., Alameda, CA 94501', '1210 N. Atlantic Blvd., Alhambra, CA 91810', '600 S. Brookhurst, Anaheim, CA 92804', '1075 W. I-20, Arlington, TX 76017' ]
```

1) Import re module and define the address regex pattern

```
import re addr_pattern = r'([0-9]+)s*([a-z.-s0-9]+).?,?s*([a-z]+),?s*([a-z]{2})s*([0-9]+)'
```

Explanation of address pattern:

Given the address structure, the five groups of interest are as below:

1. apartment number (integer of any number of digits): **[0-9]+**
2. street (alphabets, spaces, integers, special characters): **[a-z.-s0-9]+**
3. city (alphabets): **[a-z]+**
4. state (two alphabets long): **[a-z]{2}**
5. zip code (integer of any number of digits): **[0-9]+**

To obtain them as individual groups, we need to wrap them with braces (). And each group is separated either by space or comma or both.

Important Points to Note:

‘**+**‘ in the address pattern indicates one or more occurrences of character set elements. A character set is

denoted by square brackets.
‘*****‘ indicated 0 or more occurrences of the character written before it
‘**?**‘ indicates 0 or 1 occurrence of the character written before it.
Special characters are to be escaped using a backward slash ”

At this point, it is too much to take in as a beginner. So, I would like to introduce a regex visualizer to make it easier for you.

Open this URL and on the side, the panel select the language as Python and check the flags as IGNORECASE

As our regex is long, I am only showing the visualization of a part of our address regex.

Paste in your regex, input string, and scroll down to see the visualization.

2) Create a regex object. Use *flags=re.I* to keep the pattern case insensitive.

Syntax: *regex_obj_name = re.compile(pattern, flags=re.I)*

```
addr_regex_obj = re.compile(addr_pattern, flags=re.I)
```

3 call the match() method on the regex object

Syntax: *output_var = regex_obj_name.match(string)*

4 call the groups() method on the match object to obtain all the groups. This returns a tuple.

syntax: *output_var.groups()*

Code for steps 3, 4:

```
for addr in addr_lst: # call .match() method on regex object match = addr_regex_obj.match(addr) # call .groups() method on the match object print(addr, ' ' , match.groups())
```

Output:

Now, the same address pattern can be written using a shorthand notation. It’s pretty simple; you will have to carefully replace the character sets with the respective character classes by referring to the cheat sheet. The newly obtained address pattern is:

```
addr_pattern = r'(d+)s*([w.-s]+).?,?s*(w+),?s*(w{2})s*(d+)'
```

[0-9] is replaced by d, [0-9a-z] is replaced by w, [a-z] is replaced by w.

And, one interesting thing is, you can name the groups in the regex pattern.

```
addr_pattern_wth_grpnames = (?Pd+)s*(?P[a-z.-sd]+).?,?s*(?P[a-z]+),?s*(?P[a-z]{2})s*(?Pd+)
```

So now, instead of calling the groups() method we can call **group(group_name)** to obtain only a specific group value. Refer to the below code to see how it works. I am printing only the street group.

```
import re addr_pattern_wth_grpnames = r'(?Pd+)s*(?P[a-z.-sd]+).?,?s*(?P[a-z]+),?s*(?P[a-z]{2})s*(?Pd+)' addr_regex_obj = re.compile(addr_pattern_wth_grpnames, flags=re.I) for addr in addr_lst: match = addr_regex_obj.match(addr) print(addr, ' ---> ' , match.group('street'))
```

The complete code for the address matching use case is as below:

```
addr_lst = [ '555 Wille Stargell Ave., Alameda, CA 94501', '1210 N. Atlantic Blvd., Alhambra, CA 91810', '600 S. Brookhurst, Anaheim, CA 92804', '1075 W. I-20, Arlington, TX 76017' ] import re addr_pattern_wth_grpnames = r'(?Pd+)s*(?P[a-z.-sd]+).?,?s*(?P[a-z]+),?s*(?P[a-z]{2})s*(?Pd+)' addr_regex_obj = re.compile(addr_pattern_wth_grpnames, flags=re.I) for addr in addr_lst: match = addr_regex_obj.match(addr) print(addr, ' ---> ' , match.groups(), ' ---> ', match.group('street'))
```

You can also debug, and test the above example in pythex.org by clicking here.

Execute and see the output to get a good understanding.

Conclusion

In this article, we learned:

- The importance of regular expressions and the Python library used for regex
- Writing regex patterns and creating regex objects to perform a search or text-transform operations.
- Testing and visualizing regex patterns online and some sample use cases to be used daily.

References

Fork the complete code file from the GitHub repo.

The media shown in this article is not owned by Analytics Vidhya and is used at the Author's discretion.

Article Url - https://www.analyticsvidhya.com/blog/2022/04/beginners-tutorial-for-regular-expression-in-python/

Harika Bonthu