Practical Text Processing

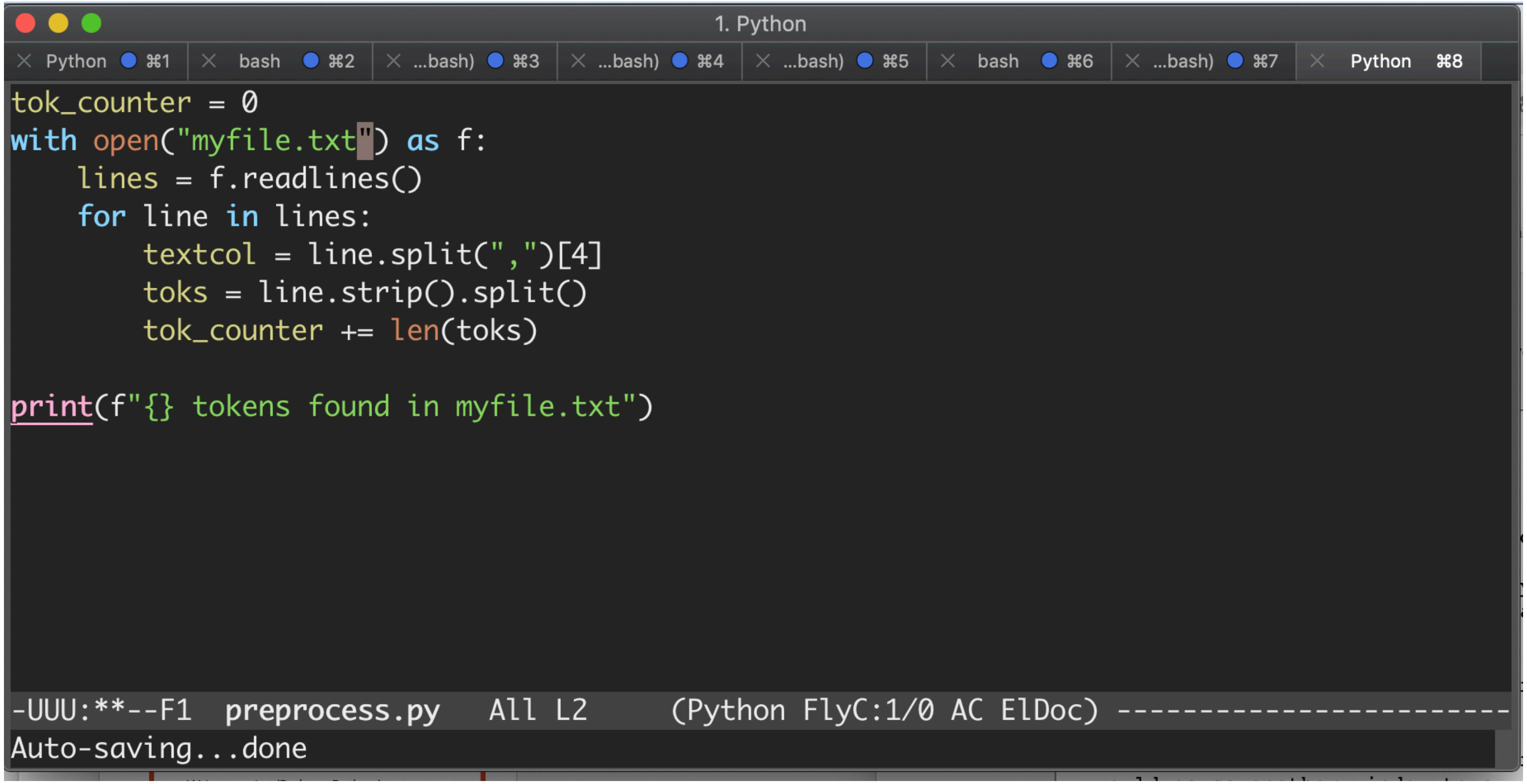
CS-585

Natural Language Processing

Derrick Higgins

Tools for text processing

- Write a script (Python, Perl, …)



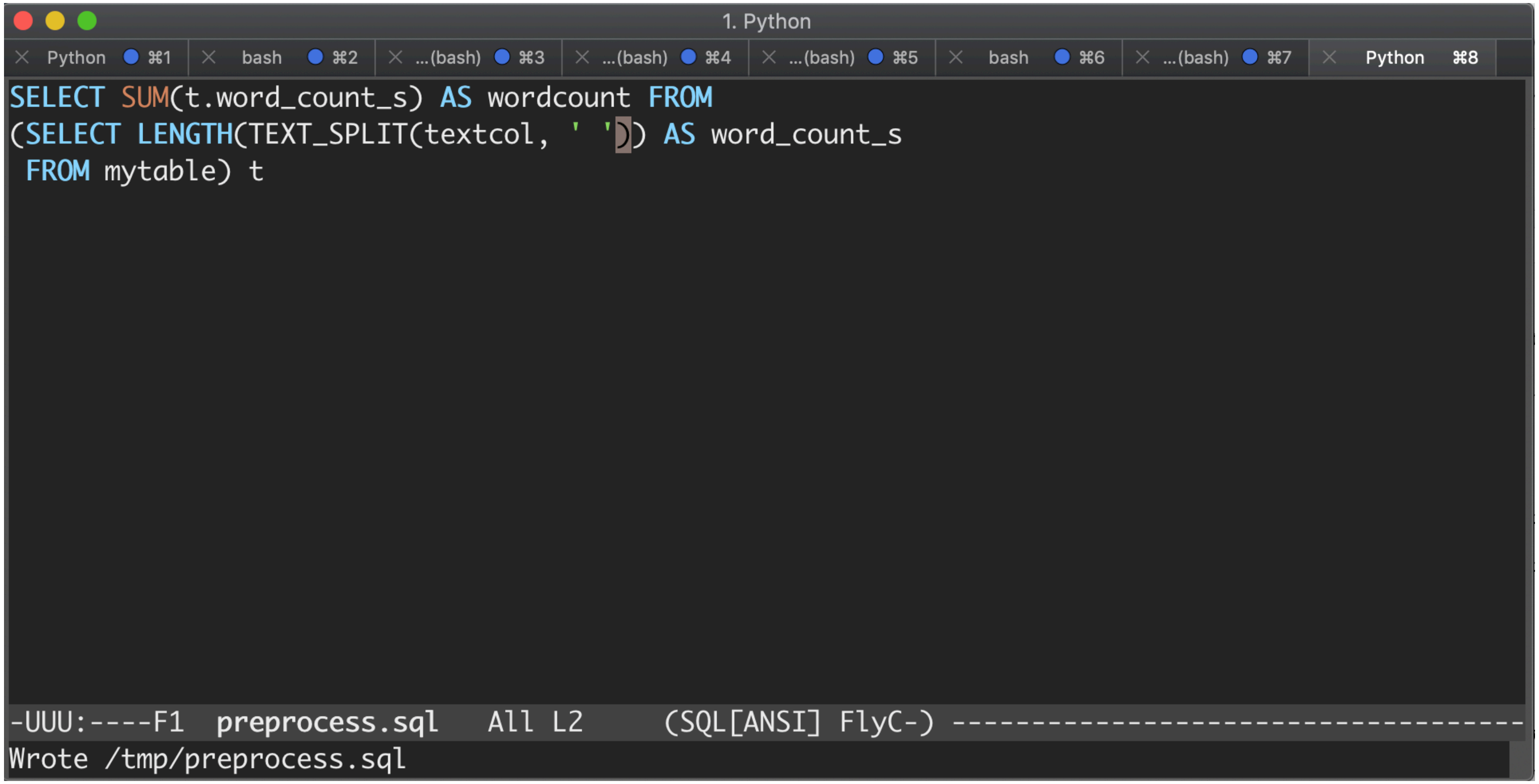
```
tok_counter = 0
with open("myfile.txt") as f:
    lines = f.readlines()
    for line in lines:
        textcol = line.split(",")[4]
        toks = line.strip().split()
        tok_counter += len(toks)

print(f"{} tokens found in myfile.txt")
```

```
-UUU:**--F1  preprocess.py    All L2     (Python FlyC:1/0 AC ElDoc) -----------------------
Auto-saving...done
```

Tools for text processing

- Use a database (NoSQL, RDBMS)



```
SELECT SUM(t.word_count_s) AS wordcount FROM
(SELECT LENGTH(TEXT_SPLIT(textcol, ' ')) AS word_count_s
 FROM mytable) t
```

```
-UUU:----F1  preprocess.sql   All L2     (SQL[ANSI] FlyC-) -----------------------------------
Wrote /tmp/preprocess.sql
```

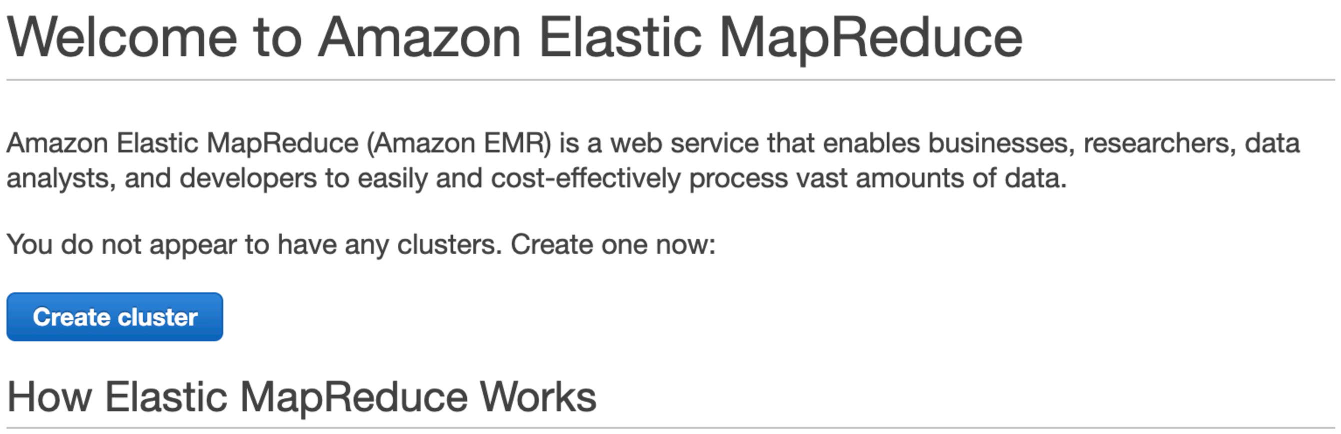
Tools for text processing

- Use a map-reduce framework like Spark

APACHE Spark™

DASK

hadoop



Welcome to Amazon Elastic MapReduce

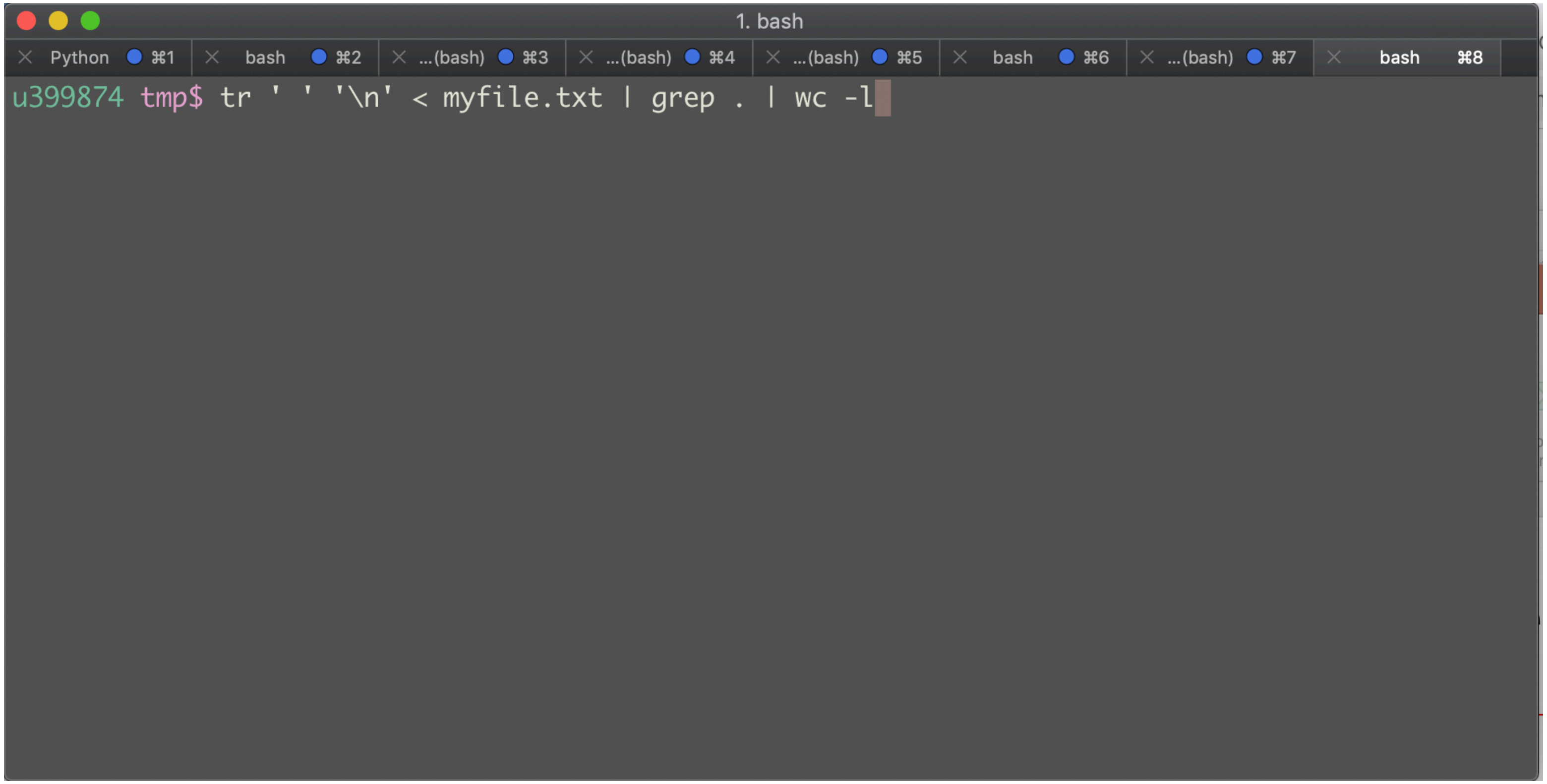
Amazon Elastic MapReduce (Amazon EMR) is a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data.

You do not appear to have any clusters. Create one now:

How Elastic MapReduce Works

Tools for text processing

- Use the command line



```
u399874 tmp$ tr ' ' '\n' < myfile.txt | grep . | wc -l
```

Tools for text processing

Approach	Development speed	Execution speed	Flexibility	Scalability
Ad hoc script	Medium	Slow/Medium	High	Low
NoSQL/SQL	Medium	Medium	Low	High
Map-Reduce	Slow	Fast	High	High++
Command line	Fast	Medium	Medium	Medium

My Interview Question

- You have 10,000 CSV files on your hard drive. Which one has the most rows in it?
  - Answer 1:

```
max_rows, max_f = 0, None
for file in os.readdir(“.”):
    r = pd.read_csv(file).shape[0]
    if r >= max_rows:
        max_rows = r
        max_f = file
```

  - Answer 2:

```
df = sqlContext.read.csv(csv_list)
df.groupby(df.fileId).count()
```

  - Answer 3: Sort by file size and inspect

My Interview Question

- You have 10,000 CSV files on your hard drive. Which one has the most rows in it?
  - Answer 4:

```
wc –l *.txt | sort –n -r
```

**But note that CSV rows can include newlines*

Unix text processing

- History
  - Unix OS developed at Bell Labs in 1970s by Ken Thompson, Dennis Ritchie and others. Included basic text processing functionality
  - Text processing tools extended and improved as part of GNU open source project under Richard Stallman
  - Gnu textutils subsumed under coreutils in 2002
- Resulting tool set is efficient, thoroughly tested and universally available

Unix text processing

- Shells
  - **bash** (Bourne shell), zsh, ksh, csh, tcsh….
  - Interactive or non-interactive
  - Simple interpreted language with minimal syntactic overhead for invoking processes
- Variables
  - Shell variables and environment variables
    - $PATH, $USER, $HOME, $CLASSPATH, …
  - Typically just strings, but some shells have support for other data types (arrays, associative arrays)

Unix text processing: streams

- By convention, processes have access to three data *streams*
  - STDIN (standard input) – An input stream from which the process can read
  - STDOUT (standard output) – An output stream to which the process can write (typically, expected program results)
  - STDERR (standard error) – An output stream to which the process can write (typically, error messages or logs)
- Most Unix programs that accept file arguments can also use `-` to represent STDIN/STDOUT. These are equivalent:

```
sort —o - -
sort
```

Unix text processing: pipes

- Programs that read from `STDIN` and write to `STDOUT` can be chained together using the Unix "pipe" operator
- For example, it is often useful to chain together the `sort` program (which sorts lines) and the `uniq` program (which eliminates successive duplicate lines):

```
sort wordlist.txt | uniq > vocab.txt
```

- Each program in the pipe runs in a separate process, and its output streams to the next program as soon as it is printed to STDOUT
  - This allows for a certain amount of parallelism, but the impact differs by program

UNIX TEXT PROCESSING TOOLS

cat

- Passes input to output unmodified

```
# Equivalent
cat file.txt | sort > sorted.out
sort < file.txt > sorted .out

# Show me the file
cat ~/.profile
```

head/tail

- Prints first/last n lines of a file/stream

```
# Show first 5 lines of a file
head -n 5 testfile.txt

# Show first 5 lines of a file
tail -n 5 testfile.txt

# Continue to print as data is appended to file
# “follow”
tail –f logfile.txt
```

tr

- Makes character-for-character substitutions globally

```
# Replace spaces with newlines
# (Put each word on its own line)
tr ‘ ’ ‘\n’ < sentences.txt > words.txt

# -d for “delete”; -c for “complement”
tr –cd “[:print:]” < messy_text.txt

# Uppercase file
tr ”[:lower:]” “[:upper:]” < lc.txt

# ROT13
tr “[a-z]” “[n-za-m]”
```

fmt/fold

- `fmt` formats text files in a "pretty" way. It limits lines to a specified length and breaks them in a way that respects word boundaries
- `fold` splits lines at a given target length, without consideration of word boundaries

```
# Format text with a line length of 100,
# replacing multiple whitespace characters
# with a single space
fmt –s –w 100 < messy.txt > pretty.txt
```

cut/paste

- `cut` selects specific columns/fields from a character-delimited (e.g., CSV, TSV, pipe-delimited) file
  - It is not smart enough to handle quoted fields
- `paste` is used to stitch columns/fields together using a specified delimiter

```
# Extract columns 2,3,4,5,7 from a CSV
cut –f “2-5,7” –d “,” < wide.csv > narrow.csv

# Add more columns to a CSV file
# Hope the rows align!
paste –d “,” df.csv  new_cols.csv > new_df.csv
```

join

- Performs an “inner join” on two text files with delimited fields.

```
# Add more columns to a CSV file
# Explicitly use first column as join key
join –t “,” df.csv  new_cols.csv > new_df.csv

# Same thing, but LEFT join
# (include all rows from file 1)
join –t “,” –a 1 df.csv  new_cols.csv > new_df.csv
```

sort

- Sorts text lines (numerically or alphabetically)

```
# Sort lines alphabetically (A-Za-z)
sort < vocabulary.txt > dictionary.txt

# Sort lines z-a, ignoring case
sort –r –f < vocabulary.txt > dictionary.txt

# Sort a list of numbers
sort –n digits.txt > digits_ascending.txt

# REALLY sort a list of numbers
sort –g floats.txt > floats_ascending.txt
```

uniq

- Eliminates successive identical lines

```
# Get a list of unique lines
uniq < words.txt > dictionary.txt

# Get unique lines (words), but use case-
# insensitive comparison and keep track
# of counts
uniq —c —i < words.txt > vocabulary.txt
```

grep

- Select lines matching a regular expression pattern from a text

```
# Find occurrences of “needle” in a text
grep needle haystack.txt

# Get 5 lines of context for each match
grep –C 5 needle haystack.txt

# Get a count of lines that do NOT
# consist solely of whitespace
grep –c –v -E "^\s+$" infile.txt
```

sed/awk

- Programmatic pattern matching and stream editing
- Regex substitution is especially useful

```
# Find the number of ’a’s at the beginning
# of each line and double it
sed –E ‘s/^(a*)/\1\1/’ < infile.txt

# Print all lines but 12-18
sed ’12,18d’ < infile.txt

# Print all lines consisting only of a
# sequence of digits
sed –E ‘/^\d+$/p’ < infile.txt
```

Example: script to reorder columns

- Input: CSV with 4 columns
  - No text with embedded newlines
- Output: same file with columns 1 and 4 swapped
- Use cut/paste and write tempfiles

```
# Take 3 columnar slices
cut –d ‘,’ –f 1 < 1234.csv > 1.csv
cut –d ‘,’ –f 2,3 < 1234.csv > 23.csv
cut –d ‘,’ –f 4 < 1234.csv > 4.csv

# Paste together in reverse order
paste –d ‘,’ 4.csv 23.csv 1.csv > 4231.csv
```

Example: script to reorder columns

- sed

```
# Use regular expression capturing groups
sed -E 's/^([^,]+)(,.*,)([^,]+)$/\3\2\1/' < 1234.csv
```

More complex scripting

- Conditional execution, loops, filename substitution

```
for file in *.txt
do
    if [[ ${file%.*} == *_lc ]]
        then
          tr ”[:lower:]” “[:upper:]” < $file > ${file%_lc.*}_uc.txt
    fi
done
```

Bonus: `xargs`

- `xargs` is a command that can be used to transform N rows of input (`STDIN`) into N command-line arguments for the next program in the pipeline
- For example, the `find` or `ls` command can be used to generate a list of files, and then those files can be passed as command-line arguments to a tool like `grep`

```
ls –R1 | xargs grep “waldo”
```