What Is a Regular Expression?

• A pattern that matches all, or part of, some desired text string
• Pattern is compared to a given text string
• Returns Success or Failure depending on whether the given string contains the desired text string
• Python syntax:
  
  ```python
  re.search( 'pattern', 'given-string' )
  ```
**Simple Examples**

- **import re**
  - use regular expression processing

- **re.search( 'dab', 'abracadabra' )**
  - is successful
  - similar to 'abracadabra'.find('dab')

- **re.search( 'dab', 'hocus-pocus' )**
  - fails

- **re.search( 'Cat', 'catch' )**
  - fails unless case-sensitivity is turned off

**Remarks**

- Regular expressions are commonly called *regexes*, or *R.E.s*

- Interesting regexes form variable patterns, *i.e. can match more than one distinct string*

- Useful regexes are formed to match a desired category of strings
  - example: a phone number
    - a string of 3 digits, a separating character, 3 more digits, another separating character, then 4 digits
Some Technical Details

• Each distinct, matchable character in the pattern is an **atom**
  - a single atom is a valid, minimal regex

• Two atoms adjacent are a **conjunction**
  - logical "AND"
  - a conjunction is also a valid regex

• A conjunction of two regexes is a regex

• A **disjunction** (logical "OR") of two regexes forms a regex
  - symbolized with "|", a "pipe" or "vertical bar"

Disjunction example

• **re.search**('a|b|c|d', 'ABCDcEGbcdefg')
  - successful
  - python returns a "match object" that indicates where the match occurred and what matched

• **re.search**('dog|cat', 'catsANDDogs')
  - successful

• **re.search**('|', 'abcd|efgh')
  - successful
  - match object probably isn't what you expect
Atoms

- A normal character matches itself
  - called a literal
  - the previous examples consisted of literals

- Some characters have special meanings in regexes
  - period ., caret ^, dollar sign $
  - vertical bar |, question mark ?, asterisk *, plus +
  - Parentheses (), square brackets [], curly braces {}

- Some escaped characters represent categories
  - ...and user-defined categories are available

Special Characters

- Periods are very special atoms
  - Match any single character (with a few exceptions)

- Caret, dollar sign
  - Positional items, match at a location instead of a character
  - caret ^ - matches the beginning of a string
  - dollar sign $ - matches the end of a string

- Example:
  - `re.search('^dog|cat$', 'catsANDdogs')`
    - fails – requires "dog" at beginning of line or "cat" at end of line
**Quantifiers**

- A particular strength of regexes is the ability to specify repetitions of a simple pattern. **Quantifiers** control how many occurrences of an atom to match.
  - `?` – match 0 or 1 occurrence of preceding atom
  - `+` – match 1 or more occurrences of preceding atom
  - `*` – match 0 or more occurrences of preceding atom
- By default, quantifiers are **greedy** – they match as many occurrences as possible

---

**Special Characters - Summary**

<table>
<thead>
<tr>
<th>name</th>
<th>symbol(s)</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>period</td>
<td>.</td>
<td>match any single character</td>
</tr>
<tr>
<td>caret</td>
<td>^</td>
<td>match at beginning of string</td>
</tr>
<tr>
<td>dollar sign</td>
<td>$</td>
<td>match at end of string</td>
</tr>
<tr>
<td>asterisk</td>
<td>*</td>
<td>match arbitrary number (0 or more) of preceding regex</td>
</tr>
<tr>
<td>plus sign</td>
<td>+</td>
<td>match 1 or more of preceding regex</td>
</tr>
<tr>
<td>question mark</td>
<td>?</td>
<td>match 0 or 1 of preceding regex</td>
</tr>
<tr>
<td>square brackets</td>
<td>[ ]</td>
<td>match any 1 of the characters within the brackets</td>
</tr>
<tr>
<td>parentheses</td>
<td>( )</td>
<td>collect (&quot;group&quot;) a regex into an atom; can be used with * + ?</td>
</tr>
<tr>
<td>curly braces</td>
<td><code>{n}</code></td>
<td>requires exactly n occurrences, or at least n and no more than m, of the preceding regex</td>
</tr>
</tbody>
</table>
## Examples of Special Characters in Regexes

<table>
<thead>
<tr>
<th>pattern</th>
<th>matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.g</td>
<td>dog, dig, dDg, d.g ...</td>
</tr>
<tr>
<td>d\g</td>
<td>d.g only</td>
</tr>
<tr>
<td>dog*</td>
<td>do, dog, dogg, dogggggggggggg ...</td>
</tr>
<tr>
<td>dog+</td>
<td>dog, dogg, doggggggggggg ...</td>
</tr>
<tr>
<td>dog?</td>
<td>do, dog</td>
</tr>
<tr>
<td>^dog</td>
<td>dog <strong>at beginning of string only</strong></td>
</tr>
<tr>
<td>dog$</td>
<td>dog <strong>at end of string only</strong></td>
</tr>
<tr>
<td>[dog]</td>
<td>d, o, or g <strong>only</strong></td>
</tr>
<tr>
<td>[aeiouAEIOU]</td>
<td>any uppercase or lowercase vowel</td>
</tr>
<tr>
<td>(dog)</td>
<td>dog <strong>as a group</strong></td>
</tr>
</tbody>
</table>

## Escaping a Special Character

- Special characters are *escaped* with a backslash "\" to match the actual character
- Compare the output of
  - `re.search('|', 'abc|def')`
- to the output from
  - `re.search('\|', 'abc|def')`
- show the patterns themselves:
  - `print( '|', '\| ')`
try

- Match strings that look like numbers
  - 0, 1, 2, etc.
  - 12345
  - 3.14159
  - 123.4567890

- Match strings that look like telephone numbers
  - 570-389-4500
  - (570) 389-4000, (570)389-4000

  what's the difference?

---

**Escaped Characters and Categories**

<table>
<thead>
<tr>
<th>escaped character</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>any digit (same as [0123456789] or [0-9])</td>
</tr>
<tr>
<td>\D</td>
<td>any non-digit</td>
</tr>
<tr>
<td>\w</td>
<td>any &quot;wordlike&quot; character (any alphanumeric) (same as [a-zA-Z0-9_])</td>
</tr>
<tr>
<td>\W</td>
<td>any non-wordlike character (punctuation, whitespace, etc.)</td>
</tr>
<tr>
<td>\s</td>
<td>any &quot;whitespace&quot; character (same as [\n\t\v\f])</td>
</tr>
<tr>
<td>\S</td>
<td>any non-whitespace character</td>
</tr>
<tr>
<td>\b</td>
<td>any word boundary</td>
</tr>
<tr>
<td>\B</td>
<td>any non-word boundary</td>
</tr>
<tr>
<td>. or ( or ) or ? or \ etc.</td>
<td>the actual character</td>
</tr>
</tbody>
</table>
**Categories of Characters**

- Square brackets `[]` create categories
  - Any character within brackets is matched
  - Contiguous ranges of characters allowed
- Examples:
  - `[aeiouAEIOU]` - matches any vowel
  - `[0123456789]` or `[0-9]` - matches any digit
  - `[-.0-9]` - matches any numeric character
    » leading dash "-" is just a dash character
    » period in a category is just a period

**Negated Categories and Characters**

- Caret at the _beginning_ of a category negates the category
  - `[^0-9]` - matches any character except a digit
  - `[0-9^]` - matches any digit or a caret
- Category can contain single character
  - Provides negation of single characters
  - `[^X]` - matches any character except X
  - `[^X]+` - matches one or more non-X chars
Putting Escaped Characters in Patterns

- '\\' is an escape character in normal text strings, as well as in RE patterns
- Text string as pattern: the escape character itself must be escaped
  - → "\\bbook\\b" matches "book" but not "textbook" or "bookie"
- Alternative: use raw strings, indicated with an 'r'
  - → r"\\bbook\\b" also matches "book" but not "textbook" or "bookie"

…general guideline

- Use raw strings to define patterns
- Always works, avoids some ambiguities
- For example:
  - re.search( 'ate\n', 'skate\n' ) – matches is the same as re.search( 'ate\n', 'skate\n' ) – matches
  - re.search( 'rate\n', 'skate\n' ) – matches is not the same as re.search( 'ate\n', 'skate\n' ) – NO match
(Review? Opening and Using Disk Files)

- **Read from a file:**
  - open filename, create handle
  - read data from handle into string variables
    - use `.readlines()`, `.readline()`, or `.read()` on the handle
  - close the handle

- **Write to a file:**
  - open filename, create handle
  - write strings to the handle
    - use `.writelines()` or `.write()` on the handle
  - close the handle

- **Easy way:**
  with open('foo', 'r') as h:
    for line in h.readlines():
      # process the line
  # closing happens automatically

- **Easy way:**
  with open('foo', 'w') as h:
    for line in mylist:
      h.write(line)
  # lines should include "\n"
  # closing happens automatically

**exercise**

- download and open file "[montcs.bloomu.edu/Datasets/Logfiles/error.log.1]"

- search for lines containing any the characters of 'X', 'Y', or 'Z' (capital letters)
  - print any matching lines

- search for occurrences of strings "123", "456", or "789"
  - print any matching lines

- **Harder? search for lines not containing '7'**
  - count number of lines
  - (should be 121 lines)
**exercise**

- Download and open file
  "montcs.bloomu.edu/Readings/Alice-in-Wonderland.txt"

- Search for, count, and save lines containing any occurrence of the word "the" - but not words containing "the", such as "other"
  - Print count of matching lines

- Search for, count, and save lines containing words that include "the", such as "other" - but not "the" by itself
  - Print count of matching lines
  - Count number of occurrences of each word?
  - Combine capitalization variants, such as "father" and "Father"?

**homework exercises**

- Chun section 1.6: 1-1 .. 1-12, pp.48-49
The re Module: Searches versus Matches

• `re.search()` function looks for a pattern match anywhere within a target string
  - "^" anchors search to beginning of line
  - "$" anchors search to end of line

• `re.match()` function matches a pattern to the entire target string
  - "^", "$" anchors not needed
  - less general, more efficient to execute than `re.search()`

`re.search()` vs. `re.match()` Example

• Find a Social Security number anywhere in a line:
  - `re.search( r'\d{3}-\d\d-\d{4}', line )`

• Find a Social Security number that is the only thing on the line:
  - `re.search( r'^\d{3}-\d\d-\d{4}$', line )`

• Also finds a Social Security number as the only thing on the line:
  - `re.match( r'\d{3}-\d\d-\d{4}', line )`
Groups

- Parentheses ("parens") collect atoms into a group that acts like an atom
  - (abc) is a group containing three characters
  - Parentheses must be escaped if you want to match literal parentheses!
    - BREs use \( and \) for grouping; ( and ) match themselves

- Groups can be affected by quantifiers
  - (abc)+ matches "abc", "abcabc", "abcabcabcabc"
    ...

- Groups can also be referred to elsewhere
  - More on this later.

Match Objects: Working With Groups

- `re.search()` and `re.match()` return a `match object` when they succeed

- Match object contains information about the match, including any parenthesized groups

- The matched, parenthesized groups can be retrieved and used in further processing

- `match_object.groups()` and `match_object.group()` return the groups
Groups Example

a) What will this pattern match with?
\b(\w+)\b.*\b(\w+)\b

b) How do you enter this pattern in Python?

→ "\b(\w+)\b.*\b(\w+)\b"

c) or with a raw string,

r"\b(\w+)\b.*\b(\w+)\b"

Groups Example 2

• Enter this function, then run it and print the result:

```python
def get_name(pat=r'\b(\w+)\b.*\b(\w+)\b'):
    fullname = input('Enter your full name: ')
    m = re.match(pat, fullname)
    if not m == None:
        first = m.group(1)
        last = m.group(2)
        return (last, first)
    else:
        return None
```
Details of `match_object.group()`

- Method `match_object.group()` returns the entire matched string
  - `match_object.group(0)` also returns the matched string

- Optional argument ≥ 1 specifies a parenthesized group
  - groups are numbered left-to-right
  - `match_object.group(1)` returns 1st (leftmost) group, etc.

`match_object.groups()`

- Similar method `match_object.groups()` returns a tuple of all parenthesized groups

- example:

```
In [137]: m = re.search(r'oo.*end', 'bookbinder')

In [138]: print(m.groups())
('oo', 'nd')

In [139]: for i in range(3):
    ...:     print(m.group(i))
    ...:
bookbinder
oo
nd
```

In [140]:
.groups() Example, Elaborated

```python
#!/usr/bin/env python3
# Demonstrate re.groups(), re.group()
#2017-02-09
import re

def try_it( str):
    print( str)
    m = re.search( r'(oo).*\(ee|nd\)', str)
    if m:
        print( 'm.groups():', m.groups())
    for i in range(3):
        print('m.group({}): {}'.format(i, m.group(i)))

words = [ 'bookkeeper', 'bookbinder', 'books feed me', 'bookee', 'ookiee']
for w in words:
    try_it( w)
```

What About Nested Parentheses?

```python
#!/usr/bin/env python3
# Demonstrate re.groups(), re.group()
#2017-02-09
import re

def try_it( str, pat=r'(oo).*\(ee|nd\)\)\)', ngroups=2):
    print( '\n', str, pat)
    p = re.compile(pat)
    m = p.search( str)
    if m:
        print( 'm.groups():', m.groups())
    for i in range(ngroups+1):
        print( 'm.group({}): {}'.format(i, m.group(i)))

for w in [ 'abcabcabcabc', 'booabcabcboo', 'abcdefg']:
    try_it( w, pat= r'(abc)\)', ngroups=1)
    try_it( w, pat= r'((abc)(abc))\)', ngroups=3)
    try_it( w, pat= r'((abc)\)', ngroups=2)
```
**Some More Details of Match Objects**

- Members of a match object `m`:

<table>
<thead>
<tr>
<th>match object's member:</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.pos</td>
<td>position (index) of beginning of match</td>
</tr>
<tr>
<td>m.endpos</td>
<td>position of end of match</td>
</tr>
<tr>
<td>m.regs</td>
<td>start &amp; end positions of each group's match</td>
</tr>
<tr>
<td>m.re</td>
<td>the regular expression that was used to make the match</td>
</tr>
<tr>
<td>m.string</td>
<td>the original target string</td>
</tr>
</tbody>
</table>

**exercise**

- Again open file "[montcs.bloomu.edu/Datasets/Logfiles/error.log.1](montcs.bloomu.edu/Datasets/Logfiles/error.log.1)"

- Search for lines containing an IP address
  - form is 4 groups of 1-3 digits, separated by a period
    - e.g. 123.145.167.189, or 172.16.0.201, or 8.8.8.8 ...

- Collect the matched IP addresses into a list, and also into a set

- When done, print the length of the list and the length of the set
  - Should be 1034 addresses in the list, and 230 in the set
More Regex Functions

• `re.findall()` and `re.finditer()` functions return all matches of a pattern within a string, as a list of match objects

• `re.sub()`, `re.subn()` functions substitute a replacement substring for the matched pattern in a target string

• `re.split()` splits a target string into substrings separated by the pattern

• `re.compile()` : precompile a regex for faster performance of repeated searches

---

**re.findall()**

• The `re.findall()` function searches all non-overlapping occurrences of the provided pattern
  - Returns a list of all matches

• The `re.finditer()` function acts like `re.findall()`, but returns an *iterator* instead of a list of matches
  - An *iterator* is an object that provides each discovered occurrence of the pattern, one at a time – useful in "for" statements, etc.
  - Iterators provide more efficiency
**re.sub()**

- The `re.sub()` function replaces pattern-matches in a target string with a replacement string
  - Returns a modified string
  - Replacement can be a string or a function

- The `re.subn()` function acts like `re.sub()`, returns the number of substitutions made as well as the modified string

**re.compile()**

- The `re.compile()` function compiles a text string that represents a regular expression into a regex object

- Compiled regex objects include methods `.search()`, `.match()`, `.findall()`/.`finditer()`, `.sub()`/.`subn()`

- Regular expressions that are used repeatedly are more efficient if compiled once beforehand
**exercise – anonymizing IP addresses**

- Open file "montcs.bloomu.edu/Datasets/Logfiles/error.log.1"
- Search for lines containing an IP address
  - form is 4 groups of 1-3 digits, separated by a period
    - e.g. 123.145.167.189, or 172.16.0.201, or 8.8.8.8 ...
- Replace every IP address with "xxx.xxx.xxx.xxx"
- Write all lines to a new text file named "error.log.2"
  - Should be identical to the original file, except that all IP addresses have been anonymized