Lab 7:
Regular Expressions

What the heck is this?

- Wildcards on steroids!

- A ***regular expression*** *(or* ***regex)*** is a sequence of characters (or rules) that describes a **pattern.**

- There is no “universal” rules for regex, so each language will implement their own variation of rules
- www.regex101.com is a good resource for testing RegEx patterns in real time
  - (make sure to select Python under “Flavor” on the left menu)

Regular Expressions - Basics

- Brackets
  - abc - Match a AND b AND c
  - [abc] - Match a OR b OR c
  - [a-z] - Indicate a range (but only match 1 character!)
- Groups
  - You can extract groups from an entire match using parenthesis
  - For regex=[abc]([0-9]), if the string is “a0bob”, then
    - “a0” would be the entire match
    - “0” would be group 1 match

Regular Expressions - Basics

- Ordinary character match themselves exactly
  - **** removes the ‘specialness’ of a meta character (\ is known as an escape character)
- **.** (period) matches **ANY** character EXCEPT newline ‘\n’
- Indicate a range of characters using [ ]
  - **[a-zA-Z]** : Matches any of lower-case and upper-case characters
  - **[0-9]** : Matches any one single number (== **\d)**
- **^** - negation (do not match these characters)
    - [^0-9] : matches not a digit (== **\D**)
- **\w** matches a ‘word’ character (== [a-zA-Z0-9_])
  - **\W** matches NOT a word
- **\s** matches single white character (==[\n\r\t\f])
  - **\S** matches NOT single white character

Regular Expressions - Basics

- Multipliers
  - * - preceding pattern occurs **zero or more times**
  - + - preceding pattern occurs **one or more times**
  - ? - preceding pattern occurs **zero or one times**
  - {N} - preceding pattern occurs **exactly N times**
  - {N,M} - preceding pattern occurs **between N and M times (inclusive)**
  - {N,} - preceding pattern occurs **at least N times**
- Anchors
  - ^ - represents the beginning of a string
  - $ - represents the end of the string
  - \b - represents a word boundary (between a word and a not word)
- Backreference (\1 \2 , etc.)
  - Refer to something matched by a previous grouping (\1 → group 1)

Some Easy Regex Examples

1. ‘[a-z][0-9]+’
   a. k9
   b. A7
   c. f001
   d. r2d2
2. ‘([a-z][0-9])+’
   a. Same results?
3. ‘(d\w+)\W(d\w+)’
   a. dog treat
   b. dog dot
   c. dog Dot
   d. dogdot
4. ‘a(ab)*a’
   a. aa
   b. aba
   c. abababa
   d. aaba
5. ‘ab+c?’
   a. abc
   b. abbbbbbbbb
   c. ac
   d. bbc

Harder Examples!

1. What numbers would r’**[1-49]’** match?
2. Does r‘**\$\d{4}**’ match $1234? How about $12345?
3. What would match: **r’abc|xyz’**
    a. abc
    b. xyz
    c. abc|xyz
4. What would match: **r’(very)+(fat)?(tall|ugly) man’**?
    a. very fat man
    b. fat tall man
    c. very very fat ugly man
    d. very very tall man

re module

- Python has a regular expression module, called **re**
  - **re.search(pattern, str, option), re.match()**
  - **re.findall(pattern,str, option), re.finditer()**
  - **re.sub(pattern,replace,str)**
- Option Flag
  - IGNORECASE - ignore case
  - DOTALL - include newline in dot (.) matching
  - MULTILINE - Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.
- To create a pattern string use:
  - r'*pattern'*
    - Note: these are SINGLE quotes
  - Or you can compile a string as a regex
    - **regex = re.compile("pattern")**

re.search

- Find the FIRST match in the string
  - Returns match if there is one, otherwise
  - Returns None

- re.match() checks if the regex is at the START of a string

- Use <match obj>.group(<index>) to get individual group matches
  - group() will return all groups

re.search - Example 1

```
haystack = 'an example word:cat!!'
#This is the needle (or search pattern)
# word: - Exactly match
# \w\w\w - 3 word characters [0-9a-zA-Z_] in a row
needle=r'word:\w\w\w'
match = re.search(needle, haystack)

if match:
    print('Found', match.group())
else:
    print("No {} in {}".format(needle, haystack))
```

re.search - Example 2

```
 haystack = 'purple alice-b@google.com monkey dishwasher'
#[\w.-]+ - A word (\w) OR any char but newline (.) OR -,  1 or
          more (+) times
#@ - Match exactly
needle = r'([\w.-]+)@([\w.-]+)'
match = re.search(needle, haystack)

if match:
    print("Group 0 (whole match): ", match.group())
    print("Group 1: ", match.group(1))
    print("Group 2: ", match.group(2))
```

re.findall

- Find every match in the string
  - Returns a list of strings of all matches
- When using groups, a list of tuples is returned (each tuple a group)
- re.finditer() returns an iterable object instead

re.findall - Example 3

```
haystack = 'purple alice@google.com, blah monkey
bob@abc.com blah dishwasher'

# [\w\.-] - Match a word (\w) OR . exactly (not the special dot)
or -, 1 or more (+) times
needle=r'[\w\.-]+@[\w\.-]+'

emails = re.findall(needle, haystack)
for email in emails:
    print(email)
```

re.findall - Example 4

```
haystack = 'purple alice@google.com, blah monkey bob@abc.com
blah dishwasher'
#Notice, same regex as Example 1,
#but now we're grabbing two groups ()
needle=r'([\w\.-]+)@([\w\.-]+)'

matches = re.findall(needle, haystack)

for (username,host) in matches:
    print("Username: {}\t\tHost: {}".format(username, host))
```

re.sub

- Substitution
  - The **re.sub(pat, replacement, str)** function searches for all the instances of pattern in the given string, and replaces them. The replacement string can include '\1', '\2' which refer to the text from group(1), group(2), and so on from the original matching text.

```
haystack = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
needle = r'([\w\.-]+)@([\w\.-]+)'
# \1 is group(1), \2 group(2) in the replacement
# Keep group 1 match and replace rest of match
substr = r'\1@yo-yo-dyne.com'
new_string = re.sub(needle, substr, haystack)
print(new_string)
```

In-lab activity: Regular Expressions

A regular expression interactive test script has been created for you to use. There are 4 exercises provided (from easy to hard). To run the script:

```
$ ./test_regex.py -e <num> --regex “<regex>”
```

-e <num>: The exercise [0-3] to run
--no-run : Only print the positive and negative examples
--regex “regex” : Your regular expression to your passing (must include “ “)

Your regular expression must match completely the positive examples and NOT match the negative examples.