SciComp with Py

Regular Expressions

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- Finite State Machines
- RegExp Terminology
- Defining Text Matching Patterns with RegExps
- Groups & Backreferences

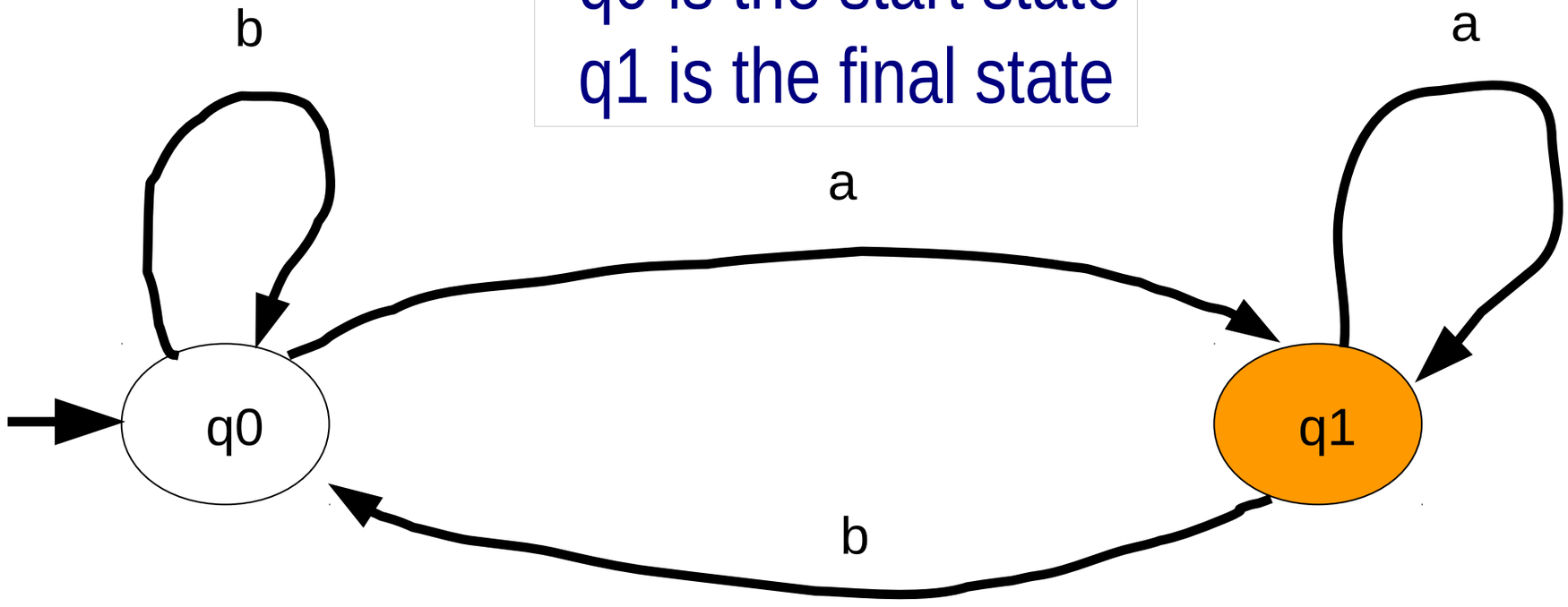
Finite State Machines

Finite State Machines (FSMs)

- Finite state machines (aka finite state automata) can be informally defined as directed graphs whose nodes are states and whose edges are transitions on specific symbols
- A finite state machine (FSM) has a ***unique*** start state and a set (possibly empty) of ***final*** or ***accepting*** states
- A FSM processes the input string one symbol at a time; when the last symbol is read, the FSM reaches a state which is either final or not; if the state is final, the FSM ***accepts*** (***recognizes***) the string; if the state is not final, the FSM rejects the string

FSM Example 1

q0 is the start state
q1 is the final state



The language of this FSM is the set of strings over the alphabet of **{a, b}** that end in **a**, e.g., **a**, **bbba**, **aaa**, etc.

RegExps

- RegExps are programmatic equivalents of finite state machines
- RegExps are compiled into finite state machines at run time; these machines are used for matching
- Another common abbreviation used for RegExps is REs
- In CS, RegExps are often referred to as ***patterns***

Applications of RegExps & FSMs

- RegExps are one of the most useful inventions in CS
- RegExps are used in various text processing applications as diverse as bioinformatics, spellchecking, and log analysis
- FSMs are used in various areas of robotics to simulate and implement animal behavior

RegExps Support

- Many modern programming languages provide support for regular expressions
- Perl, JS, Ruby, AWK provide built-in support
- Python, Java, C++ provide RegExp libraries (e.g. **import re** in Py)
- Historical note: Perl REs have become a de facto standard for other programming languages

RegExp Terminology

Special Characters & Character Classes

- Special characters match all characters in a character class
- Character ranges match all characters in a specified character range
- Special characters are typically escaped (with \) in strings that specify regular expressions

Common Special Characters

- **\d** – any digit character
- **\D** – any non-digit character
- **\w** – any word character (alphanumeric or underscore)
- **\W** – any non-word character
- **\s** – any whitespace character (space, tab, carriage return, newline)
- **\S** – any non-whitespace character

Common Metacharacters

Some two-character sequences that start with **** are called ***metasymbol*** or ***metacharacters***. Examples:

- **\t** defines a tab
- **\n** defines a newline
- **\b** defines a word boundary
- **(** defines the beginning of a group (more on this later)
- **)** defines the end of a group (more on this later)
- If you want to match a metacharacter as a regular character in a string, you must escape it: ****, **\(**

Quantifiers

Quantifiers are special symbols for matching multiple instances of the same pattern. Examples:

- **pattern*** – match zero or more occurrences of **pattern** (e.g., **\d***, **\w***)
- **pattern+** – match one or more occurrences of **pattern** (e.g., **\d+, \w+**)
- **pattern?** – match zero or one occurrence of **pattern** (e.g., **\d?, \w?**)
- **pattern{n}** – match exactly **n** occurrences of **pattern** (e.g., **\d{3}** , **\w{3}**)
- **pattern{n, m}** – match from **n** to **m** occurrences of **pattern** (e.g., **\d{3, 5}** , **\w{5, 10}**)

RegExp Specifications

- Many Py programmers use raw string notation **r''** for regular expression specifications
- Raw strings preserve backslashes w/o interpolating them, which makes the RegExp developers' job a bit easier
- It is possible to use regular strings to specify patterns, but using **r''** is more conventional

Defining Text Patterns with RegExps

Common RE Use

```
import re
##########################################################
## Match pattern pat against text txt case sensitively
match = re.search(r'pat', txt)
```

Literal Matching

In literal matching, we are interested in finding in a given text the presence/absence of some exact sequence of characters (exactly as they are spelled – hence the term literal). Code sample below looks for the sequence **'knowledge'** in the text, finds it, and prints **'match found'**.

```
import re

txt = "Assuredly there is a price on this knowledge. It is to be\n\
given only to those who can keep it and not lose it.\n\
\n\
\tAl-Ghazali, Book of Knowledge\n\
"

if re.search(r'knowledge', txt): print 'match found'
```

Case-Sensitive vs. Case-Insensitive Matching

Default matching is case sensitive. If you want to do case-insensitive matching, you need the re.IGNORECASE flag

```
##############################################################

## Match pattern pat against text txt case insensitively

match = re.search(r'pat', txt, re.IGNORECASE)
```

Case-Sensitive vs Case-Insensitive Matching

In the code sample below the sequence **'al-ghazali'** is searched for case sensitively and case insensitively. Only case insensitive match succeeds, so only **'match 2 found'** is printed.

```
import re

txt = "Assuredly there is a price on this knowledge. It is to be\n\
given only to those who can keep it and not lose it.\n\
\n\
\tAl-Ghazali, Book of Knowledge\n\
"

if re.search(r'al-ghazali', txt): print 'match 1 found'
if re.search(r'al-ghazali', txt, re.IGNORECASE): print 'match 2 found'
```

Problem

Let's write functions to check the strings below for the presence/absence of these special characters **\d**, **\D**, **\w**, **\W**, **\s**, **\S**:

1) **'12345'**
2) **'abcde'**
3) **' .;!?\~'**
4) **' .;!?_'**
5) **' .;!?\~_\n'**
6) **' \t\n'**

source code is in py2_re_trials.py and py3_re_trials.py

Looking for Digit Chars with \d

Py

```
import re


txt_01 = '12345'
txt_02 = 'abcde'
txt_03 = ' .;!?\\'
txt_04 = ' .;!?\\_'
txt_05 = ' .;!?\\_\n';

txt_lst = (txt_01, txt_02, txt_03, txt_04, txt_05)

def find_digit_char(txt):
    if re.search(r'\d', txt):
        print 'there is at least one digit char in ' + repr(txt)
    else:
        print 'there are no digit chars in ' + repr(txt)

def digit_char_tests(txts):
    print r'**** \d Tests ****'
    for txt in txts: find_digit_char(txt)
```

Output

```
>>> digit_char_tests(txt_lst)
***** \d Tests *****
There is at least one digit char in '12345'
There are no digit chars in 'abcde'
There are no digit chars in ' .;!?\\~'
There are no digit chars in ' .;!?\\_'
There are no digit chars in ' .;!?\\~_\n'
There are no digit chars in ' \t\n'
```

Looking for Non-Digit Chars with \D

Py

```
def find_nondigit_char(txt):
    if re.search(r'\D', txt):
        print 'there is at least one nondigit char in ' + repr(txt)
    else:
        print 'there are no nondigit chars in ' + repr(txt)



def nondigit_char_tests(txts):
    print r'**** \D Tests ****'
    for txt in txts: find_nondigit_char(txt)
```

Output

```
>>> nondigit_char_tests(txt_lst)
***** \d Tests *****
There are no nondigit chars in '12345'
There is at least one nondigit char in 'abcde'
There is at least one nondigit char in ' .;!?\\~'
There is at least one nondigit char in ' .;!?\\_'
There is at least one nondigit char in ' .;!?\\~_\n'
There is at least one nondigit char in ' \t\n'
```

Generalizing Previous Two Cases with Functional Abstraction

make_re_match_fun is a function that returns another Function (search_fun). Such functions are called closures.

```
def make_re_match_fun(regex, pos_msg, neg_msg):
    def search_fun(txt):
        if re.search(regex, txt):
            print pos_msg + repr(txt)
        else:
            print neg_mssg + repr(txt)
    return search_fun

def run_re_tests(re_match_fun, txts):
    for txt in txts:
        re_match_fun(txt, 'match found', 'match not found')
```

Test 1: NON-DIGIT CHARS

Py

```
txt_01 = '12345'
txt_02 = 'abcde'
txt_03 = ' .;!?\~'
txt_04 = ' .;!?\_'
txt_05 = ' .;!?\~_\n'
txt_06 = ' \t\n'

txt_lst = (txt_01, txt_02, txt_03, txt_04, txt_05, txt_06)
print '******Test 1: NON-DIGIT CHARS\n',
run_re_tests(make_re_match_fun(r'\D',
                    'There is a non-digit character in ',
                    'There are no non-digit characters in '),
                txt_lst)
```

Output

```
******Test 1: NON-DIGIT CHARS
There are no non-digit characters in '12345'
There is a non-digit character in 'abcde'
There is a non-digit character in ' .;!?\\~'
There is a non-digit character in ' .;!?\\_'
There is a non-digit character in ' .;!?\\~_\n'
There is a non-digit character in ' \t\n
```

Test 2: WORD CHARS

Py

```
txt_01 = '12345'
txt_02 = 'abcde'
txt_03 = ' .;!?\~'
txt_04 = ' .;!?\_'
txt_05 = ' .;!?\~_\n'
txt_06 = ' \t\n'

txt_lst = (txt_01, txt_02, txt_03, txt_04, txt_05, txt_06)
print '******Test 2: WORD CHARS\n',
run_re_tests(make_re_match_fun(r'\w',
                               'There is a word character in ',
                               'There are no word characters in '),
                             txt_lst)
```

Output

```
******Test 2: WORD CHARS
There is a word character in '12345'
There is a word character in 'abcde'
There are no word characters in ' .;!?\\~'
There is a word character in ' .;!?\\_'
There is a word character in ' .;!?\\~_\n'
There are no word characters in ' \t\n'
```

Debugging RegExps

- RegExps pack a lot of information and can be difficult to understand and debug
- When you work on defining a regular expression, write a few simple test cases and run them on short strings instead of large strings or files
- When your RE works on all your test cases, start testing it on longer strings or files
- Document your regular expressions with sample inputs and outputs: your fellow programmers appreciate your efforts even if they never let you know about it

Groups & Backreferences

Grouping RegExps

- Regular expressions can be broken into components called **groups**
- A group match is a specific part of text that matches a specific regular subexpression in a larger regular expression
- Groups are specified with a pair of matching parentheses: **( )**
- Group matches are numbered 1, 2, 3, etc. These numbers are called **backreferences**, because they refer back to specific text segments that match specific regular subexpressions in a larger regexp

Finding Number of Captured Groups (Clusters)

We can find the total number of captured (matched) groups, as shown below. If there is no match, the returned math object is equal to None, hence the check.

```
match = re.search(r'pat', txt, re.IGNORECASE)
if match != None:
    num_of_groups = len(match.groups())
```

Matching a Group of 1 Digit Character

Py

```
def test_1(txt):
    match_01 = re.search(r'(\d)', txt)
    if not match_01 is None:
        print "group = '%s'" % match_01.group()
        print "span  = %s" % str(match_01.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

Output

```
>>> txt_01
'12345'
>>> test_1(txt_01)
group = '1'
span  = (0, 1)
---------------
>>> txt_02
'1+1=2'
>>> test_1(txt_02)
group = '1'
span  = (0, 1)
---------------
>>> txt_03
' a + b =     c   '
>>> test_1(txt_03)
no match found
---------------
```

Matching a Group of 1 or More Digit Characters

Py

```
def test_2(txt):
    match_02 = re.search(r'(\d+)', txt)
    if not match_02 is None:
        print "group = '%s'" % match_02.group()
        print "span = %s" % str(match_02.span())
        print('---------------')
    else:
        print 'no match found'
        print '---------------'
```

Output

```
>>> txt_01
'12345'
>>> test_2(txt_01)
group = '12345'
span = (0, 5)
---------------
>>> txt_02
'1+1=2'
>>> test_2(txt_02)
group = '1'
span = (0, 1)
---------------
>>> txt_03
' a + b =     c   '
>>> test_2(txt_03)
no match found
---------------
```

Problem: A Sample RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'(\d)(\w*)(\d)'
```

Problem: A Sample RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'(\d)(\w*)(\d)'
```

This is the 1st group that matches exactly 1 digit character.

Problem: A Sample RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'(\d)(\w*)(\d)'
```

This is the 2nd group that matches exactly 0 or more alphanumerics.

Problem: A Sample RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'(\d)(\w*)(\d)'
```

This is the 3rd group that matches exactly 1 digit character.

Question

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('12345')
```

Answer

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('12345')
groups  = ('1', '234', '5')
span    = (0, 5)
group 1 = 1
group 2 = 234
group 3 = 5
---------------
```

Question

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('1abcd_efg5')
```

Answer

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('1abcd_efg5')
groups  = ('1', 'abcd_efg', '5')
span    = (0, 10)
group 1 = 1
group 2 = abcd_efg
group 3 = 5
---------------
```

Question

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('12 345 abcd 9')
```

Answer

Py

```
def test_3(txt):
    match_03 = re.search(r'(\d)(\w*)(\d)', txt)
    if not match_03 is None:
        print 'groups  = %s' % str(match_03.groups())
        print 'span    = %s' % str(match_03.span())
        print 'group 1 = %s' % str(match_03.group(1))
        print 'group 2 = %s' % str(match_03.group(2))
        print 'group 3 = %s' % str(match_03.group(3))
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_3('12 345 abcd 9')
groups  = ('1', '', '2')
span    = (0, 2)
group 1 = 1
group 2 =
group 3 = 2
---------------
```

Problem: Another RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'(\d+)\+(\d+)=(\d+)'
```

Question

Py

```
def test_4(txt):
    match_04 = re.search(r'(\d+)\+(\d+)=(\d+)', txt)
    if not match_04 is None:
        print 'group 1 = %s' % str(match_04.group(1))
        print 'group 2 = %s' % str(match_04.group(2))
        print 'group 3 = %s' % str(match_04.group(3))
        print 'span    = %s' % str(match_04.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_4('10+15=25')
```

Answer

Py

```
def test_4(txt):
    match_04 = re.search(r'(\d+)\+(\d+)=(\d+)', txt)
    if not match_04 is None:
        print 'group 1 = %s' % str(match_04.group(1))
        print 'group 2 = %s' % str(match_04.group(2))
        print 'group 3 = %s' % str(match_04.group(3))
        print 'span    = %s' % str(match_04.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_4('10+15=25')
group 1 = 10
group 2 = 15
group 3 = 25
span    = (0, 8)
```

Question

Py

```
def test_4(txt):
    match_04 = re.search(r'(\d+)\+(\d+)=(\d+)', txt)
    if not match_04 is None:
        print 'group 1 = %s' % str(match_04.group(1))
        print 'group 2 = %s' % str(match_04.group(2))
        print 'group 3 = %s' % str(match_04.group(3))
        print 'span    = %s' % str(match_04.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_4('10+x=25')
```

Answer

Py

```
def test_4(txt):
    match_04 = re.search(r'(\d+)\+(\d+)=(\d+)', txt)
    if not match_04 is None:
        print 'group 1 = %s' % str(match_04.group(1))
        print 'group 2 = %s' % str(match_04.group(2))
        print 'group 3 = %s' % str(match_04.group(3))
        print 'span    = %s' % str(match_04.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What is the output?

```
>>> test_4('10+x=25')
no match found
```

Problem: Another RE with Groups

Here is a regexp with groups. Let's try to identify how many groups it contains and what each group will match.

```
r'\s*(\w+)\s*(\+|\-|\*)\s*(\w+)\s*=\s*(\w+)\s*'
```

Question

Py

```
def test_5(txt):
    match_05 = re.search(r'\s*(\w+)\s*(\+|\-|\*)\s*(\w+)\s*=\s*(\w+)\s*', txt)
    if not match_05 is None:
        print 'group 1 = %s' % str(match_05.group(1))
        print 'group 2 = %s' % str(match_05.group(2))
        print 'group 3 = %s' % str(match_05.group(3))
        print 'span    = %s' % str(match_05.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What's the output?

```
>>> test_5('x - 100= 5')
```

Answer

Here's the output

```
>>> test_5('x - 100= 5')
group 1 = x
group 2 = -
group 3 = 2
span    = (0, 5)
```

Question

Py

```
def test_5(txt):
    match_05 = re.search(r'\s*(\w+)\s*(\+|\-|\*)\s*(\w+)\s*=\s*(\w+)\s*', txt)
    if not match_05 is None:
        print 'group 1 = %s' % str(match_05.group(1))
        print 'group 2 = %s' % str(match_05.group(2))
        print 'group 3 = %s' % str(match_05.group(3))
        print 'span    = %s' % str(match_05.span())
        print '---------------'
    else:
        print 'no match found'
        print '---------------'
```

What's the output?

```
>>> test_5('   100 +x = 200 ')
```

Answer

Here's the output

```
>>> test_5('   100 +x = 200 ')
group 1 = 100
group 2 = +
group 3 = x
span    = (0, 16)
---------------
```

More Examples of Group References

Problem

Design a RegExp that parses email addresses into user name, host name, and host extension.

source code is in py2_email_regexp.py and py3_email_regexp.py

Sample String with Emails

```
import re
TXT_01 = '\n\
John   Balbiro   A1001  john.balbiro@usu.edu\n\
Alice  Nelson    A0011  alice.nelson@workflow.net\n\
Jacob  Roberts A1100  j.s.roberts@gmail.com\n\
'
```

A Possible Solution

```
'([\w\.-]+)@([\w\.-]+)\.(com|net|org)'
```

Find All RE Matches with re.findall()

```
## findall is a useful function in the re module
## it takes a pattern and a text and returns a list
## of matches of the pattern in the text
match_list = re.findall(pat, txt)
```

Matching Emails with Two Groups

```
## group 1 - user name @ host name; group 2 extension
email_pat_with_2_groups = r'([\w.-]+@[\w.-]+)\.(com|net|org|edu)'
emails_02 = re.findall(email_pat_with_2_groups, TXT_01)
```

Matching Emails with 3 Groups

```
## g1 - user name; g2 - host name; g3 - extension
email_pat_with_3_groups = r'([\w.-]+)@([\w.-]+)\.(com|net|org)'
emails_03 = re.findall(email_pat_with_3_groups, TXT_01)
```

RegExp Tutorial

https://regexone.com/ is a decent regexp tutorial that you may want to take a look at