



TATA ELXSI

Python Scripting

Learning & Development Team

Monday, November 2, 2020

Disclaimer

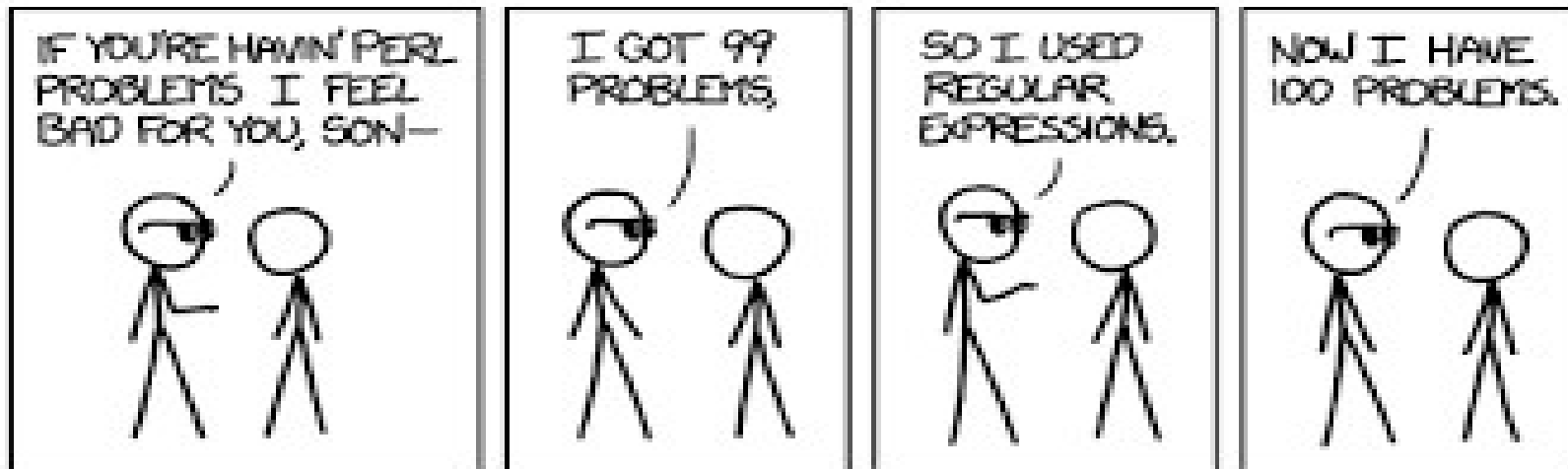
- This material is developed for in house training at **TATA ELXSI**.
- *TATA ELXSI is not responsible for any errors or omissions, or for the results obtained from the use of this information. All information in this material is provided "as is", with no guarantee of completeness, timeliness or of the results obtained from the use of this information...*
- *This is a training material and the information contained herein is not intended to be a source of advice or research outcome with respect to the material presented, and the information and/or documents/examples contained in this material do not constitute authors or TATA ELXSI's advice.*
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage.
- **TATA ELXSI**, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books.
- Excerpts of these material has been taken where there is no copy right infringements.

Module – 7 : Regular Expressions

Regular Expressions

“Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.”

— [Jamie Zawinski](#)



- Regular Expressions are used in programming languages to filter texts or text strings.

Need of Reg-Ex

- In last few years, there has been a dramatic shift in usage of general purpose programming languages for data science and machine learning.
- This means that more people / organizations are using tools like Python / JavaScript for solving their data needs.
- This is where Regular Expressions become very useful.
- Regular expressions are normally the default way of data cleaning and wrangling in most of these tools.



- Regular expressions are used to sift through text-based data to find things.
- Regular expressions express a pattern of data that is to be located.
- Regex is its own language, and is basically the same no matter what programming language you are using with it.
- <http://www.rexegg.com/>

Regexes are used for five main purposes:

- Parsing: identifying and extracting pieces of text that match certain criteria
- Searching: locating substrings that can have more than one form, for example, finding any of “pet.png”, “pet.jpg”, “pet.jpeg”, or “pet.svg” while avoiding “carpet.png” and similar
- Searching and replacing: replacing everywhere the regex matches with a string, for example, finding “bicycle” or “human powered vehicle” and replacing either with “bike”

Regexes are used for five main purposes:

- Splitting strings: splitting a string at each place the regex matches, for example, splitting everywhere colon-space or equals (“: ” or “=”) occurs
- Validation: checking whether a piece of text meets some criteria, for example, contains a currency symbol followed by digits.

Re - module

- In Python 3, the module to use regular expressions is re, and it must be imported to use regular expressions.
- Re is a part of the standard library.
- When writing regular expression in Python, it is recommended that you use raw strings instead of regular Python strings.
- Raw strings begin with a special prefix (r) and signal Python not to interpret backslashes and special metacharacters in the string.

Re module

- The 're' package provides multiple methods to perform queries on an input string. Here are the most commonly used methods:
- `re.match()` : Determine if the RE matches at the beginning of the string.
- `re.search()` : Scan through a string, looking for any location where this RE matches.
- `re.findall()` : Find all substrings where the RE matches, and returns them as a list.
- `re.split()` : Split *string* by the occurrences of *pattern*.
- `re.sub()` : substitutes in the entire substring matched by the RE.
- `re.compile()` : Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods.
- `finditer()` : Find all substrings where the RE matches, and returns them as an iterator.

Regular expression cheat sheet

- <http://regexlib.com/CheatSheet.aspx?AspxAutoDetectCookieSupport=1>
- Identifiers:
 - \d = any number
 - \D = anything but a number
 - \s = space
 - \S = anything but a space
 - \w = any letter
 - \W = anything but a letter
 - . = any character, except for a new line
 - \b = space around whole words
 - \. = period. must use backslash, because . normally means any character.

Modifiers:

- `{1,3}` = for digits, u expect 1-3 counts of digits, or "places"
- `+` = match 1 or more
- `?` = match 0 or 1 repetitions.
- `*` = match 0 or MORE repetitions
- `$` = matches at the end of string
- `^` = matches start of a string
- `|` = matches either/or. Example `x|y` = will match either x or y
- `[]` = range, or "variance"
- `{x}` = expect to see this amount of the preceding code.
- `{x,y}` = expect to see this x-y amounts of the preceding code

White Space Charts:

- `\n` = new line
- `\s` = space
- `\t` = tab
- `\e` = escape
- `\f` = form feed
- `\r` = carriage return

Re.match

- `regex.match(string[, pos[, endpos]])`
- If zero or more characters at the beginning of string match this regular expression, return a corresponding match object.
- Return `None` if the string does not match the pattern; note that this is different from a zero length match.
- The optional `pos` and `endpos` parameters have the same meaning as for the `search()` method.

Re.match

```
>>> text="Jack and Jill went up the hill"  
>>> pattern=re.compile('Jill')  
>>> pattern.match(text)  
>>> pattern.match(text[9:])  
<_sre.SRE_Match object at 0x03B5C4F0>
```

I : IGNORECASE Perform case-insensitive matching.

L : LOCALE Make \w, \W, \b, \B, dependent on the current locale.

M: MULTILINE "^" matches the beginning of lines (after a newline)
as well as the string.
"\$" matches the end of lines as well as the end of the string.

S : DOTALL "." matches any character at all, including the newline.

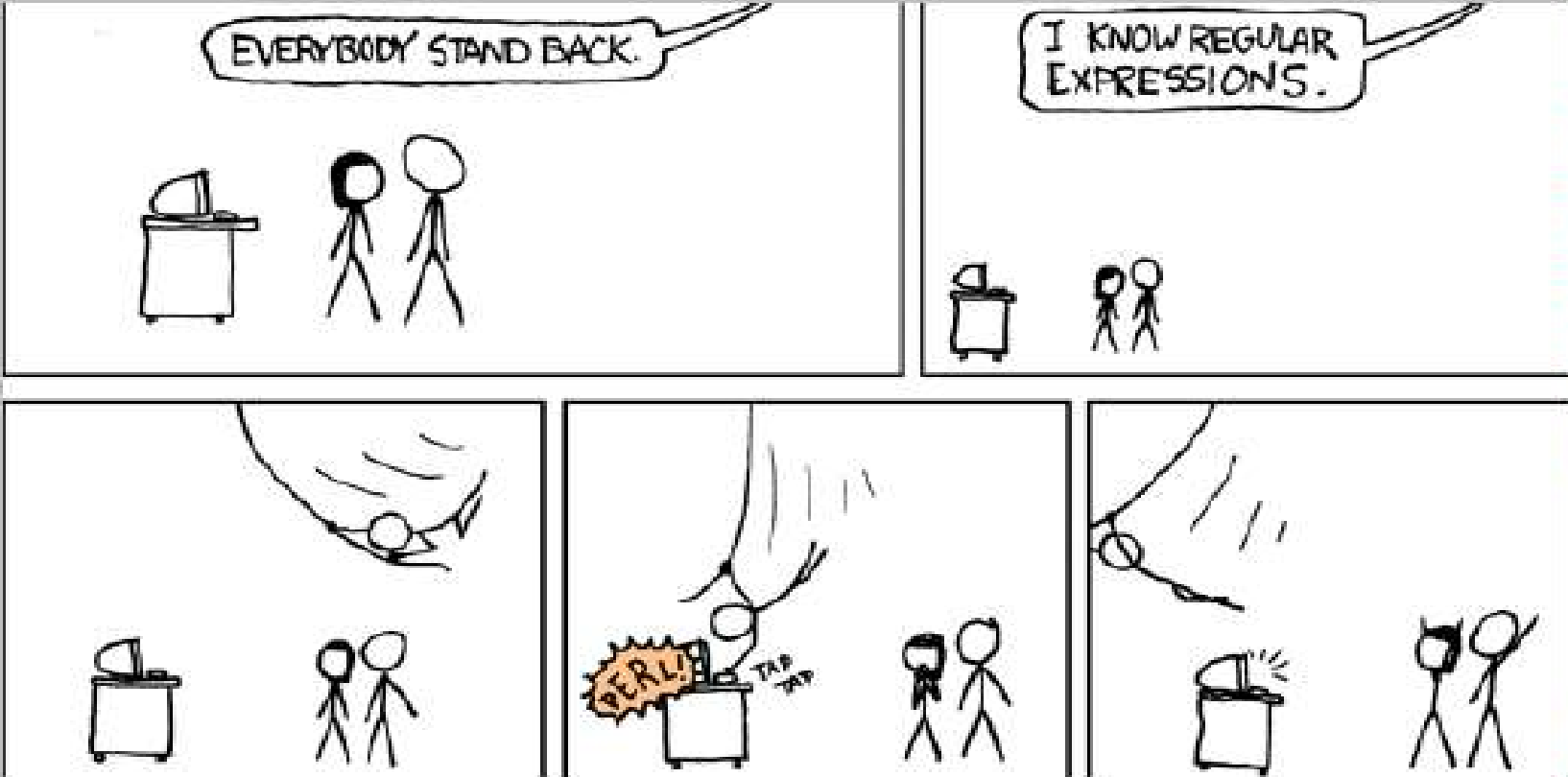
X : VERBOSE Ignore whitespace and comments for nicer looking RE's.

Example 1:

```
import re
text="""
    Mona and Reena are two sisters, of age 12 and 15 respectively.
    lived with their mother Ritha and father Jhon who are 30 and 35.
    """
ages=re.findall(r'\d{1,3}',text)
names=re.findall(r'[A-Z][a-z]+',text)

print(ages)
print(names)
```


Let us expertise regular expressions



EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

```
Yatri@Budha-7 ~/r  
$ grep -E '\<'tom  
tomatoes  
green tomatoes  
tom0  
tom5  
tom9  
tomA  
tomZ  
tomF  
tomy
```

```
Yatri@Budha-7 ~/r  
$ grep -E tom'\>'  
phantom  
phantom of the op  
atom  
atom and his pack
```

```
Yatri@Budha-7 ~/re  
$ grep tom[^F\|0-9  
tomatoes.
```

RegEX:

```
import re
```

```
print(re.split(r'\s*', 'here are some words'))
```

Output: ['here', 'are', 'some', 'words']

\s: is space

```
print(re.split(r'(\s*)', 'here are some words'))
```

Output: ['here', ' ', 'are', ' ', 'some', ' ', 'words']

(\s*): include space

RegEX:

```
print(re.findall(r'\d{1,3}\s\w+\s\w+', 'here Are No.435 3rd cross 4th main blah blah'))  
['435 3rd cross']
```

```
print(re.findall(r'\d{1,3}\s\w+\s\w.{4}', 'here Are No.435 3rd cross 4th main blah blah'))  
['435 3rd cross']
```

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.
- `re.compile(pattern, flags=0)`
- Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()` and `search()` methods, described below.
- The expression's behaviour can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the `|` operator).

Compiling Regular Expressions: Example 1

```
import re
pattern = 'start'
text = 'start shall start by learning about the simplest possible regular expressions. Since regular
expressions are used to operate on strings.'
text2 = 'This is new string to match re module pattern start'

result = re.match(pattern, text)
print(result)
print("_____")
```

- `prog = re.compile(pattern)`
- `result = prog.match(text)`
- `if result:`
 - `print(result.group())`
 -
- `result = prog.match(text2)`
- `if result:`
 - `print(result.group())`
- `else:`
 - `print("Not found")`

Compiling Regular Expressions: Example 2

```
pattern = re.compile("o")  
print(pattern.match("dog")) # No match as "o" is not at the start of "dog".  
print(pattern.match("dog", 1)) # Match as "o" is the 2nd character of "dog".
```

- Regular expressions helps in string searching and manipulations.
- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.
- When writing regular expression in Python, it is recommended that you use raw strings instead of regular Python strings.

Regular Expression Objects

- `regex.search(string[, pos[, endpos]])`
- Scan through string looking for a location where this regular expression produces a match, and return a corresponding match object.
- Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

```
>>> import re
```

```
>>> line = "He is a German called Mayer."
```

```
>>> if re.search(r"M[ae][iy]er",line): print "I found one!"
```

```
>>> s1 = "Mayer is a very common Name"
```

```
>>> s2 = "He is called Meyer but he isn't German."
```

```
>>> print re.search(r"^M[ae][iy]er", s1)
```

```
<_sre.SRE_Match object at 0x7fc59c5f26b0>
```

```
>>> print re.search(r"^M[ae][iy]er", s2)
```


re.search

```
>>> s = s2 + "\n" + s1
```

```
>>> print re.search(r"^M[ae][iy]er", s)
```

```
None
```

```
>>> print re.search(r"^M[ae][iy]er", s, re.MULTILINE)
```

```
<_sre.SRE_Match object at 0x7fc59c5f26b0>
```

```
>>> print re.search(r"^M[ae][iy]er", s, re.M)
```

```
<_sre.SRE_Match object at 0x7fc59c5f26b0>
```

```
>>> print re.match(r"^M[ae][iy]er", s, re.M)
```

```
None
```

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs\n Dogs are loyal than Cats";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print ("match --> matchObj.group() : ", matchObj.group())
else:
    print ("No match!!")
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
    print ("search --> searchObj.group() : ", searchObj.group())
else:
    print ("Nothing found!!")
```

Re.search() vs re.match()

- Both functions do exactly the same, with the important distinction that re.search() will attempt the pattern throughout the string, until it finds a match.
- re.match() on the other hand, only attempts the pattern at the very start of the string

Match Objects

- Match objects always have a boolean value of True.
- Since `match()` and `search()` return `None` when there is no match, you can test whether there was a match with a simple if statement:

- Syntax:

```
match = re.search(pattern, string)
```

```
if match:
```

```
    process(match)
```

Match Objects

- Match objects support the following methods and attributes:
- `match.expand(template)`
- Return the string obtained by doing backslash substitution on the template string template, as done by the `sub()` method.
- `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.

Match Objects

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
```

```
>>> m.group(0) # The entire match
```

```
'Isaac Newton'
```

```
>>> m.group(1) # The first parenthesized subgroup.
```

```
'Isaac'
```

```
>>> m.group(2) # The second parenthesized subgroup.
```

```
'Newton'
```

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

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

```
import re
# Lets use a regular expression to match a date string. Ignore
# the output since we are just testing if the regex matches.
regex = r"([a-zA-Z]+) (\d+)"
if re.search(regex, "June 24"):
    match = re.search(regex, "June 24")
    print ("Match at index %s, %s" % (match.start(), match.end()))
    # this will print "June 24"
    print ("Full match: %s" % (match.group(0)))
    # this will print "June"
    print "Month: %s" % (match.group(1))
    # So this will print "24"
    print "Day: %s" % (match.group(2)) # If re.search() does not match, then None is returned
else:
    print "The regex pattern does not match. :("
```

module exports the following functions

- `split(pattern, string, maxsplit=0, flags=0)`
- Split the source string by the occurrences of the pattern, returning a list containing the resulting substrings.
- `sub(pattern, repl, string, count=0, flags=0)`

Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl.

RegEX:

- `print(re.split(r'(s*)','here are some words'))`
- `print(re.split(r'[a-h]','here are some words blah blah'))`
`['', '', 'r', ' ', 'r', ' som', ' wor', 's ', 'l', '', ' ', 'l', '', '']`
- `print(re.split(r'[a-zA-F0-9p-r]','here Are some Words bLAh blah'))`
- `print(re.split(r'[a-zA-F][a-zA-H]','here Are some Words bLAh blah'))`
- `print(re.findall(r'\d{1,3}\s\w+', 'here Are No.435 3rd cross 4th main blah blah'))`
`['435 3rd']`

module exports the following functions

- `findall(pattern, string, flags=0)`
Return a list of all non-overlapping matches in the string.
- `finditer(pattern, string, flags=0)`
Return an iterator over all non-overlapping matches in the string.
For each match, the iterator returns a match object.
- `compile` : Compile a pattern into a `RegexObject`.
- `purge` : Clear the regular expression cache.

Example:findall()

```
import re
# Lets use a regular expression to match a few date strings.
regex = r"[a-zA-Z]+ \d+"
matches = re.findall(regex, "June 24, August 9, Dec 12")
for match in matches:
    # This will print:
    #  June 24
    #  August 9
    #  Dec 12
    print "Full match: %s" % (match)
```

Example: finditer()

```
import re
```

```
# If we need the exact positions of each match
```

```
regex = r"([a-zA-Z]+) \d+"
```

```
matches = re.finditer(regex, "June 24, August 9, Dec 12")
```

```
for match in matches:
```

```
    # This will now print:
```

```
    # 0 7
```

```
    # 9 17
```

```
    # 19 25
```

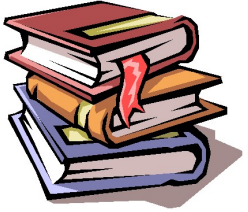
```
    # which corresponds with the start and end of each match in the input string
```

```
    print "Match at index: %s, %s" % (match.start(), match.end())
```

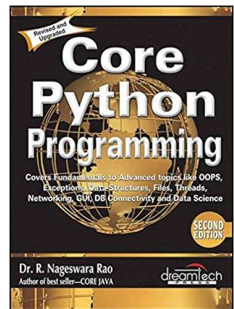
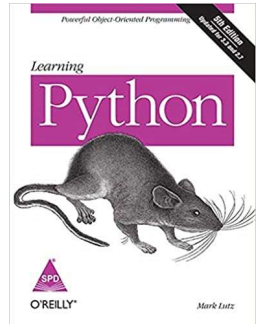
RegEX:

```
def Main():  
    parser = argparse.ArgumentParser()  
    parser.add_argument('word',help='specify word to search for')  
    parser.add_argument('fname',help='specify file to search for')  
    args = parser.parse_args()  
    search_file = open(args.fname)  
    line_num = 0  
    for line in search_file.readlines():  
        line = line.strip('\n\r')  
        line_num +=1  
        search_result = re.search(args.word,line,re.M|re.I)  
        if(search_result):  
            print(str(line_num)+' : '+line)  
if(__name__ == "__main__"):  
    Main()
```

References



- Python 3.x.x documentation: <https://docs.python.org/3/>
- Learning Python: Powerful Object-Oriented Programming: 5th Edition
- Core Python Programming [by [R. Nageswara Rao](#) (Author)]



Thank you

Learning & Development

Tata Elxsi

Bangalore

www.tataelxsi.com

Confidentiality Notice

This document and all information contained herein is the sole property of Tata Elxsi Limited and shall not be reproduced or disclosed to a third party without the express written consent of Tata Elxsi Limited.

TATA ELXSI