Week 6: Processing Text with Python and Unix commands for data analysis

The volume of unstructured data (e.g.text) is growing immensely in the past decade. Unstructured data does not have a pre-defined model and it cannot fit into the relational database tables. Quite often, the data scientist or data analyst will have to extract some specific or interesting information such as:

- analyzing social media trends to identify the current trending or "viral" potential.
- web scraping websites, and extract topics, themes, and sentiment levels.

Basics about strings

The String is a basic type in Python. To compose string literals in Python, either a single or double quotes can be used as far as opening and closing characters. Though, triple quote is used to define a string when line breaks or embedded quotes include in your string.

Example:

```
>>> str = """ Twinkle, twinkle, litter star,
... How I wonder what you are.
... Up above the world so high
... Like a diamond in the sky."""
>>> print str
 Twinkle, twinkle, litter star,
How I wonder what you are.
Up above the world so high
Like a diamond in the sky.
```

The letter "r" that precedes either single-quoted or triple-quoted strings, signifies that the special characters, such as newline, should not be interpreted by Python.

Example:

```
>>> str = r"little star,\n How I wonder \n what you are\n"
>>> print str
litter star,\n How I wonder \n what you are\n
```

Standard Module: string

The string module is generally faster than the regular expression (re) module and it is easier to understand and implement.

Example:

```
>>> import string
>>> str = "Twinkle Twinkle litter star"
>>> string.capwords(str)
'Twinkle Twinkle Litter Star'
>>> string.replace(str,'Twinkle','Flickle')
'Flickle Flickle litter star'
```

Example:

```
>>> import string
>>> str = "twinkle twinkle little star"
>>> string.count(str,'l')
4
```

Standard functions in Python to work with strings include:

- split()
  break up a line into a list using specific character or spaces
- strip(), lstrip(), rstrip()
  strip white space off both sides, the left, and the right side of the string
- upper(), lower()
  convert letters into upper or lower case
- startswith()
  check the starting of string with a specified pattern
- endswith()
  check the ending of string with a specified pattern
- find()
  return index of given pattern or string (or -1)

Examples:

```
>>> a = "       test string     "
>>> stripA = a.strip()
>>> stripA
'test string'

>>> stripA.split()
['test', 'string']

>>> stripA.upper()
'TEST STRING'

>>> stripA.startswith('es')
False

>>> stripA.startswith('te')
True

>>> stripA.endswith('st')
False

>>> stripA.endswith('ng')
True

>>> stripA.find('in')
8
```

Standard Module: re

A regular expression (RE) is a pattern that defines class of strings. For a given string, we can test whether that string belongs to the class patterns. A regular expression is a concise way to describe pattern that may occur in a text. Regular expression is a great tool for text pattern matching. The regular expression roots in automata and formal language theory. It was later integrated in Unix utilities such as grep, sed, and awk. Regular expressions have been recognized for compactness and flexibility.

Regular expressions are embedded within Python and available through the package *re* (https://docs.python.org/2/library/re.html). Access this package by *import re*

Note that, Regex and regsub modules are used in older Python code.

Metacharacters are special characters that signify things should be matched. The complete list of metacharacters and their meanings can be found at (https://docs.python.org/3/library/re.html#re-syntax):

- . (dot) match any character except newline
- ^ (caret) match the start of the string
- $ match the end of the string or just before the newline at the end of string
- * match zero or more of the preceding RE pattern
- + match one or more of the preceding RE pattern
- ? match zero or one of the preceding RE pattern
- *?, +?, ?? match in minimal or non-greedy fashion (as few characters as possible)
- {m} match exactly m copies of the previous RE pattern
- {m,n} match from m to n repetitions of the preceding RE pattern
- {m,m}? match from m to n repetitions of the preceding RE pattern, and try to match as few repetitions as possible (non-greedy)
- \ - special character (match characters like '*', '?', etc) ???
- [ ] – indicate a set of characters.
- [n-m] match any characters in the range from m to n if the normal character set (can apply only characters and numbers)
- \d match any decimal digit [0-9]
- \D match any non-digit character [^0-9]
- \w match any alphanumeric character and underscore [a-zA-Z0-9_]
- \W match any non-alphanumeric character [^a-zA-Z0-9_] this is the opposite of \w
- \s match whitespace characters including [\t\n\r\f\v]
- \S match non-whitespace characters. This is the opposite of \s [^\t\n\r\f\v]

Simple Regular Expressions

The Module Contents describes functions, constants and exceptions. For a complete list, please refer to: https://docs.python.org/3/library/re.html#re-syntax

re.compile (pattern, flag=0) compile regular expression into regular expression object to be used in match() and search() methods.

re.search (pattern, string, flags=0) looks for the first location of string where regular expression generates a match, and return the corresponding match object. Return none, if no position matches the pattern. (search anywhere within the string)

re.match (pattern,string, flags=0) check for a match at the beginning of string. Return none, if the string does not match the pattern.

re.findall (pattern, string, flags=0) scan string from left to right and return all non-overlapping matches of pattern in string as a list of strings.

re.sub (pattern,repl, string, count=0, flags=0) replace the leftmost non-overlapping occurrences of pattern in string by *repl*. If pattern is not found, return unchanged string.

re.split (pattern, string, maxsplit=0, flags=0) split a string into a list delimited by the given pattern.

re.finditer() match objects instead of strings.

To enter a regular expression into a python RE method, you can either use quotes or with precede quotes with ***r***.

Example: look for characters from a selection of possibilities.

```
>>>import re
>>> text  = "ant any and ankle"
>>> pat = re.compile(‘an[ty]’)
>>> pat.findall(text)
['ant', 'any']
```

Example: similar to previous example but ignore case

```
>>> text = "Ant aNy and ankle"
>>> pat = re.compile("an[ty]",re.IGNORECASE)
>>> pat.findall(text)
['Ant', 'aNy']
```

Example, find all adverb;

```
>>> text = "He completely does not understand why she absolutely refuse to stay"
>>> for k in re.finditer(r"\w+ly", text):
    ...      print('%2d-%2d: %s' % (k.start(),k.end(), k.group(0)))
```

...
3-13: completely
42-52: absolutely

Example script
This following line is an example of Apache (Unix) log file sample from:
http://www.monitorware.com/en/logsamples/apache.php

64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/Main/Double_bounce_sender?topicparent=Main.ConfigurationVariables HTTP/1.1" 401 12846

The log file retains page requests history (who, when, what). The web server log file is a standardized text file format maintained by W3C. There are many proprietary formats that exist. The given example is known as Common Log Format or NCSA Common log format using the following syntax: http://en.wikipedia.org/wiki/Common_Log_Format

1) IP address of the host name (client) which made the request
2) Client identity ('-' means missing data)
3) Userid of the person requesting the document ('-' means missing data)
4) The Time that server finished processing the request (e.g. Date, month, year and time zone)
5) GET is the request line from the client in the double quotes
6) HTTP status code that sends back to the client (e.g. 2xx is a successful response, 3xx redirection, 4xx client error, 5xx server error)
7) Size (bytes) of objects returned to the client

The following script shows number of bytes and total request in the log file:

```
#!/usr/bin/env python
import re    # import the regular expressions modules for matching patterns
import sys   #import system modules so we can read /write from std input/error stream

logfile = open("access_log.txt")
total_req = 0
total_bytes = 0

for line in logfile:
        bytestr = line.rsplit(None,1)[1] #strip leading and trailing whitespace
        if bytestr != '-':
                total_bytes += int(bytestr)
        total_req += 1

print 'Bytes requests: %i' %total_bytes
print 'Total request: %i' %total_req
```

Can you develop scripts for the following tasks?

- Finding the set of all documents that are broken (404)
- Finding all requested documents that transferred over a megabyte
- Finding the largest document

Unix command for data analysis

The Unix Shell offers a large set of command line utilities that can be used for data analysis. Brown (2012) illustrated how to use Unix for exploratory data analysis to inspect, reshape, enumerate, describe, and visualize the data. The Unix commands are simple but yet powerful to do the job quickly as introduced here.

Back in 1969 at Bell Labs, Ken Thomson, Dennis Ritchie and others developed a time-sharing system that was to become UNIX. Linux is the open source offspring of Unix. Basically, Unix tasks are divided into:

1) interaction between OS and computer (kernel)
2) interaction between OS and user (the shell)

There are several shells available such as sh, csh, tcsh, bash, etc.

The commands that let you enter from the keyboard and control the computer is known as the command line interface (CLI) or shell. The shell provides many utilities and mini-languages to interface with your computer's OS and automate tasks. The common format of the shell utilities are:

Utilityname [options] [arguments]

Where, Utilityname is the name of a utility (e.g. head, tail, cut, sort). The options define the program behavior, and are often optional. The arguments are the external input to the program.

The term ***stream*** is a general term referred to data elements available over time. There are three standard stream channels on most of machines, namely: standard input (STDIN), standard output (STDOUT), and standard error (STDERR). These streams are used as the communication methods between one process to another process or computer to another machine.

- STDIN: can be text or binary data steaming. By default, the stdin is connected to your keyboard.
- STDOUT: this is where the program writes its data. A terminal display or a screen is the default STDOUT.
- STDERR this is the stream where error messages are printed. The screen is the default STDERR. It is similar to STDOUT but used for printing or debugging errors.

Here, we will use the shell referred to as Bourne-again or bash shell. It is important for the modern data scientist to facilitate data analysis using these powerful utilities.

A pipe is written with a vertical bar (|) provided by Unix as a way to connect standard output of one process to the standard input of another process. This is also known as a filter or a program that processes a stream and producing another stream. The utilities (name) can be combined using these pipes to form a pipeline.

For example, $ cat data.csv | cut –d, -f3 | sort | uniq –c
Explanations:
A file, data.csv is read in and printed to standard output using 'cat'. The pipe "|" redirects the standard out from cat to standard in of 'cut'. 'cut' extracts the third column, the results are passed to 'sort' to order the results. Finally, the results from 'sort' are passed to 'uniq' for counting the number of times each word appears in the output.

Utility name: 'cat' filename
Purpose: print specified filename to stdout (default is terminal)

Utility name: 'cut'
Purpose: extract the third field (-f3) of the comma delimited (-d) file

Utility name: 'sort'
Purpose: order the elements

Utility name: uniq
Purpose: count unique occurences in consecutive lines of text (therefore, it is important to sort data first, otherwise, the same text can appear more than once since same text might not appear in consecutive lines)

Please remember that the manual files or help invoked by 'man' or 'info' command such as:
$ man sort
$ info sort

Unix filter programs frequently used are: cat, cut, grep, sort, uniq, head, tail, sed, and awk. The last two commands can be used to create more sophisticated filters. By employing these filters, data scientists can get a quick overview of the dataset [Zuther, Bernd]

cat: output or concatenate file(s) to the standard output (e.g. screen), or print to a new file.
Format: cat [options] [filenames]
$ cat file1
Explanation: print the contents of file1 on the screen

$ cat file1 file2 > newfile
Explanation: combine these two files (file1,file2) in the given order, and then create a new file

$ cat file1 file 2 | less
Explanation: send data to another program (e.g. less). 'less' is a utility for viewing the contents of a (large) file. Through the file, it has the capability to go both forward and backward. The saying "less is more" is the implication of 'less' with a better functionality than 'more'.

cut: extract the segments of line from the file, columns of a file. It is a useful command for working with tabular data and combined with head/tail.

Format: cut [options][filenames]
Where options include -b (byte),-c(character),-f(field), -d(delimiter e.g. comma, tab is the default), and ranges (N : until end of the line, N-M : specified range starting from 1, -M : from beginning of the line to M)

$ cut –c 5-10 file1
Explanation: Get the fifth through the tenth characters from each line of the file

$ cut –d " ," –f 3- file1
Explanation: Use comma as the field delimiter, and get the third field through the end of the line.

head : examine the first few lines of a file or first part of the file(s). The default is the first 10 lines. This command is helpful for debugging and inspecting files. An example data set can be download from: https://github.com/comsysto/city-country-dataset

Suppose the data is named city.txt:

$ head city.txt
Explanation: show the first 10 lines of the file

$ head -20 city.txt
Explanation: show the first 20 lines of the file (same as head –n20 city.txt)

$ head city.txt country.txt
Explanation: display head of multiple files

$ head –c100 city.txt
Explanation: display the first 100 characters of the file

tail: examine the bottom of a file or last part of the file(s). The default is the last 10 lines. This command is useful debugging, inspecting, and get subset of files. It can be combined with other commands such as head.

$ tail city.txt
Explanation: display the last 10 line of the file

$ tail –n+5 city.txt | head
Explanation: begin at the fifth line of the file

$ head –n 200 city.txt | tail –n 30
Explanation: take out some part of the file

$ (head -5; tail -5) < city.txt
Explanation: display the first 5 lines and the last 5 lines of the file

sort : order lines of text files
$ sort city.txt | less
Explanation: sort by default

$ sort –r city.txt | less
Explanation: reverse the order (with –r option)

$ sort –t "," –k 3 city.txt | less
Explanation: use comma as the delimiter and sort using the third field as a key

paste: put/concatenate data together by column. This command is helpful for reshaping the data.
$ head –n 100 city.txt | cut –d "," –f 2 > temp1
Explanation: from the first 100 lines, get only the second field by using comma as a delimiter (save the results in temp1 file)

$ head –n 100 city.txt | cut –d "," –f 5 > temp2
Explanation: from the first 100 lines, get the fifth field using comma as delimiter (save the results in temp2 file)

$ paste temp1 temp2 |less
Explanation: concatenate temp1 with temp2 by column (get the second and fifth column of the file)

uniq : is used to analyze data with duplicated elements. Typically, it used in pipeline with the command 'sort'. The 'uniq' only compares with the immediately preceding line whether it is the same, if so the duplicated line is not written. Thus, sort is need prior to the uniq command to get the distinct record only once.

$ tail –n+100 city.txt | cut –d "," –f 2,5 | sort | uniq -c
Explanation: Start from the 100^{th} line, extract the second and the fifth field using comma as the delimiter. Then sort and get rid of (consecutive) duplicated lines and count (with –c option) for each distinct entry

grep (**g**lobally search a **r**egular **e**xpression and **p**rint : g/re/p) is typically used to search for strings in a file. This is useful for text processing. Other applications of grep include matching regular expressions in files, search for files, search and filter files, display

number of lines before (-A), after (-B), around (-C) the search string, count number of lines matches the pattern (-c), ignore case search (-i), check for full words not sub-strings (-w), invert match (-v), show line number that match (-n), etc.

Format: grep “string” filename
$grep –B 3 –A 2 “Denver” city.txt
Explanation: display 3 lines before (-B) the match string “Denver”, and display 2 lines after (-A) the match “Denver”

$grep “Da.*as” city.txt
Explanation: search all string patterns that start with “Da” and end with “as”

$grep –v –c “USA” city.txt
Explanation: count how many lines that do not match the pattern

$grep –n “main” *.py
Explanation: display the line number (-n option) of the given string “main” in all files with extension .py

$grep “Da.*as” city.txt | cut –d, -f 2 | sort | uniq | wc –l
Explanation: extract second field separated by comma, sort, eliminate duplicate, count number of lines, which contains a string beginning with “Da” and ending with “as”.

$export GREP_OPTIONS = ‘—color=auto’
Explanation: highlight of the “match” search

‘sed’ (**s**tream **ed**itor) is a command line utility which is a text stream editor. This command is helpful for making changes (search and replace) within a file. It is known for its simplicity, speed and can handle large files.
Format: sed ‘s/search_pattern/replace_pattern/’ input file

Example: sed ‘s/Dallas/BigD/g’ city.txt > temp
Explanation: replace every occurrence of “Dallas” with “BigD” in the city.txt file and save the result in temp file. Note that, ‘s/Dallas/BigD’ without ‘g’ would replace only the first match. In addition, sed writes the output on STDOUT by default. Thus, the original file has not been modified.

Example: $ echo “http://www.regis.edu” | sed ‘s/edu/org/’
Explanation: replace edu with org (use sed ‘s/edu/ /’ to replace edu with empty space)

Example: $ sed –n ‘1,10p’ city.txt
Explanation: print 10 lines

Example: $ sed ‘1,50d’ city.txt
Explanation: delete the first 50 lines

Example: $ sed ‘2,7!d’ city.txt
Explanation: delete everything except the 2nd to the 7th line

Example: $ sed ‘/^root/d’ testing
Explanation: delete all lines starting with root in the testing file (use sed ‘/sh$/d’ to delete all lines ending with sh)

Suppose, the phone.txt consists of
1112223333
2223334444
3334445555
Example: $ sed -e 's/^[[:digit:]]\{3\}/(&)/g' \
>   -e 's/)[[:digit:]]\{3\}/&-/g' phone.txt
Examplanation: apply with multiple sed commands (Note that, \{3\} means match regular expression three times), and the final results are:
(111)222-3333
(222)333-4444
(333)444-5555

For comprehensive examples, follow this URL: http://sed.sourceforge.net/sed1line.txt

‘awk’ (**A**ho, **W**einberger and **K**ernighan): is a minilanguage to handle simple data formatting with a few lines of codes. It has all ‘grep’ functionality plus the add-on logical, numerical (e.g. cos, exp, sqrt), and string functions (e.g. Gsub, Length, Split, Substr, Tolower) capability. If use appropriately, awk can be a powerful tool for pre-processing data, data conversion. Nevertheless, syntax of awk can be complicated. Therefore, simple python scripts become handy.

Format: awk (search_pattern) {actions}

If the input record matchs the pattern, the action is employed. For awk, a file is divided into records and fields. By default, each line represents a record. Fields are delimited by a special character. White space is the default delimiter. Use ‘$’ to access fields such as ***$1*** is the first field, ***$3*** is the third field. But ***$0*** is the special field representing the entire line. The special variable ***NF*** is equal to the number of fields in the current record, and ***NR*** is the current record number.

If the search pattern is empty, it signifies match everything.
Example: $ awk ‘{print $0}’ phone.txt OR $ cat phone.txt | awk ‘{print $0}’
Explanation: The above command will print every record.

Example: $ awk -F, '{sum+=$5} END {print sum}' city.txt
Explanation: specified comma as the input field separator (-F,), sum the fifth field, and print the sum result

Example: $ awk -F, 'BEGIN { OFS="," } {if ($3 == "USA") { $3 = "United States" } print $0}' city.txt

Explanation: specify comma as the input field separator (-F,), specify comma as the output field separator (OFS=",,"), change the third field from "USA" to "United States", then print the whole line. Note that, space is the field delimiter by default. Without specifying (OFS=",,"), the space will be used (the comma will be missing in the replaced records)

The format of text files on Windows and Unix is slightly different. The Unix uses a line feed (\n) for ending line, whereas, Windows carriage return (\r) ASCII characters to end the lines. https://kb.iu.edu/d/acux

To convert a window file to Unix file
$awk '{sub("\r$", ""); print }' winfile.txt > unixfile.txt

To convert a Unix file to window file
$awk 'sub("$", "\r")' unixfile.txt > winfile.txt

More about awk: http://en.wikibooks.org/wiki/AWK or http://www.faqs.org/faqs/computer-lang/awk/faq/

wc : count number of lines, words, characters. This command is useful for debugging and confirmation of the file sizes
$ wc city.txt
Explanation: display number of lines, words, and character of the file

$wc –l city.txt
Explanation: display on number of lines

Reference:

Beazley, M.D. (2008). *Generator Tricks for Systems Programmers*. Retrieved from: http://www.dabeaz.com/generators-uk/

Bird, S., Klein, E. and Loper, E. (2009) *Natural Language Processing with Python*. O'Reilly Media. Retrieved from: http://www.nltk.org/book/

Brown, S. (2012). *Explorations in Unix*. Retrieved from: http://www.drbunsen.org/explorations-in-unix/

Gunawardema, A. (2009). *Regular Expressions*. CMU. Retrieved from: http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture20-1.pdf

Knowledge Base: *How do I convert between Unix and Windows Text files?* Indiana University. Access data 01/15/2015. Retrieved from: https://kb.iu.edu/d/acux

Langmore, I. & Krasner, D. (date) *Applied Data Science*. Retrieved from: http://columbia-applied-data-science.github.io/appdatasci.pdf

Mertz, D. (2010) *Charming Python: Text Processing in Python*: Retrieved from: https://www.ibm.com/developerworks/library/l-python5/

Pement, E.(2005) (Compiled by).*Useful One-Line Script SED (Unix stream editor)*. Retrieved from: http://sed.sourceforge.net/sed1line.txt

Reda, G. (2013) Useful Unix Commands for Data Science: Retrieved from: http://www.gregreda.com/2013/07/15/unix-commands-for-data-science/

Srinivasan, B. (nd). *The Linux Command Line*. class.coursera.org/startup-001. Stanford University. Retrieved from: http://www.academia.edu/7680615/Lecture_slides_lecture4a-linux-command-line

Zuther, B. (2013). *Data Analysis with the Unix Shell*. Retrieved from: http://blog.comsysto.com/2013/04/25/data-analysis-with-the-unix-shell/