SI 330: Data Manipulation

Finding Patterns in Text with Regular Expressions

Chris Teplovs (cteplovs@umich.edu)

Some material courtesy of: Kevyn Collins-Thompson, Yuhang Wang, Qiaozhu Mai, Charles Severance, Patrick Dudas

Course announcements

- Data access ethics
  - Data Governance
    - Data Stewardship
    - Data Stewards & Managers
    - Data Management & Classification

- Where we're at: a Memorandum of Understanding is being drafted. I will need to sign this.
- Impact:
  - Extension to Sunday midnight
  - in theory all you should need to do is download a data file and change the name of the file in your code

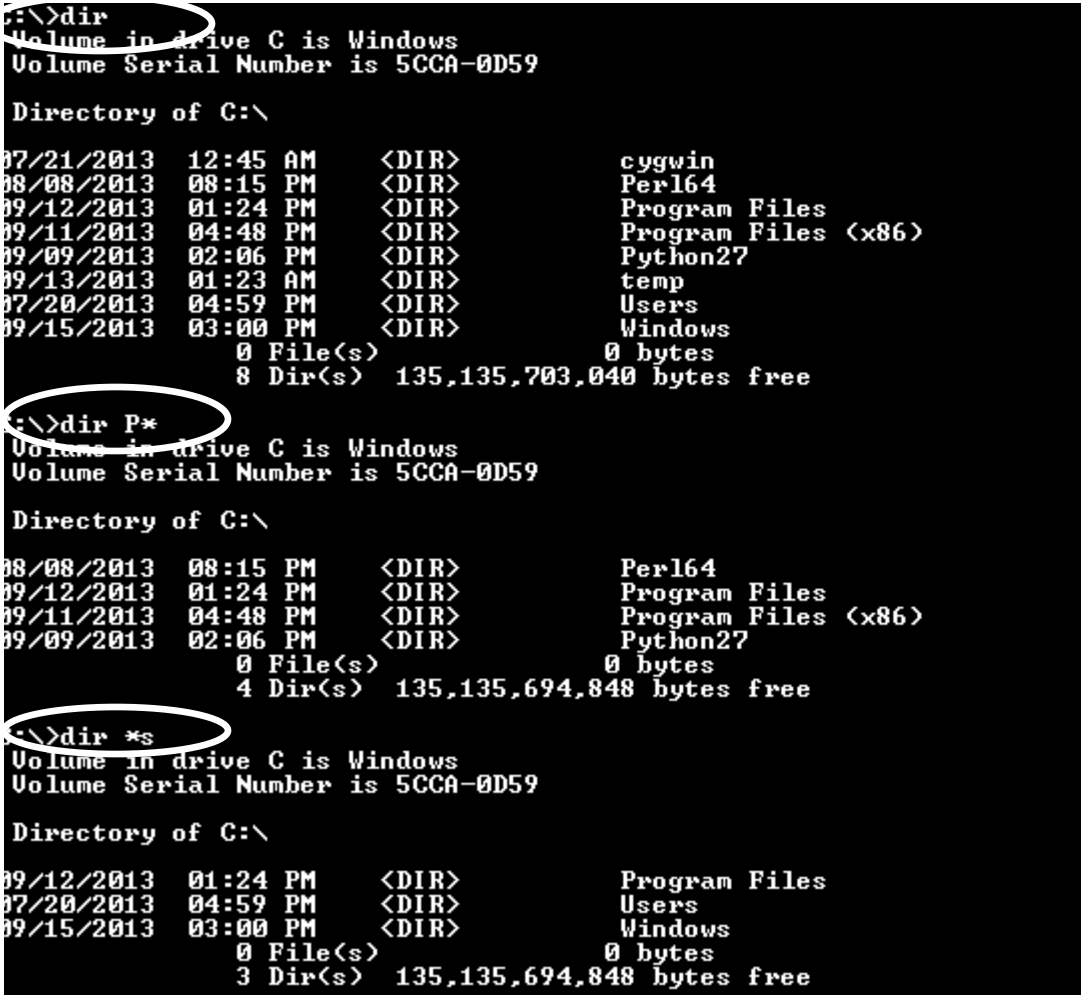
Course Roadmap

Week	Topics
1	Course introduction & review of python basics
2	Basic and compound data structures
3	Extracting patterns from text with regular expressions
4	Fetching and parsing web content

Jupyter notebook for today

- download and open the Jupyter notebook for today – it contains a lot of the code from this slides

You may have seen some kinds of regular expressions before



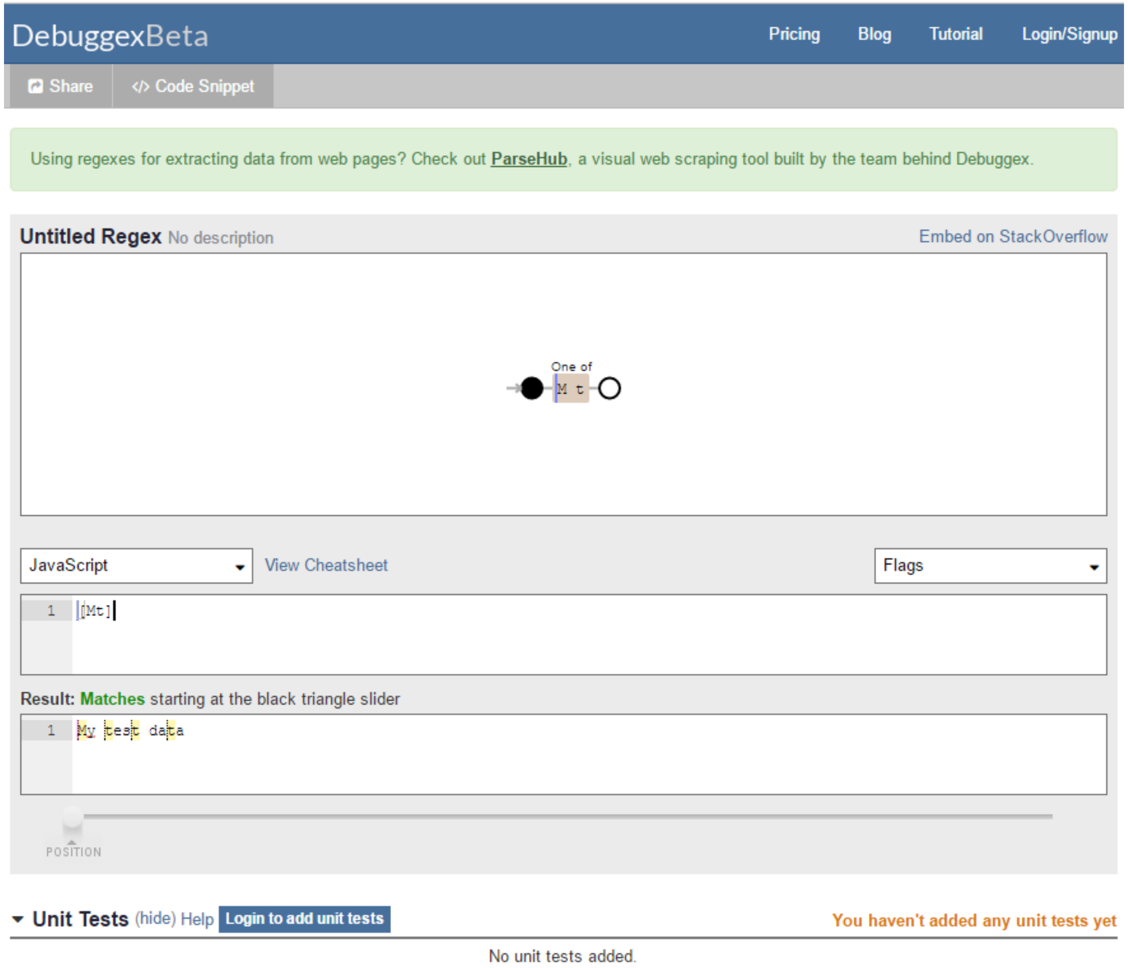
```
:\>dir
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

07/21/2013  12:45 AM    <DIR>          cygwin
08/08/2013  08:15 PM    <DIR>          Perl64
09/12/2013  01:24 PM    <DIR>          Program Files
09/11/2013  04:48 PM    <DIR>          Program Files (x86)
09/09/2013  02:06 PM    <DIR>          Python27
09/13/2013  01:23 AM    <DIR>          temp
07/20/2013  04:59 PM    <DIR>          Users
09/15/2013  03:00 PM    <DIR>          Windows
               0 File(s)              0 bytes
               8 Dir(s)  135,135,703,040 bytes free

:\>dir P*
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

08/08/2013  08:15 PM    <DIR>          Perl64
09/12/2013  01:24 PM    <DIR>          Program Files
09/11/2013  04:48 PM    <DIR>          Program Files (x86)
09/09/2013  02:06 PM    <DIR>          Python27
               0 File(s)              0 bytes
               4 Dir(s)  135,135,694,848 bytes free

:\>dir *s
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

09/12/2013  01:24 PM    <DIR>          Program Files
07/20/2013  04:59 PM    <DIR>          Users
09/15/2013  03:00 PM    <DIR>          Windows
               0 File(s)              0 bytes
               3 Dir(s)  135,135,694,848 bytes free
```

Regular Expressions (or 'regexp' or 'regex')

- A concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters.
- Similar regular expression syntax appears in many other tools
  - grep, flex, editors, ….
  - So you'll be able to re-apply most of what you learn in many other computing settings

Online visual regular expression testers

debuggex.com   regexr.com



Using regexes for extracting data from web pages? Check out ParseHub, a visual web scraping tool built by the team behind Debuggex.

Untitled Regex No description

Embed on StackOverflow

```
[Mt]
```

Result: Matches starting at the black triangle slider

```
My test data
```

You haven't added any unit tests yet

No unit tests added.

The Python `re` module

- `import re`
- Three Python functions:

  `re.search()` finds first occurrence of a pattern anywhere in string

  `re.match()` checks for a match only at beginning of string

  `re.findall()` finds all occurrences of a pattern, not just first one

- Some new regexp-enabled text operations:

  `re.split()`

  ```
  entries = re.split("\n+", text)
  ```

  `re.sub()`

  ```
  re.sub(r"(\w)(\w+)(\w)", repl, text)
  ```

re.search

`search(pattern, string, flags=0)`

Scan through string looking for the first match to the pattern anywhere in the string

Returns:

A MatchObject or None if not found

re.match

`match(pattern, string, flags=0)`

Try to apply the pattern at the start of the string. *Flags will be covered later*.

Returns:

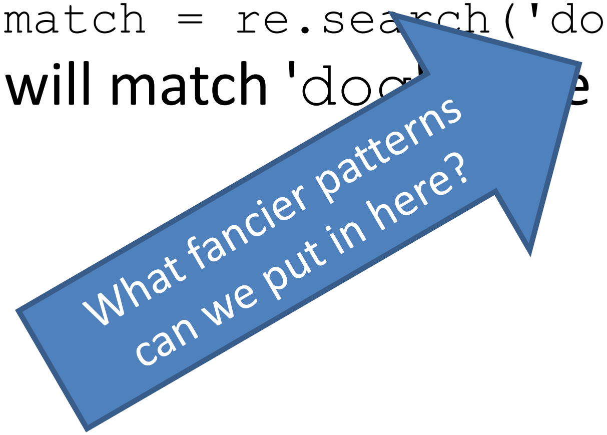
A MatchObject or None if no position in the string matches the pattern

Basic Patterns

- Ordinary characters just match themselves.

```
match = re.search('dog','The lazy dog went to sleep.')
```

will match 'dog' in the right-hand string.



What fancier patterns can we put in here?

https://developers.google.com/edu/python/regular-expressions

A simple example: re methods return match objects

```
import re

str = 'a simple example!'

# want to see if ‘simple’ appears in the
# test string
match = re.search('simple', str)

if match:
    print('found', match.group())
else:
    print('did not find')
```

Returns a match object or None on failure
Important match object methods:
group() start() end() span()

Basic Patterns

- Ordinary characters just match themselves.

```
match = re.search(r'dog','The lazy dog went to sleep.')
```

will match 'dog e right-hand string.

What fancier patterns can we put in here?

- Special rs:

  \ , \r tab, newline, return

  The meta-characters which do not match themselves because they have special meanings are:

  `. ^ $ * + ? { } [ ] \ | ( )`

https://developers.google.com/edu/python/regular-expressions

Very important single-character regular expression symbols

`.` Matches any char except newline `\n` `'F..m:'`

Yes: `Farm:` **Yes**: `Foom:` **No**: `Firm.`

`\s` matches whitespace `'Pine`**`\s`**`apple'`

Yes: `Pine apple` **No**: `Pinesapple`

`\S` matches non-whitespace `'Pine`**`\S`**`pple'`

Yes: `Pineapple` **No**: `Pine pple`

`\d` Decimal digit, 0-9

`\D` Matches any non-digit character.

Very important single-character regular expression symbols

`^` Beginning of the line `'^From: '`

Yes: `From: Chris` **No**: `It said, 'From:…`

`$` End of the line (just before newline) `'Michigan$'`

Yes: `Michigan`**`\n`** **No**: `Michigan, U.S.A.`**`\n`**

Escape character

What if we really want to look for `'$'`?

Use an escape character: BACKSLASH

Examples:

`'\$19\.99'` will match `$19.99`

`'\\folder'` will match `\folder`

Python raw string notation: `r'text'`

- Keeps regular expressions sane
- Without it, every backslash `'\'` in a regexp would need `'\'` prefix
- `r'\n'` is a two-character string containing `'\'` and `'n'`
- `'\n'` is a one-character string containing newline character
- Use `r'\\'` instead of `'\\\\'`

Special commands for finding words (note upper and lower-case versions)

A word is considered a sequence of letters, digits, or underscore (_)
Any other characters are considered to separate words. Which of these are words?

```
my_token_2
my-token-2
734.83
```

`\w` Matches a 'word' character: a **letter** or **digit** or underscore.
Note that although "word" is the mnemonic for this, it
only matches a single word char, not a whole word.

`\W` Matches any non-word character.

`\b` Matches boundary between word \w and non-word \W chars:
`r'py\b' matches 'py', 'py.', or 'py!'`
but not `'python', 'py3', 'py2'`

`\B` Matches NOT at beginning or end of a word.
`r'py\B' matches 'python', 'py3', 'py2'`
but not `'py', 'py.', or 'py!'`

https://developers.google.com/edu/python/regular-expressions

Often we want to search for repeated patterns: Wildcards and matching repetitions

`*`	**Zero or more** of the previous thing
`+`	**One or more** of the previous thing
`?`	**Zero or one** of the previous thing
`{3}`	Matches exactly 3 of the previous thing
`{3,6}`	Matches between 3 and 6 of the previous thing
`{3,}`	Matches 3 or more of the previous thing

Wildcard examples

`ab*` will match

- ‘a’ (must have)
- followed by zero or more 'b's

`ab+` will match

- ‘a’ (must have)
- followed by one or more ‘b’s.
  It will not match just ‘a’.

`ab?` will match

- ‘a’ (must have)
- Followed by zero or one b's

Sets, ranges and alternatives

Specifying a set of characters using `[ ]`

- `[aeiou]` Matches a single character in the given set {a, e, i, o, u}
- `[^aeiou]` Matches a single character NOT in the given set {a, e, i, o, u}

Example:

What substrings does `[aeiou]{2,}` match in

```
The eerie wind said "Oooo" and "Rrr".
```

```
The eerie wind said "Oooo" and "Rrr".
```

Why not O?

How you would use this in Python

```
>>> import re
>>> s = "The eerie wind said Oooo and Rrrr"
>>> match = re.search('[aeiou]{2,}', s)
```

Note that there are multiple matches for this pattern in the text. `re.search` will only find the first one.

`re.findall` will return all strings that match the pattern.

```
>>> re.findall("[aeiou]{2,}", s)
['ee', 'ie', 'ai', 'ooo']
```

The `finditer()` method returns a list of match objects (not just strings)

```
>>> s = "The eerie wind said Oooo and Rrrr"
>>> matches = re.finditer("[aeiou]{2,}", s)
>>> for m in matches:
...   print(m.group())
...
ee
ie
ai
ooo
```

A range of characters can be defined using dash (-) as part of a set `[ ]`

- Valid:

`[A-Z]`	Upper Case Roman Alphabet
`[a-z]`	Lower Case Roman Alphabet
`[A-Za-z]`	Upper/Lower Case
`[A-F]`	Upper Case (only A – F)
`[0-9]`	All Digits \d
`[a-zA-Z0-9_]`	\w

- Invalid:

```
[a-Z]
[F-A]
[9-0]
```

Example application: extracting email spam headers

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

We need to extract numbers from lines with the above syntax.

We don’t just want any floating-point numbers from any email lines.

We can construct the following regular expression to select the lines:

```
^X-.*:\s[0-9.]+
```

Example using multiple operators

```
X-DSPAM-Confidence: 0.8475
```

```
^X-.*:\s[0-9.]+
```

- What does this say?
  - We want strings that start (`'^'`) with `X-`
  - Followed by zero or more of any character `'.*'`
  - Then a colon (`':'`) and a whitespace `\s` char.
  - After the whitespace, look for **one or more** characters
    - That are either a digit (0-9) or a period
- Note that special characters are not active inside ranges, so `'.'` is treated as a period.

Example using multiple operators

`^X-.*:\s[0-9.]+`

Match? `xX-abd:_487.3` No

Match? `X-abd:_487.34.2` Yes

Match? `X-:_.` Yes

Match? `X-abd:_iii.3` No

Using the wildcard in Python with `finditer()`

```
>>> etext = read_email_text("email.txt")
>>> matches = re.finditer(r"^X-.*:\s[0-9.]+", etext)
>>> for m in matches:
...   print(m.group())
...
X-DSPAM-Confidence: 0.8475
X-Mail-Word-Count: 873
X-DSPAM-Confidence: 0.7323
(and more matches…)
```

Negation of Ranges of Regular Expressions

`[^0-9]`	Anything BUT digits
`[^a]`	Anything BUT a lower case a
`[^A-Z]`	Anything BUT upper case letters
`[^,]`	Anything BUT ,

What kind of strings does this match?

`^[^^]`

Match? `^foo` No

Match? `foo^` Yes

Strings that start with a character that is NOT `'^'`

Defining alternatives using the pipe | metacharacter

- `th(is|at|e other)`
  - matches 'this', 'that', or 'the other'
- `tha[nt]|re`
  - matches 'than' 'that' or 're'
- Each alternative can be a regular expression

```
(success | failure code: [0-9]+ | maybe[!?]*)
```

- Pipe is never greedy. As the target string is scanned:
  - REs separated by `'|'` are tried from left to right.
  - When one pattern completely matches, that branch is accepted.
  - This means that once A matches, B will not be tested further.
  - Even if it would produce a longer overall match.
- What does this match?

  `^(T|t)oday`

Group Extraction: A more sophisticated type of match

Problem:

Often you want to extract parts of the matching text for later use. e.g. find email addresses, and extract user and hostname.

Solution: Use parentheses to create groups showing the parts you want to save for later.

```
str =  'My email addre Group 1 anta@ Group 2 edu. Hohoho.'
match = re.search(r'([\w.-]+)@([\w.-]+)', str)
if match:
  print(match.group())    # the whole match
  print(match.group(1))   # the username part
  print(match.group(2))   # the hostname part
```

`re.findall` with groups

```
findall(pattern, string, flags=0)
```

- Returns a list of all non-overlapping matches as a list of strings
- If one or more groups are present in the pattern, return a list of groups.
- This will be a list of tuples if the pattern has more than one group.
- Empty matches are included in the result.

findall() Example

```
str = 'I have two email addresses: santa@umich.edu \
and santa@northpole.org. Hohoho.'

# Here re.findall() returns a list of all the found
# email strings
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str)
```

findall returns a list of strings.

```
['santa@umich.edu', 'santa@northpole.org']
```

findall() and Group Extraction

```
str = 'I have two email addresses: santa@umich.edu \
and santa@northpole.org. Hohoho.'

# Here re.findall() returns a list of all the found
# email strings
emails = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
email[0] = ('santa', 'umich.edu')
email[1] = ('santa', 'northpole.org')
```

finditer() and Group Extraction

`m.group(0)`	The <u>entire</u> match string	`santa@umich.edu`
`m.group(1)`	The first group	`santa`
`m.group(2)`	The second group	`umich.edu`

```
str = 'I have two email addresses: santa@umich.edu \ and
santa@northpole.org. Hohoho.'

>>> matches = re.finditer(r"([\w\.-]+)@([\w\.-]+)", str)
>>> for m in matches:
...   print("first group: "+m.group(1)+", second group: "+m.group(2))
...
first group: santa, second group: umich.edu
first group: santa, second group: northpole.org.
```

re.search() returns a match object, so groups work there too

m.group(0)	The entire match string	santa@umich.edu
m.group(1)	The first group	santa
m.group(2)	The second group	umich.edu

```
str = 'I have two email addresses: santa@umich.edu \ and
santa@northpole.org. Hohoho.'

>>> m = re.search(r"([\w\.-]+)@([\w\.-]+)", str)
>>> print("first group: "+m.group(1)+", second group: "+m.group(2))
first group: santa, second group: umich.edu
```

Advanced matching: more subtle ways to modify searching

- Greedy vs. non-greedy matching
- Zero-width lookahead

Greedy Matching is the Default

- Python always tries to match as <u>much</u> as possible.
- Example:

```
str = 'the cat in the hat'
match = re.search(r'^(.*)(at)(.*)$', str)
```

Now, what do we have in

`match.group(1), match.group(2), match.group(3)?`

```
'the cat in the h'
'at'
''
```

Non-greedy Matching:
Add an extra ? To your wildcard

- Non-greedy versions try to match as minimally as possible.

  `??` ,`*?` , `+?`, and `{}?`

- Example 1:

```
x = 'the cat in the hat';
 match = re.search(r'^(.*?)(at)(.*)$', str)
```

Now, what do we have in

```
match.group(1), match.group(2) and match.group(3)?
 'the c'    'at'   ' in the hat'
```

- Example 2: `<H1>title</H1>`

  `<.*>` will match the whole string .

  `<.*?>` will match `<H1>`

Very useful power: You can refer back to an earlier group match within the same regular expression. How?

- \N where N is the group number
- \1 matches group 1 result

Example:

```
r'<(.*?)>(.*?)</\1>'
```

Matches tag pairs with matching begin/end tags

```
 <X>foobar</X>    <ABCD>baz</ABCD>
```

Stop and look ahead (without adding to the current match): zero-width matching

- Problem:
  - We want to match any single character *q* that is not followed by *u*?
  - Why not use `q[^u]`

    Means: `q` followed by a character that is not a `u`

  `Iraqi population`

  `q[^u]` returns `qi` (q followed by i). This is two characters.
- What's the problem?
  - The regexp matcher has just 'used up' the `i` as part of this match and is now looking past it, at the 'space' character.
- But the 'i' may be important in an upcoming regexp match
  - Solution: check for the presence of 'not u' without letting regexp 'eat' it…
  - You do this by using a zero-width *negative lookahead assertion* **`q(?!u)`**
- Assertions do not 'use up' characters: they are zero-width, like start/end of line, or start/end of word
- This will match the single character *q only,* not trailing letters

Other types of zero-width assertions

- *Negative lookbehind assertion*:

  (?<!`abc`**)**`def` will not match `abcdef`, but will match `acbdef`
- *Positive lookbehind assertion* **(?<=**`abc`**)**`def` will first match `def`, then back up 3 characters and check for the contained pattern `abc`.
- What does `(?<=-)\w+` do?
  - Matches a word preceded by a hyphen

```
m = re.search('(?<=-)\w+','hard-boiled')
m.group(0): 'boiled'
```

Options

- The option flag can be added as an extra argument to search(), findall() etc.,
  - e.g. `re.search(pat, str, re.IGNORECASE)`

- `re.IGNORECASE` Ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
- `re.DOTALL` Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.
- `re.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.
- `re.UNICODE` Match against Unicode strings: invoke Unicode character properties for word-vs-nonword characters, etc.

Substitution

```
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.
- repl can be either a string or a callable.
- If a string, backslash escapes in it are processed.
- If it is a callable, it's passed the match object and must return a replacement string to be used.

Substitution Example

```
str = 'My email is santa@umich.edu. Hohoho.’

print(re.sub(r'@[\w\.-]+', '@northpole.org', str))
# prints out My email is santa@northpole.org Hohoho.
```

Compile regex Patterns

- If a regex pattern is going to be reused, it is a good idea to compile it first.
- Example:

```
p = re.compile('\d+')
# search demo
m = p.search('12 drummers drumming, 11 pipers piping, 10
lords a-leaping')
if m:
  print('Match found: ', m.group())
else:
  print('No match’)
# findall demo
print(p.findall('12 drummers drumming, 11 pipers piping,
10 lords a-leaping'))
```

What you should know

- How to write useful types of text matching patterns as regular expressions
- How to specify and extract groups in a match
- How to use the python `re` library functions to search and extract all matches in a text

Review Resources

- Readings:
  - Severance, Chapter 11
- Excellent free online tools for debugging your tricky regexps:
  - https://www.debuggex.com/
  - http://regexr.com
- Other references:
  - https://developers.google.com/edu/python/regular-expressions
  - http://docs.python.org/3/howto/regex.html
  - http://docs.python.org/3/library/re.html

Next week sneak peak: XML, JSON, Web APIs

highway
regulatory
protection
security
revitalization
defense
workforce
school
cybersecurity
authorization
insurance
agricultural
standards
industry
consumer
tourism
regarding
construction
partnership
coverage
veterans
treatment
regulation
investment
physician
reimbursement
domestic
advocate
safety
legislative
agriculture
pertaining
products
transparency

```
<?xml version='1.0' encoding='UTF-16'?>
<PublicFilings>
<Filing ID="306B3144-3E4F-48CF-98F1-C6BF455B6A6B" Year="2012" Received="2013-01-01T00:58:03.067"
Amount="15000" Type="FOURTH QUARTER REPORT" Period="4th Quarter (Oct 1 - Dec 31)"><Registrant xmlns=""
RegistrantID="6848" RegistrantName="Marshall Brachman" GeneralDescription="lobbyist" Address="634 A
Street, NE&#xD;&#xA;Washington, DC 20002" RegistrantCountry="USA" RegistrantPPBCountry="USA" /><Client
xmlns="" ClientName="ADAMS COUNTY COLORADO" GeneralDescription="" ClientID="12" SelfFiler="FALSE"
ContactFullname="MARSHALL BRACHMAN" IsStateOrLocalGov="TRUE" ClientCountry="USA" ClientPPBCountry="USA"
ClientState="" ClientPPBState="" /><Lobbyists><Lobbyist xmlns="" LobbyistName="BRACHMAN, MARSHALL A"
LobbyistCoveredGovPositionIndicator="NOT COVERED" OfficialPosition=""
/></Lobbyists><GovernmentEntities><GovernmentEntity xmlns="" GovEntityName="SENATE" /><GovernmentEntity
xmlns="" GovEntityName="Federal Aviation Administration (FAA)" /><GovernmentEntity xmlns=""
GovEntityName="HOUSE OF REPRESENTATIVES" /></GovernmentEntities><Issues><Issue xmlns=""
Code="BUDGET/APPROPRIATIONS" SpecificIssue="DERA funding regarding Rocky Mountain Arsenal&#xA;DoD
Appropriations for the above&#xA;TTHUD funding for railroad grade separation&#xA;funding for contract
tower program and commercial flight program" /><Issue xmlns="" Code="DEFENSE" SpecificIssue="DERA funding
regarding Rocky Mountain Arsenal&#xA;DoD Authorization" /><Issue xmlns="" Code="NATURAL RESOURCES"
SpecificIssue="land trade issues regarding the Rocky Mountain Arsenal NWP" /></Issues></Filing>
```

Extra slides