SENG 265 - Fall 2017

Regular Expressions I

Bill Bird

Department of Computer Science
University of Victoria

November 6, 2017

Data Processing (1)

Input

```
Latitude      Longitude      Conditions
  48.465        236.686        Overcast
  48.461       -123.311            Rain
  40.133       -105.282         Flurries
  50.725         15.608            Fair
```

Output

```
Latitude 48.465, Longitude 236.686 - Overcast
Latitude 48.461, Longitude -123.311 - Rain
Latitude 40.133, Longitude -105.282 - Flurries
Latitude 50.725, Longitude 15.608 - Fair
```

- **Easy Problem**: Write a program to convert the table of tab-separated data at the top to the formatted output shown below it.

Data Processing (2)

Input

```
Latitude      Longitude      Conditions
  48.465        236.686        Overcast
  48.461       -123.311            Rain
  40.133       -105.282         Flurries
  50.725         15.608            Fair
```

Output

```
<table><tbody>
<tr style="text-weight:bold">
<td>Latitude</td><td>Longitude</td><td>Conditions</td>
</tr>
<tr><td>48.465</td><td>236.686</td><td>Overcast</td></tr>
<tr><td>48.461</td><td>-123.311</td><td>Rain</td></tr>
<tr><td>40.133</td><td>-105.282</td><td>Flurries</td></tr>
<tr><td>50.725</td><td>15.608</td><td>Fair</td></tr>
</tbody></table>
```

- **Another Easy Problem**: Convert the tab separated input data to an HTML table.

Data Processing (3)

Input

```
Latitude 48.465, Longitude 236.686 - Overcast
Latitude 48.461, Longitude -123.311 - Rain
Latitude 40.133, Longitude -105.282 - Flurries
Latitude 50.725, Longitude 15.608 - Fair
```

Output

Latitude	Longitude	Conditions
48.465	236.686	Overcast
48.461	-123.311	Rain
40.133	-105.282	Flurries
50.725	15.608	Fair

- **Harder Problem**: Convert the formatted output back to a table.

Data Processing (4)

Input

```
<table><tbody>
<tr style="text-weight:bold">
<td>Latitude</td><td>Longitude</td><td>Conditions</td>
</tr>
<tr><td>48.465</td><td>236.686</td><td>Overcast</td></tr>
<tr><td>48.461</td><td>-123.311</td><td>Rain</td></tr>
<tr><td>40.133</td><td>-105.282</td><td>Flurries</td></tr>
<tr><td>50.725</td><td>15.608</td><td>Fair</td></tr>
</tbody></table>
```

Output

Latitude	Longitude	Conditions
48.465	236.686	Overcast
48.461	-123.311	Rain
40.133	-105.282	Flurries
50.725	15.608	Fair

- **Annoying Problem**: Convert the HTML table back to text.

Pattern Matching (1)

Exercise: Write a python function `match(s)` which tests whether the string `s`...

- Starts with 'aa' or 'ee'.
- Consists of one or more words containing only uppercase or lowercase letters.
- Has all occurrences of the letter 'a' come before any occurrence of the letter 'b'.
- Is a single HTML tag (such as '`<br>`' or '`<td style="text-weight:bold">`')
- Contains each of 'a', 'e', 'i', 'o' and 'u' exactly once, in alphabetical order (e.g. '`abstemious`').
- Is a C variable declaration (e.g. '`int x;`' or '`int (*(*A)[10])(int,float);`')

Pattern Matching (2)

```
def startswith_ee_or_aa(s):
  """Returns True if s starts with 'ee' or 'aa'
  Returns False otherwise."""
  return s.startswith('ee') or s.startswith('aa')

>>> startswith_ee_or_aa('aaron')
True
>>> startswith_ee_or_aa('Aaron')
False
>>> startswith_ee_or_aa('eerie')
True
>>> startswith_ee_or_aa('eagle')
False
>>> startswith_ee_or_aa('e')
False
```

- 'Pattern Matching' of a string is simply determining whether or not the string is of a certain form.

Pattern Matching (3)

```
def ContainsOnlyWords(s):
  """Returns True if s consists of one or more words
  containing only uppercase and lowercase letters.
  Returns False otherwise."""
  #Split s by whitespace.
  tokens = s.split()
  #If any token does not contain only letters, return False.
  for token in tokens:
    if not token.isalpha():
      return False
  return True

>>> ContainsOnlyWords('Gregor')
True
>>> ContainsOnlyWords('Gregor Samsa')
True
>>> ContainsOnlyWords('GregorSamsa1915')
False
```

- Often, pattern matching is used to locate interesting information, which can then be extracted for later use.

Pattern Matching (4)

```
def A_before_B(s):
  """Returns True if all occurrences of 'a' in s
  appear before any occurrence of 'b'."""
  found_b = False
  for c in s:
    if c == 'b':
      found_b = True
    if c == 'a' and found_b:
      return False
  return True

>>> A_before_B('apple')
True
>>> A_before_B('banana')
False
>>> A_before_B('cranberry')
True
```

- Ad hoc functions can be written to match particular patterns, but are tedious and cumbersome and often obscure the nature of the pattern itself.

grep (1)

```
$ grep
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep --help' for more information.
$ grep ytho english_words.txt
mythology
python
mythological
mythologies
pythons
$
```

- The Unix grep command is used to search a stream of characters for a provided pattern.
- grep prints all lines containing the provided pattern, with the pattern itself highlighted.

grep (2)

```
#Search for all lines beginning with 'xy'
$ grep ^xy english_words.txt
xylophone
xylophones
#Search for all lines ending with 'ba'
$ grep ba$ english_words.txt
amoeba
tuba
$
```

- The patterns used by grep are **regular expressions**.
- Regular expressions are a compact way to represent many types of patterns.
- In the examples above, the metasymbols '^' and '$' are used to match the beginning and end of a line.

grep (3)

```
#Search for all lines containing 'a', 'b'
#and 'c' separated by one character.
$ grep a.b.c english_words.txt
drawback
drawbacks
barbecue
barbecued
barbecues
barbecuing
playback
$
```

- The metasymbol '.' (dot) will match any character except a newline.
- For example, the pattern 'a...' matches all four character sequences beginning with 'a'.

grep (4)

```
#Search for occurrences of the word 'prince'
#followed by zero or more occurrences of 's'
$ grep princes* english_words.txt
prince
princes
princess
princesses
$
```

- The metasymbol '*' (Kleene Star) matches the previous character zero or more times (with no limit).
- The pattern 'ab*a' will match 'aa', 'aba', 'abba','abbba', etc.
- A common mistake is to forget about the case where * matches zero times.

grep (5)

```
#Search for all lines that begin and end
#with two vowels. Note that '.*' matches any
#sequence of characters (of any length).
$ grep ^[aeiou][aeiou].*[aeiou][aeiou]$ english_words.txt
audio
eerie
eigenvalue
euthanasia
$
```

- The metasymbol `[ ]` can be used to match one of a collection or range of characters.
- The pattern `[aeiou]` matches any vowel.
- Ranges can be specified with a hyphen. For example, `[A-Za-z0-9]` matches uppercase and lowercase letters as well as numerals.

grep (6)

```
#Search for all lines that end in 'ou'
#but contain no other occurrences of 'u'
$ grep ^[^u]*ou$ english_words.txt
you
thou
bayou
caribou
$
```

- Bracket expressions can be inverted by adding the '^' character after the opening bracket.
- The pattern `[A-Z][^A-Z]*` matches any sequence starting with a capital letter which contains no other capital letters, such as '`Gregor`' or '`Fish and chips`' but not '`Gregor Samsa`'.

grep (7)

```
#Search for all lines that contain
#the substring 'ba' before the substring 'ab'
$ grep ba.*ab english_words.txt
debatable
#Perform the same search, but match the entire line.
$ grep ^.*ba.*ab.*$ english_words.txt
debatable
#Search for words which contain all five vowels in
#alphabetical order (split between lines for clarity)
$ cat english_words.txt |
     grep ^[^aeiou]*a[^eiou]*e[^iou]*i[^ou]*o[^u]*u.*$
facetious
$
```

- The name 'grep' comes from the command 'g/*re*/p' (where '*re*' is a regular expression) in the old Unix editor `ed`.
- The 'g/*re*/p' command still works in `ed`-based editors like `vim`.

Regular Expressions in Python (1)

```
>>> import re

>>> m = re.match('[A-Z][a-z]* [A-Z][a-z]*', 'Bela Lugosi')
>>> print(m)
<_sre.SRE_Match object at 0x7f2a575c8988>

>>> m = re.match('[A-Z][a-z]* [A-Z][a-z]*', 'count dracula')
>>> print(m)
None

>>> m = re.match('[A-Z][a-z]* [A-Z][a-z]*', 'Elvis')
>>> print(m)
None

>>> m = re.match('[A-Z][a-z]* [A-Z][a-z]*', 'Bob 4Apples')
>>> print(m)
None
```

- The `re` module provides regular expression support in Python.
- Regular expression syntax is not consistent between implementations; the Python dialect is among the more readable variants.

Regular Expressions in Python (2)

```
import re

def is_non_negative_int(s):
  if re.match('[0-9][0-9]*',s):
    return True
  return False

>>> is_non_negative_int('10')
True
>>> is_non_negative_int('0')
True
>>> is_non_negative_int('-5')
False
>>> is_non_negative_int('Number = 10')
False
```

- The function `re.match(pattern, s)` returns a 'match object' if `s` matches `pattern` and returns `None` otherwise.
- The `re.match` function only finds matches at the start of the provided string.

Regular Expressions in Python (3)

```
>>> import re

>>> S = '10, (11*3), -25,   Number = 10,    0, 6'

>>> re.findall('[0-9][0-9]*',S)
['10', '11', '3', '25', '10', '0', '6']
```

- The function `re.findall(pattern, s)` returns a list of all occurrences of `pattern` in `s`.
- This can often defeat the purpose of the pattern. In the example above, the pattern '`[0-9][0-9]*`' only matches non-negative integers (like 0 or 10), but the value 25 is matched despite appearing as a negative number in the string.

Metasymbols Available in Python (1)

Symbol	Matches
.	Any character except '\n'
^	Start of string
$	End of string
\w	Any alphanumeric character or underscore.
\s	Any whitespace character
\d	Any digit (i.e. [0-9])
\b	A word boundary (including string start, string end and boundary between spaces and alphanumeric characters)
\\	The '\' (backslash) character.

- Programs using POSIX regular expressions (such as `grep`), use a different format for some metasymbols. For example, '\w', '\d' and '\s' are roughly equivalent to '`[[:alnum:]]`', '`[[:digit:]]`', '`[[:space:]]`' in `grep`.

Metasymbols Available in Python (2)

Symbol	Matches
x*	Zero or more occurances of pattern `x`.
x+	One or more occurances of pattern `x`.
x?	Zero or one occurances of pattern `x`.
x{n}	Exactly n repetitions of pattern `x`
x{n, m}	At least n and at most m repetitions of pattern `x`
x|y	One of pattern `x` or pattern `y`.
[abc]	Any character in the set $\{a, b, c\}$.
[^abc]	Any character not in the set $\{a, b, c\}$
(x)	Pattern x (and creates a group capturing the matching text).
(?:x)	Pattern x (without capturing).

- The patterns `x` and `y` in the above table can be any regular expressions (such as 'a' or '(ab*c)').