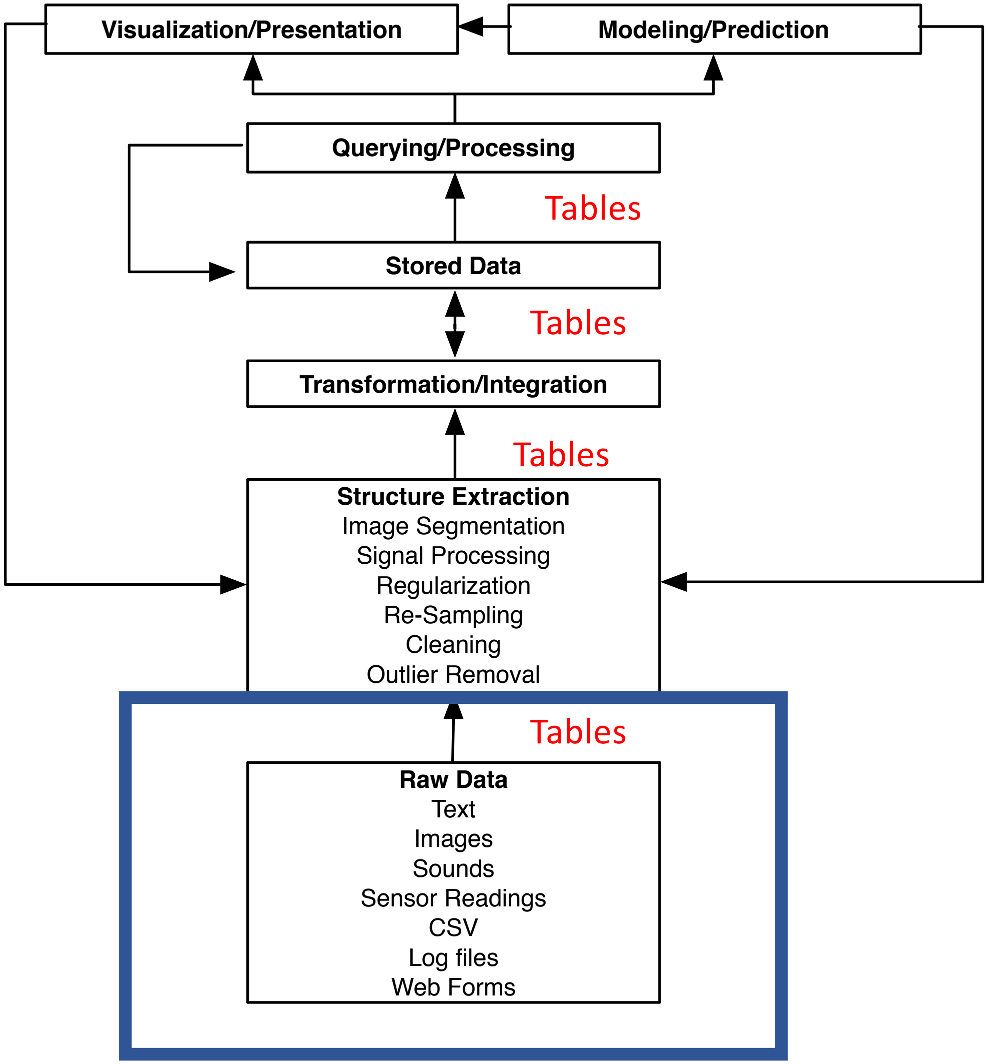
Lec 5: Data Wrangling And Working With Strings

Key ideas: regular expressions, sed/awk/grep, working with text

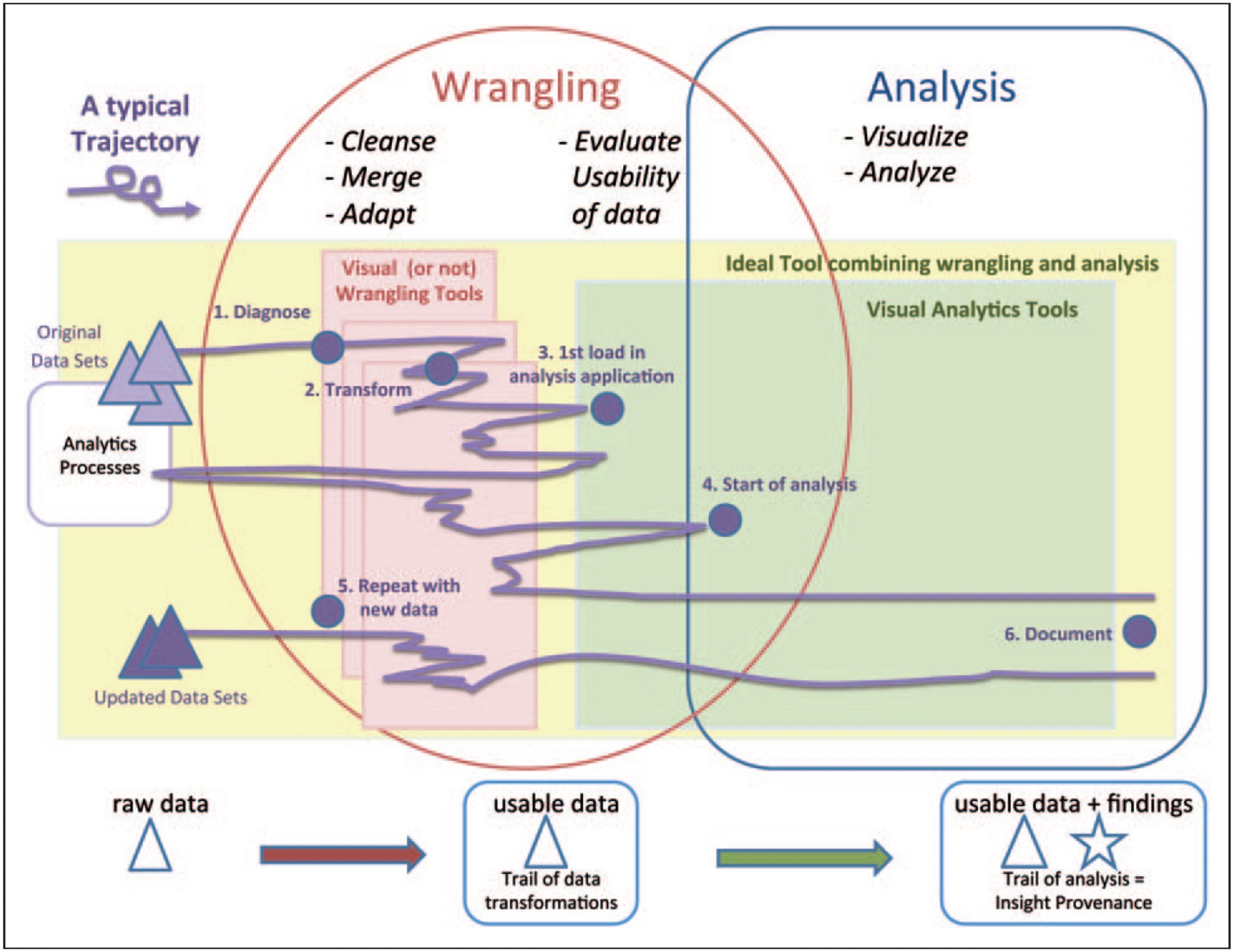
Lab 2 due next Tuesday

Project proposals & teams due next Friday – Use Piazza to find a team

Data Science Pipeline



DATA WRANGLING



[Sean Kandel et al: Research directions in data wrangling: Visualizations and transformations for usable and credible data, Information Visualization, 2011]

THREE POWERFUL TOOLS

1) grep – find text matching a regular expression

Basic syntax:

```
grep 'regexp' filename
```

or equivalently (using UNIX pipelining):

```
cat filename | grep 'regexp’
```

2) sed – stream editor

3) awk – general purpose text processing language

WHAT IS A REGULAR EXPRESSION?

A regular expression (*regex*) describes a set of possible input strings.

Regular expressions descend from a fundamental concept in Computer Science called *finite automata* theory

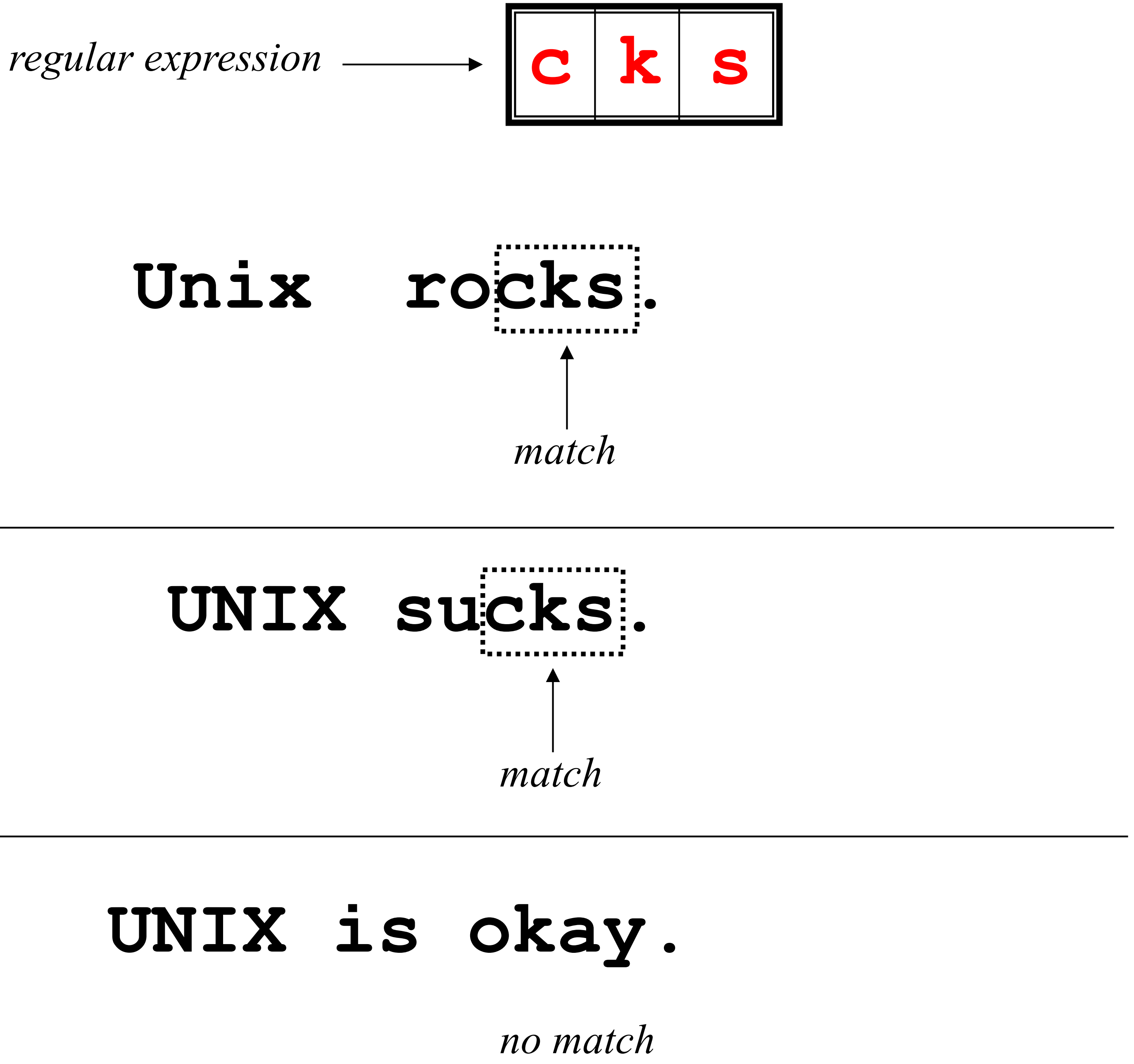
Regular expressions are used in many *nix tools

- vi, ed, sed, and emacs
- awk, tcl, perl and Python
- grep, egrep, fgrep
- compilers

REGULAR EXPRESSIONS

The simplest regular expressions are a string of literal characters to match.

The string *matches* the regular expression if it contains the substring.



regular expression → `c k s`

`Unix rocks.`

match

`UNIX sucks.`

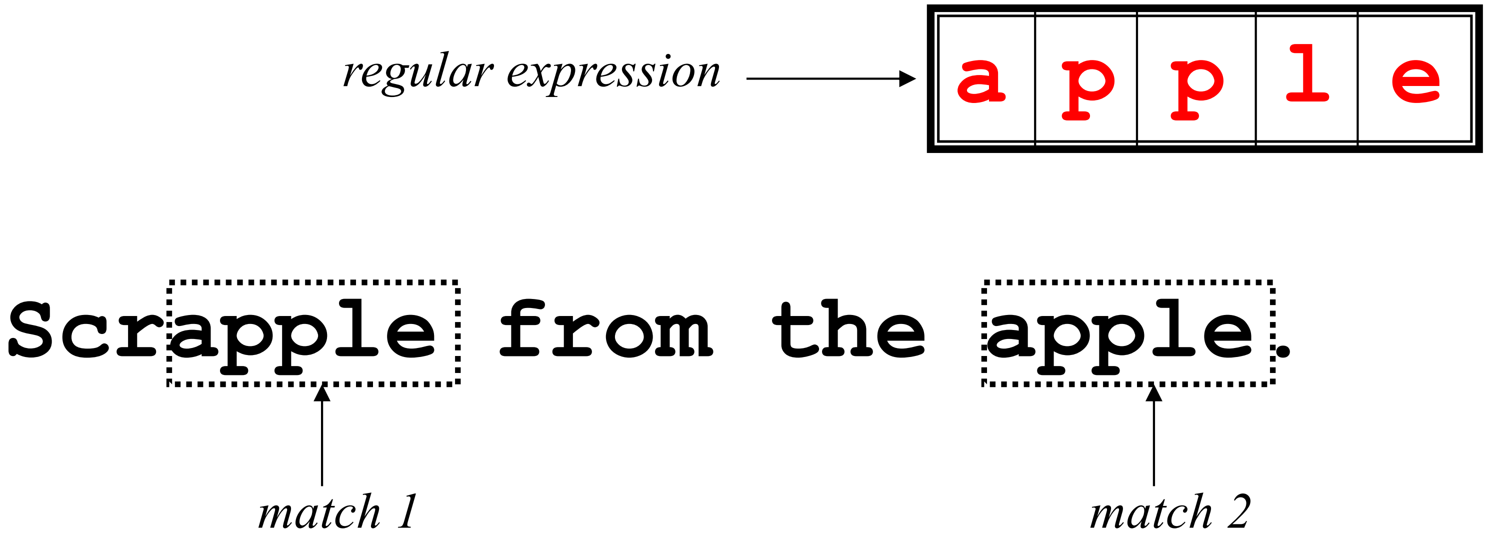
match

`UNIX is okay.`

no match

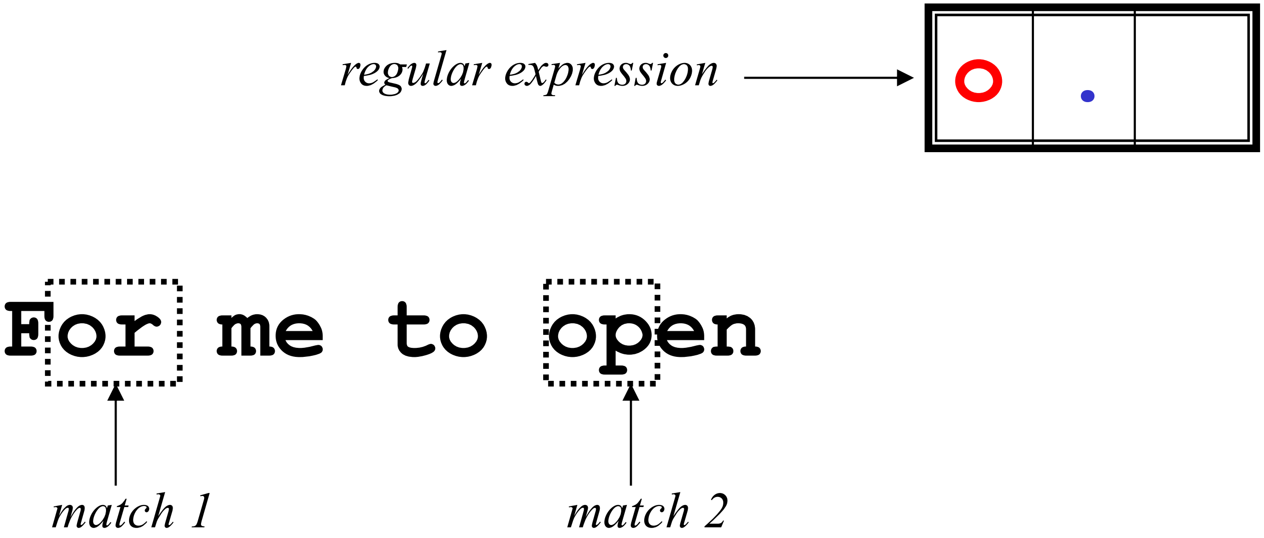
REGULAR EXPRESSIONS

A regular expression can match a string in more than one place.



REGULAR EXPRESSIONS

The `.` regular expression can be used to match any character.



regular expression → `o.`

`For me to open`

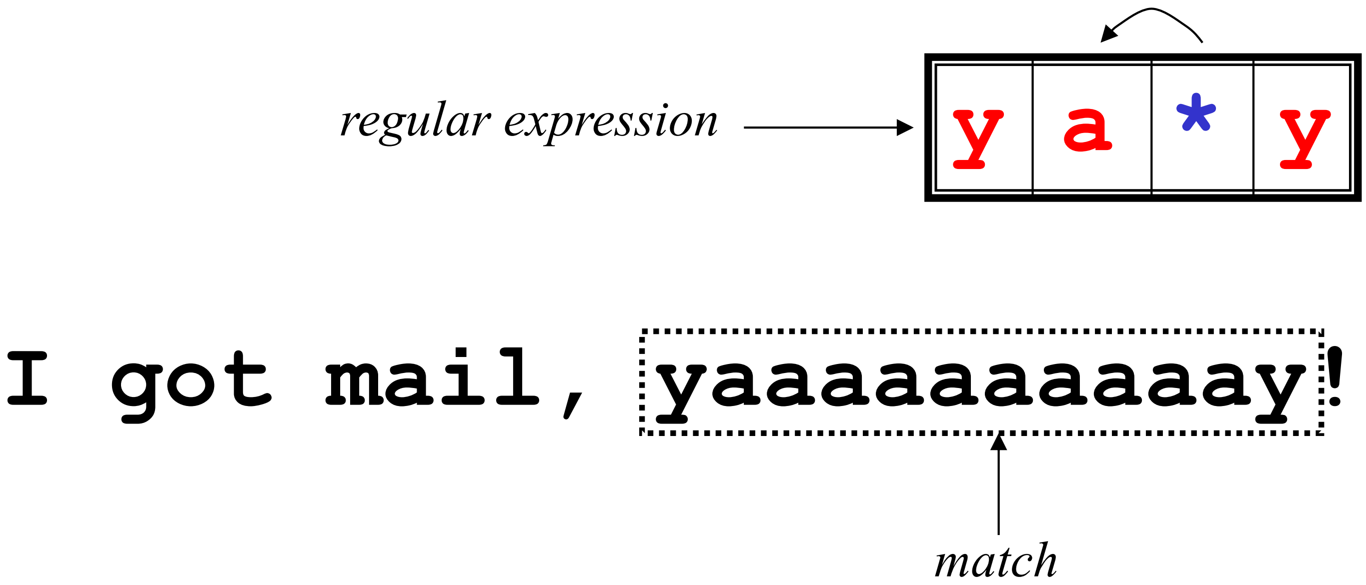
match 1

match 2

REPETITION

The ***** is used to define **zero or more** occurrences of the *single* regular expression preceding it.

\+ Matches one or more occurrences

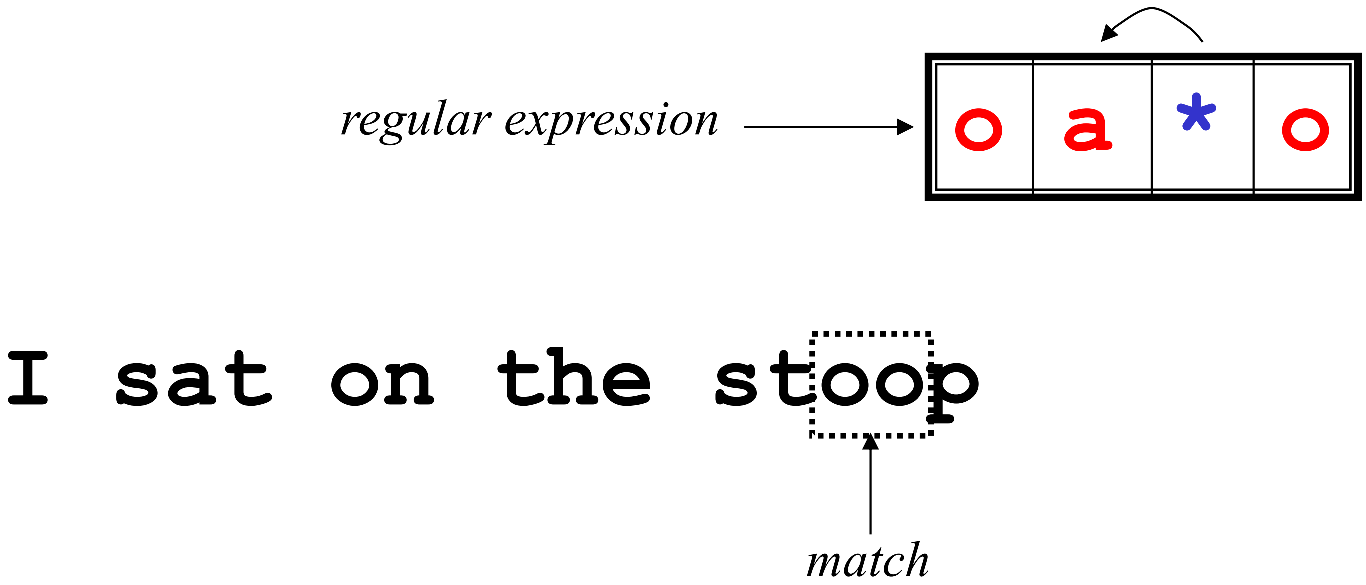


regular expression

`y a * y`

`I got mail, yaaaaaaaaaay!`

match



regular expression

`o a * o`

`I sat on the stoop`

match

REPETITION RANGES

Ranges can also be specified

- `{ }` notation can specify a range of repetitions for the immediately preceding regex
- `{`*n*`}` means exactly *n* occurrences
- `{`*n*`,}` means at least *n* occurrences
- `{`*n*`,`*m*`}` means at least *n* occurrences but no more than *m* occurrences

Example:

- `.{0,}` same as `.*`
- `a{2,}` same as `aaa*`

OR

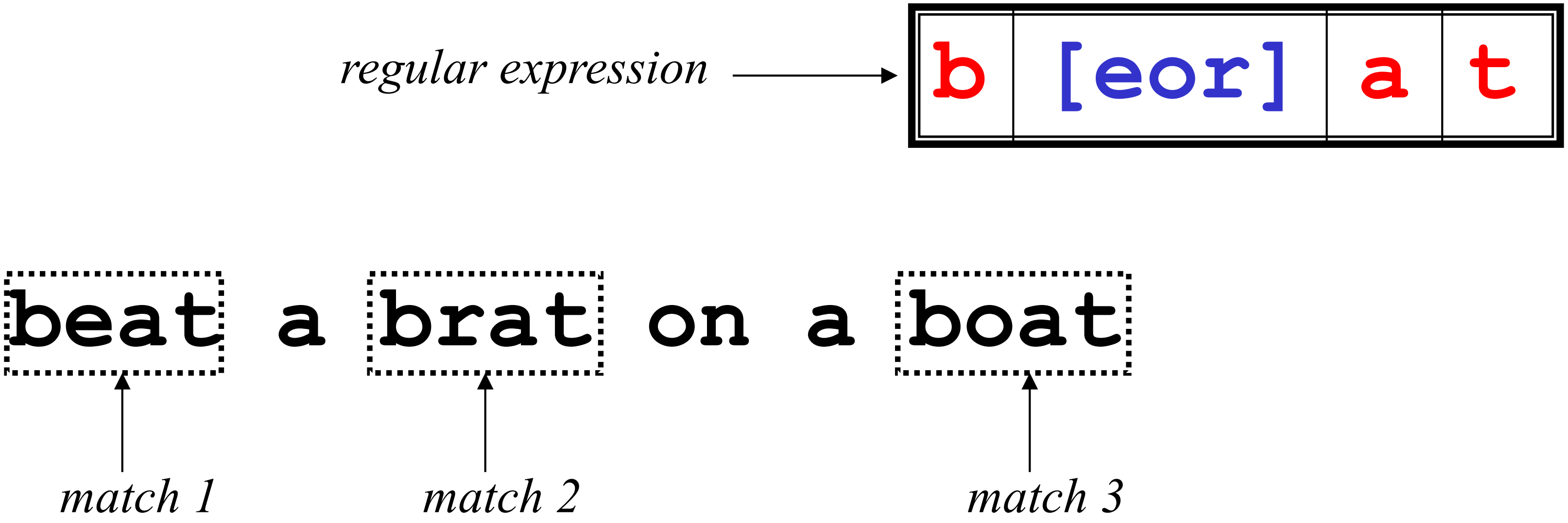
a|b* denotes {ε, "a", "b", "bb", "bbb", ...}

(a|b)* denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}

ab*(c) denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", ...}

CHARACTER CLASSES – OR SHORTHAND

Character classes `[]` can be used to match any specific set of characters.



regular expression →

`b`	`[eor]`	`a`	`t`

`beat a brat on a boat`

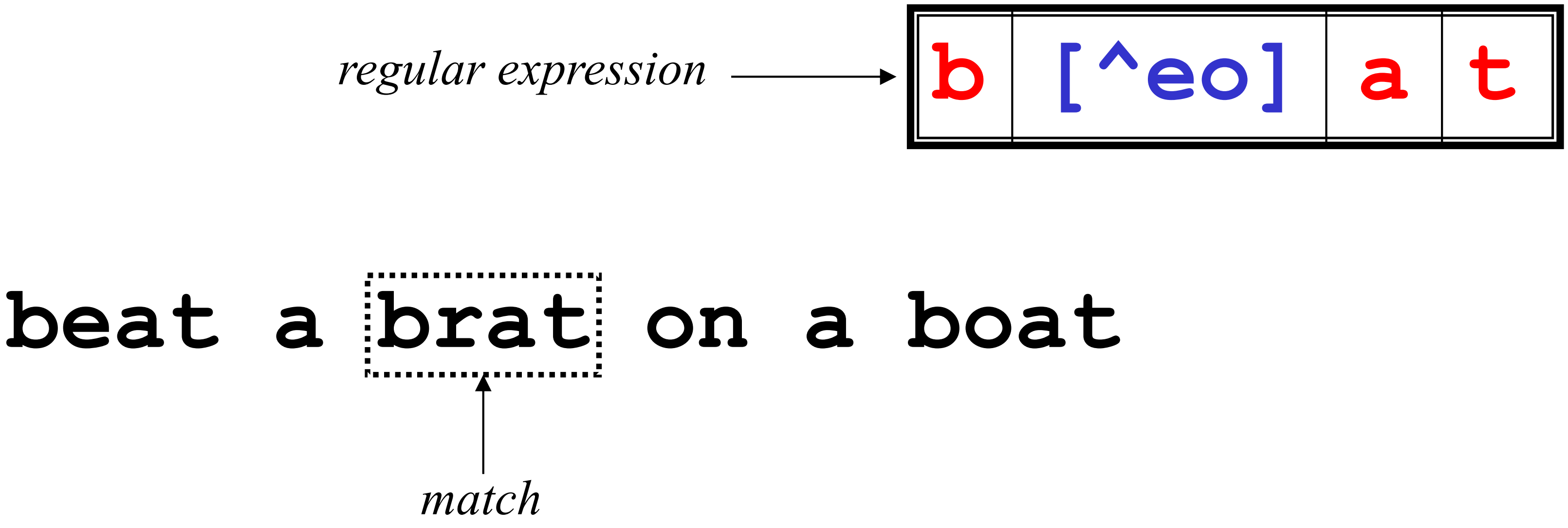
match 1

match 2

match 3

NEGATED CHARACTER CLASSES

Character classes can be negated with the `[^]` syntax.



regular expression → `b [^eo] a t`

`beat a brat on a boat`

match

MORE ABOUT CHARACTER CLASSES

- `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
- `[kK]orn` will match `korn` or `Korn`

Ranges can also be specified in character classes

- `[1-9]` is the same as `[123456789]`
- `[abcde]` is equivalent to `[a-e]`
- You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
- Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range, `[-123]` would match the characters `-`, `1`, `2`, or `3`

NAMED CHARACTER CLASSES

Commonly used character classes can be referred to by name (*alpha*, *lower, upper, alnum*, *digit*, *punct*, *cntrl*)

Syntax [:*name*:]

- [a-zA-Z]   [[:alpha:]]
- [a-zA-Z0-9]   [[:alnum:]]
- [45a-z]   [45[:lower:]]

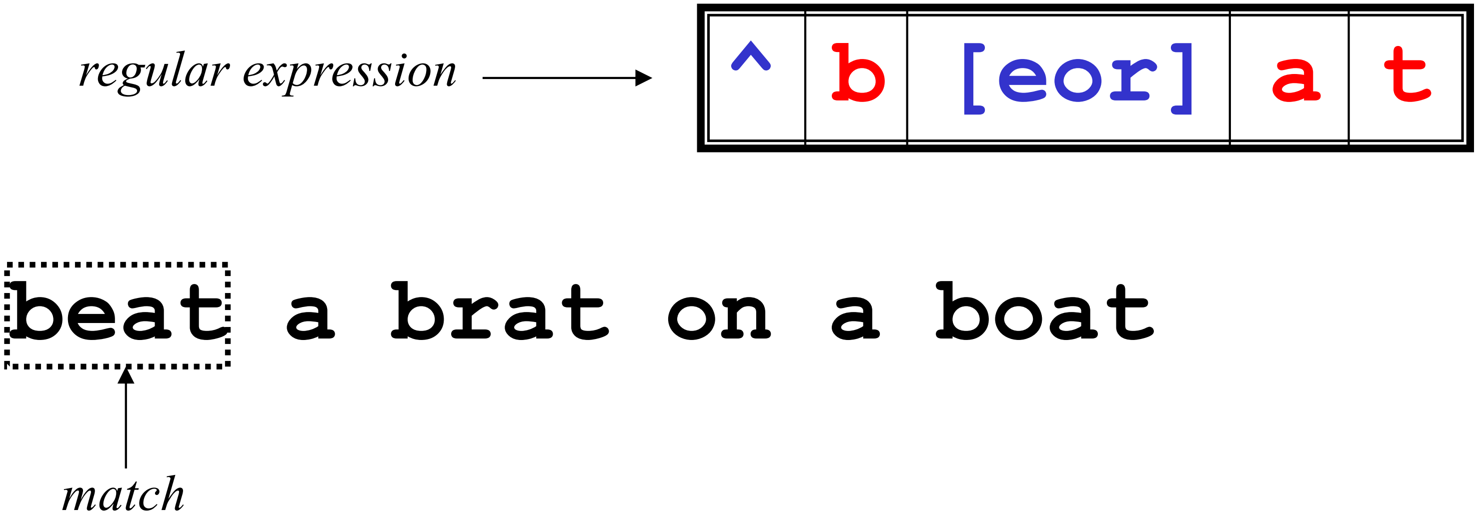
Important for portability across languages

ANCHORS

Anchors are used to match at the beginning or end of a line (or both).

^ means beginning of the line

$ means end of the line

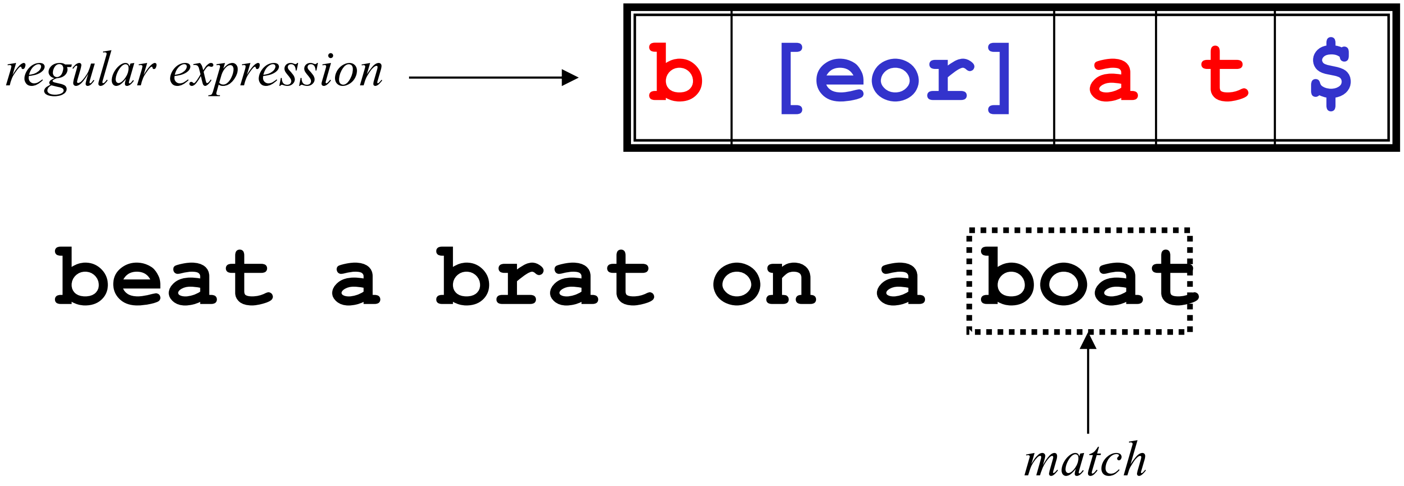


regular expression →

^	b	[eor]	a	t

beat a brat on a boat

match



regular expression →

b	[eor]	a	t	$

beat a brat on a boat

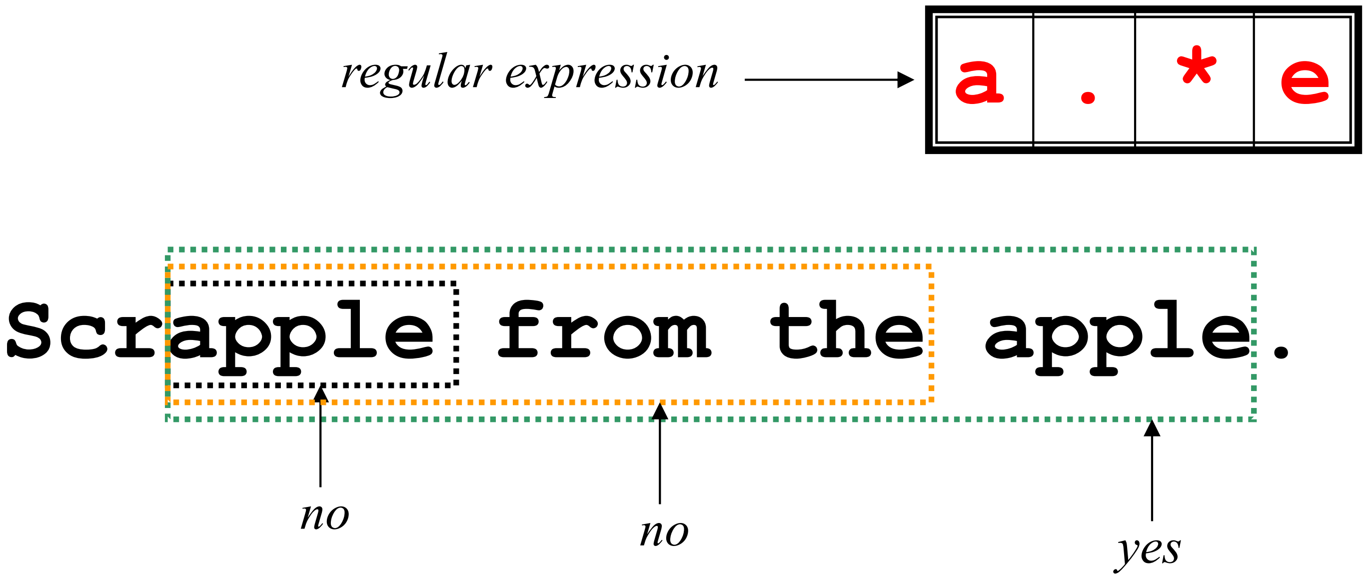
match

---

^word$** **^$

MATCH LENGTH

By default, a match will be the longest string that satisfies the regular expression.



regular expression → `a.*e`

`Scrapple from the apple.`

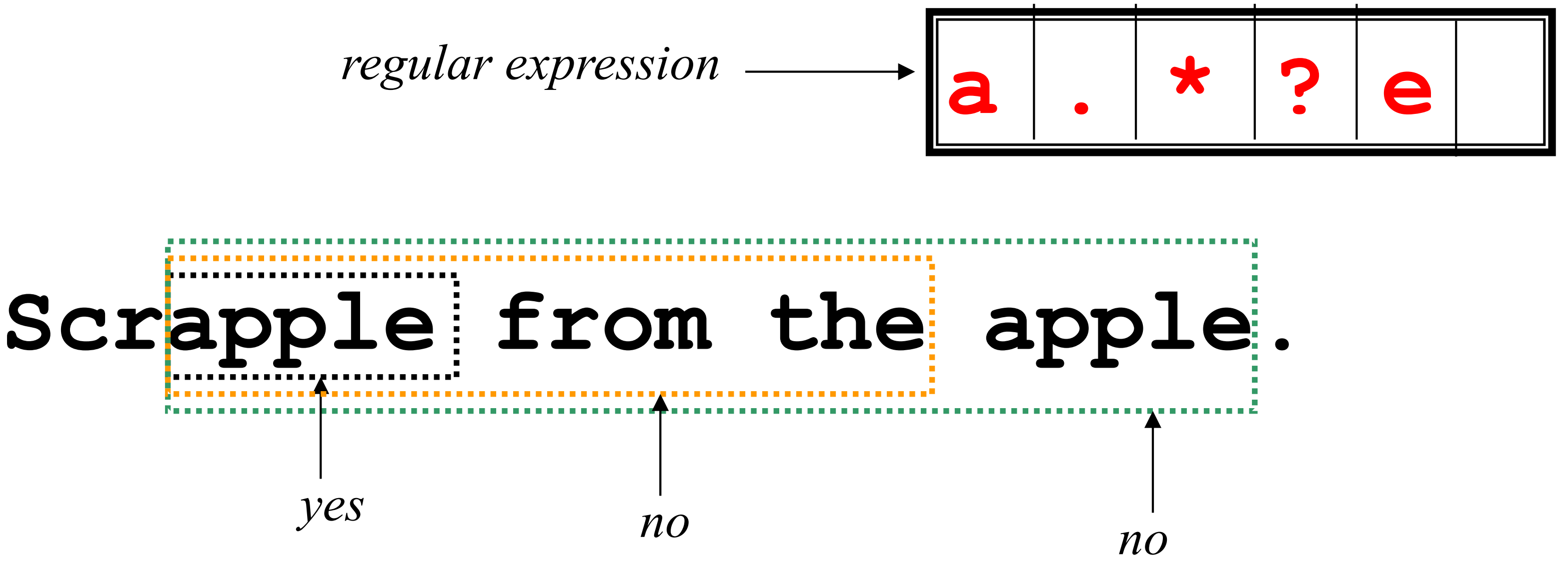
no

no

yes

MATCH LENGTH

Append a ? to match the shortest string possible:



PRACTICAL REGEX EXAMPLES

Dollar amount with optional cents

- `\$[0-9]+(\.[0-9][0-9])?`

Time of day

- `(1[012]|[1-9]):[0-5][0-9] (am|pm)`

HTML headers <h1> <H1> <h2> …

- `<[hH][1-4]>`

GREP

- grep comes from the ed (Unix text editor) search command “global regular expression print” or g/*re*/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, *egrep* and *fgrep* that comprise the *grep* family
- *grep* is the answer to the moments where you know you want the file that contains a specific phrase but you can’t remember its name

GREP DEMO

```
grep '\"text\": ".*location.*"' twitter.json
```

```
 "text": "RT @TwitterMktg: Starting today, businesses can request and share locations when engaging with people in Direct Messages. https://t.co/rpYn…",
      "text": "Starting today, businesses can request and share locations when engaging with people in Direct Messages. https://t.co/rpYndqWfQw",
```

BACKREFERENCES

Used to refer to a match that made earlier in a regex

- *n* is a *backreference* specifier, where *n* is a number

Matches the *n*th subexpression specified by (…)

E.g., to find if the first word of a line is the same as the last:

- **^([[:alpha:]]+) .* \1$**

Here,

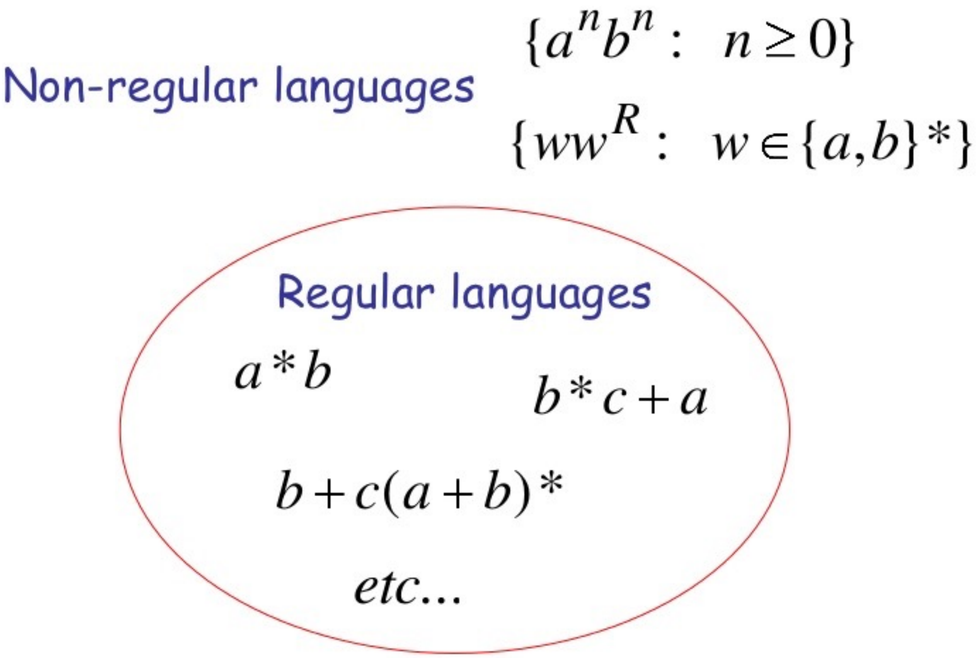
[[:alpha:]]+ matches 1 or more letters

([[:alpha:]]+) is the first subexpression

\1 matches the first subexpression

FORMALLY

Regular expressions are “regular” because they can only express languages accepted by finite automata.   Backreferences allow you to do much more.



Non-regular languages

$\{a^n b^n : \ n \geq 0\}$

$\{ww^R : \ w \in \{a,b\}^*\}$

Regular languages

a^*b

b^*c+a

$b+c(a+b)^*$

etc...

See: https://link.springer.com/article/10.1007%2Fs00224-012-9389-0

BACKREFERENCE TRICKS

Can you find a regex to match L=ww ; w in {a,b}*

e.g., aa, bb, abab, or abbabb

Cannot be expressed with a FA, because need to revisit the tokens in w exactly once, and w is an unknown length.

([ab]*)\1

BACKREFERENCE TRICKS

```
def f(n):  //n is number we are testing for primality
    s = "x" * n //string of “x”’s of length n
    return re.match("^x?$|^(xx+?)\\1+$", s)
```

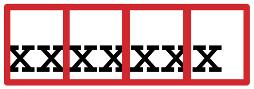
//a single backslash
//in the string

Generates a string of length n, to test if n is prime

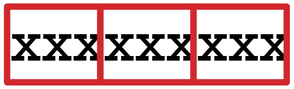
^x?$	*base case*: 0 and 1 are not prime (? matches preceding character 0 or 1 times)
|	*or*
^(xx+?)\1+$	*two or more xs* *repeated one or more times, followed by $*

A prime is a number that cannot be factored. If we find a sequence of N xs that repeats two or more times without any xs left over, we know N is a factor, and the number is not prime.

Example:


xxxxxxx

Doesn’t match, can’t consume all xs with repeated pattern, ==> Prime


xxxxxxxxx

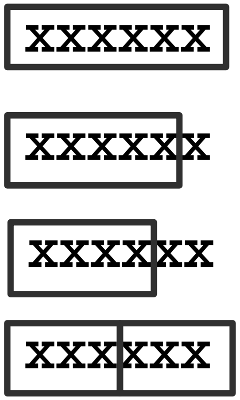
Matches, we consume all xs with 3x repeated pattern, ==> Not Prime

`^x?$|^(xx+?)\1+$`

Generates a string of length n, to test if n is prime

^x?$	*base case*: 0 and 1 are not prime (? matches preceding character 0 or 1 times)
|	*or*
^(xx+?)	*two or more xs* (? makes + match smallest substring)

Without ?:



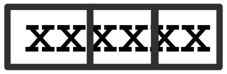
xxxxxx *No match*

xxxxxx *No match*

xxxxxx *No match*

xxxxxx *Match! ➔ Prime*

With ?:

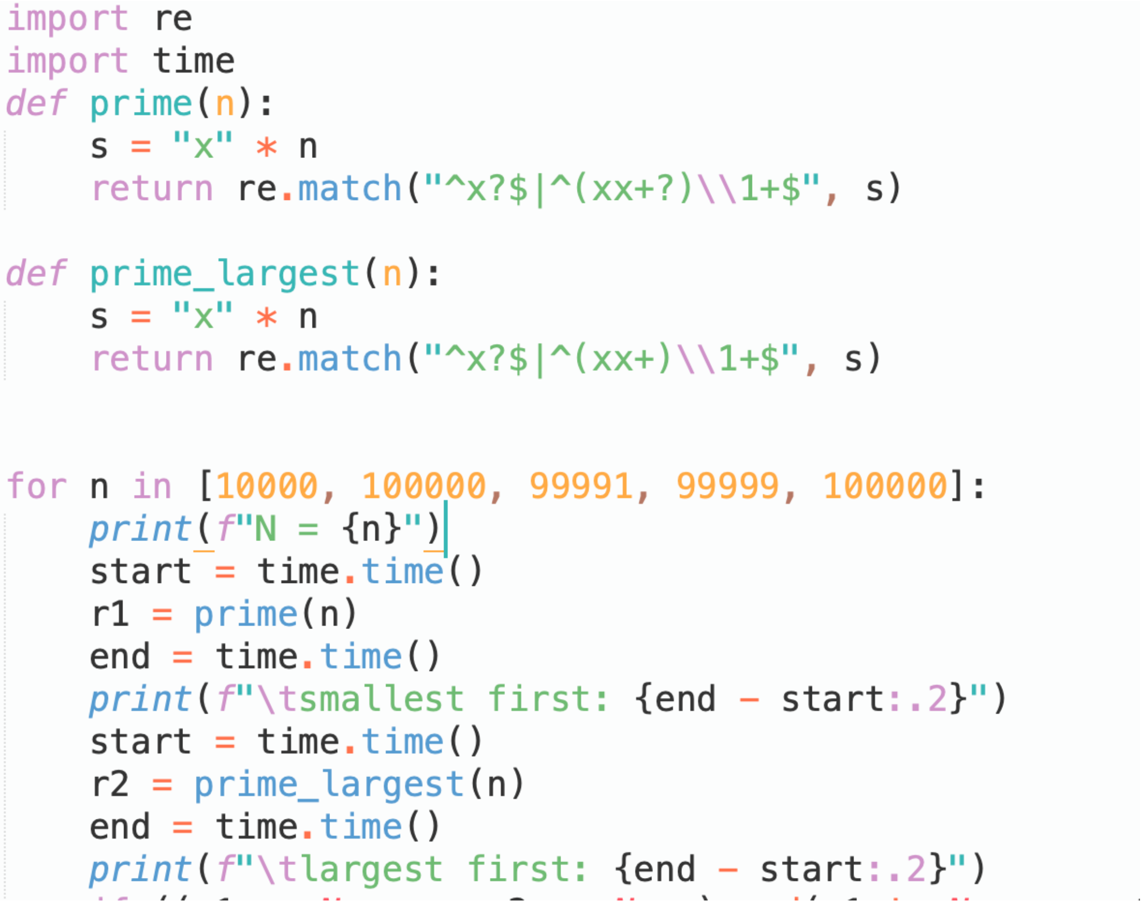


xxxxxx *Match!*

? does not affect correctness; any match indicates non-prime

Search algorithm is to look for (largest | smallest) match; if none found, backtrack and repeated with one (smaller | larger) subsequence

PERFORMANCE EXAMPLE



```python
import re
import time
def prime(n):
    s = "x" * n
    return re.match("^x?$|^(xx+?)\\1+$", s)

def prime_largest(n):
    s = "x" * n
    return re.match("^x?$|^(xx+)\\1+$", s)


for n in [10000, 100000, 99991, 99999, 100000]:
    print(f"N = {n}")
    start = time.time()
    r1 = prime(n)
    end = time.time()
    print(f"\tsmallest first: {end - start:.2}")
    start = time.time()
    r2 = prime_largest(n)
    end = time.time()
    print(f"\tlargest first: {end - start:.2}")
```

```
N = 10000
	smallest first: 0.00021
	largest first: 0.0085
N = 100000
	smallest first: 0.0013
	largest first: 0.79
N = 99991
	smallest first: 3.2
	largest first: 3.2
N = 99999
	smallest first: 0.0026
	largest first: 1.4
N = 100000
	smallest first: 0.0015
	largest first: 0.79
```

https://clicker.mit.edu/6.S079/

CLICKER QUESTION

Select the string for which the regular expression ‘..\.19..’ would find a match:

a) “12.1000”

[✓ b)] “123.1900”

c) “12.2000”

d) the regular expression does not match any of the strings above

CLICKER QUESTION

Choose the pattern that finds all filenames in which

1. the first letters of the filename are chap,
2. followed by two digits,
3. followed by some additional text,
4. and ending with a file extension of .doc

For example : chap23Production.doc

a) chap[0-9]*.doc

b) chap*[0-9]doc

[✓ c)] chap[0-9][0-9].*\.doc

d) chap*doc

THREE POWERFUL TOOLS

1) grep

Basic syntax:

```
grep 'regexp' filename
```

or equivalently (using UNIX pipelining):

```
cat filename | grep 'regexp'
```

2) sed – stream editor

Basic syntax

```
sed 's/regexp/replacement/g' filename
```

For each line in the intput, the portion of the line that matches regexp (if any) is replaced with replacement.

Sed is quite powerful within the limits of operating on single line at a time.

You can use \(\) to refer to parts of the pattern match.

SED EXAMPLE

```
File = Trump is the president.  His job is to tweet.
```

```
sed 's/Trump/Biden/g' file
```

```
sed 's/\(His job is to\).*/\1 run the country./g' file
```

```
Biden is the president. His job is to tweet.
Trump is the president. His job is to run the country.
```

COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```
[
  {
    "created_at": "Thu Apr 06 15:28:43 +0000 2017",
    "id": 850007368138018817,
    "id_str": "850007368138018817",
    "text": "RT @TwitterDev: 1/ Today we’re sharing our vision for the 
future of the Twitter API platform!nhttps://t.co/XweGngmxlP",
    "truncated": false,
…
    "user": {
      "id": 6253282,
      "id_str": "6253282",
      "name": "Twitter API",
      "screen_name": "twitterapi",
```

```
grep \"screen_name\": twitter.json |
sed 's/[ ]*\"screen_name\": \"\(.*\)\",/\1/g'
```

COMBINING TOOLS

Suppose we want to extract all the “screen_name” fields from twitter data

```
[
  {
    "created_at": "Thu Apr 06 15:28:43 +0000 2017",
    "id": 850007368138018817,
    "id_str": "850007368138018817",
    "text": "RT @TwitterDev: 1/ Today we’re sharing our vision for the
future of the Twitter API platform!nhttps://t.co/XweGngmxlP",
    "truncated": false,
…
    "user": {
      "id": 6253282,
      "id_str": "6253282",
      "name": "Twitter API",
      "screen_name": "twitterapi",
```

```
grep \"screen_name\": twitter.json |
sed 's/[ ]*\"screen_name\": \"\(.*\)\",/\1/g'
```

EXAMPLE 2: LOG PARSING

```
192.168.2.20 - - [28/Jul/2006:10:27:10 -0300] "GET /cgi-bin/try/ HTTP/1.0" 200 3395
127.0.0.1 - - [28/Jul/2006:10:22:04 -0300] "GET / HTTP/1.0" 200 2216
```

```
sed -E 's/^([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)[^\"]*\"([^\"]*)\".*/\1,\2/g' apache.txt
```

IP Address **Stuff** **URL**

up to quote

```
192.168.2.20,GET /cgi-bin/try/ HTTP/1.0
127.0.0.1,GET / HTTP/1.0
```

THREE POWERFUL TOOLS

Awk

Finally, awk is a powerful scripting language (not unlike perl). The basic syntax of awk is:

```
    awk -F',' 'BEGIN{commands}
               /regexp1/ {command1} /regexp2/ {command2}
               END{commands}'
```

- For each line, the regular expressions are matched in order, and if there is a match, the corresponding command is executed (multiple commands may be executed for the same line).
- BEGIN and END are both optional.
- The -F',' specifies that the lines should be split into fields using the separator ",", and those fields are available to the regular expressions and the commands as $1, $2, etc.
- See the manual (man awk) or online resources for further details.

AWK COMMANDS

`{ print $1 }` *– Match any line, print the 1st field*

```
$1=="Obama"{print $2}’
```

If the first field is “Obama”, print the 2nd field

```
'$0 ~ /Obama/ {t = gsub("Obama","Trump","g", $0); print t}'
```

If the line contains Obama, globally replace “Trump” for ”Obama” and assign the result to the variable “txt”. Then print it.

Awk commands:

https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html

WRANGLING IN AWK

Input data

```
Reported crime in Alabama,
,
2004,4029.3
2005,3900
2006,3937
2007,3974.9
2008,4081.9
,
Reported crime in Alaska,
,
2004,3370.9
2005,3615
2006,3582
2007,3373.9
2008,2928.3
,
Reported crime in Arizona,
,
2004,5073.3
2005,4827
```

Desired Output:

```
2004,Alabama,4029.3
2005,Alabama,3900
2006,Alabama,3937
2007,Alabama,3974.9
2008,Alabama,4081.9
2004,Alaska,3370.9
2005,Alaska,3615
2006,Alaska,3582
2007,Alaska,3373.9
2008,Alaska,2928.3
2004,Arizona,5073.3
2005,Arizona,4827
2006,Arizona,4741.6
2007,Arizona,4502.6
2008,Arizona,4087.3
2004,Arkansas,4033.1
2005,Arkansas,4068
```

AWK EXAMPLE

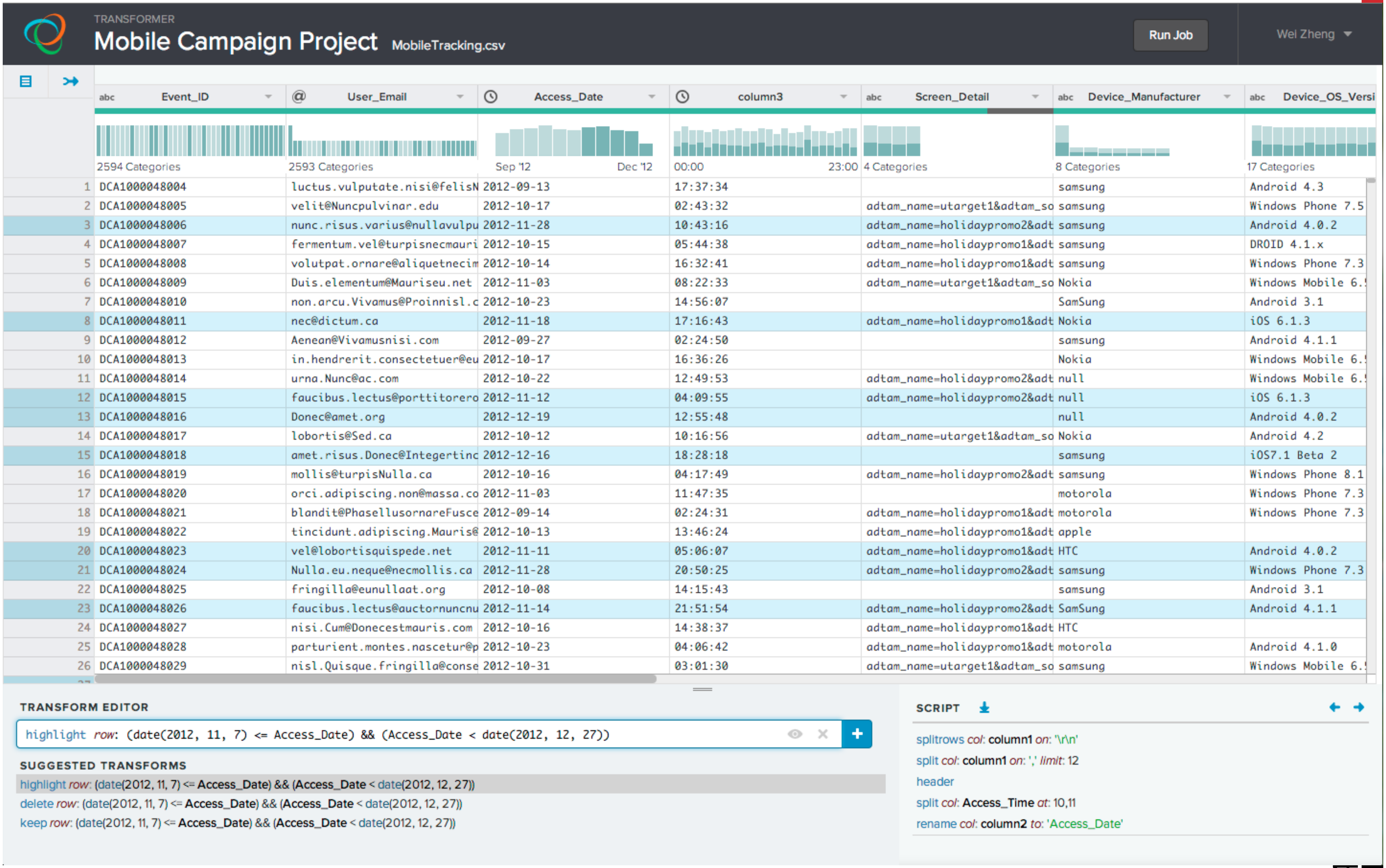
```
Reported crime in Alabama,
,
2004,4029.3
2005,3900
2006,3937
2007,3974.9
2008,4081.9
```

```
BEGIN {FS="[, ]"}
$1=="Reported" {           line begins w/ reported
state = $4" "$5;              some states are two words
gsub(/[ \t]+$/, "", state) strip trailing spaces
}
$1 ~ 20 {print $1","state","$2}
```

line begins with 20 *print year, state, and amount*

DATA WRANGLER / TRIFACTA

http://vis.stanford.edu/wrangler/app/



	Event_ID	User_Email	Access_Date	column3	Screen_Detail	Device_Manufacturer	Device_OS_Versi
1	DCA1000048004	luctus.vulputate.nisi@felisN	2012-09-13	17:37:34		samsung	Android 4.3
2	DCA1000048005	velit@Nuncpulvinar.edu	2012-10-17	02:43:32	adtam_name=utarget1&adtam_so	samsung	Windows Phone 7.5
3	DCA1000048006	nunc.risus.varius@nullavulpu	2012-11-28	10:43:16	adtam_name=holidaypromo2&adt	samsung	Android 4.0.2
4	DCA1000048007	fermentum.vel@turpisnecmauri	2012-10-15	05:44:38	adtam_name=holidaypromo1&adt	samsung	DROID 4.1.x
5	DCA1000048008	volutpat.ornare@aliquetnecim	2012-10-14	16:32:41	adtam_name=holidaypromo1&adt	samsung	Windows Phone 7.3
6	DCA1000048009	Duis.elementum@Mauriseu.net	2012-11-03	08:22:33	adtam_name=utarget1&adtam_so	Nokia	Windows Mobile 6.
7	DCA1000048010	non.arcu.Vivamus@Proinnisl.c	2012-10-23	14:56:07		SamSung	Android 3.1
8	DCA1000048011	nec@dictum.ca	2012-11-18	17:16:43	adtam_name=holidaypromo1&adt	Nokia	iOS 6.1.3
9	DCA1000048012	Aenean@Vivamusnisi.com	2012-09-27	02:24:50		samsung	Android 4.1.1
10	DCA1000048013	in.hendrerit.consectetuer@eu	2012-10-17	16:36:26		Nokia	Windows Mobile 6.
11	DCA1000048014	urna.Nunc@ac.com	2012-10-22	12:49:53	adtam_name=holidaypromo2&adt	null	Windows Mobile 6.
12	DCA1000048015	faucibus.lectus@porttitorero	2012-11-12	04:09:55	adtam_name=holidaypromo2&adt	null	iOS 6.1.3
13	DCA1000048016	Donec@amet.org	2012-12-19	12:55:48		null	Android 4.0.2
14	DCA1000048017	lobortis@Sed.ca	2012-10-12	10:16:56	adtam_name=utarget1&adtam_so	Nokia	Android 4.2
15	DCA1000048018	amet.risus.Donec@Integertinc	2012-12-16	18:28:18		samsung	iOS7.1 Beta 2
16	DCA1000048019	mollis@turpisNulla.ca	2012-10-16	04:17:49	adtam_name=holidaypromo2&adt	samsung	Windows Phone 8.1
17	DCA1000048020	orci.adipiscing.non@massa.co	2012-11-03	11:47:35		motorola	Windows Phone 7.3
18	DCA1000048021	blandit@PhasellusornareFusce	2012-09-14	02:24:31	adtam_name=holidaypromo1&adt	motorola	Windows Phone 7.3
19	DCA1000048022	tincidunt.adipiscing.Mauris@	2012-10-13	13:46:24	adtam_name=holidaypromo1&adt	apple	
20	DCA1000048023	vel@lobortisquispede.net	2012-11-11	05:06:07	adtam_name=holidaypromo1&adt	HTC	Android 4.0.2
21	DCA1000048024	Nulla.eu.neque@necmollis.ca	2012-11-28	20:50:25	adtam_name=holidaypromo2&adt	samsung	Windows Phone 7.3
22	DCA1000048025	fringilla@eunullaat.org	2012-10-08	14:15:43		samsung	Android 3.1
23	DCA1000048026	faucibus.lectus@auctornuncnu	2012-11-14	21:51:54	adtam_name=holidaypromo2&adt	SamSung	Android 4.1.1
24	DCA1000048027	nisi.Cum@Donecestmauris.com	2012-10-16	14:38:37	adtam_name=holidaypromo1&adt	HTC	
25	DCA1000048028	parturient.montes.nascetur@p	2012-10-23	04:06:42	adtam_name=holidaypromo1&adt	motorola	Android 4.1.0
26	DCA1000048029	nisl.Quisque.fringilla@conse	2012-10-31	03:01:30	adtam_name=utarget1&adtam_so	samsung	Windows Mobile 6.

TRANSFORM EDITOR

```
highlight row: (date(2012, 11, 7) <= Access_Date) && (Access_Date < date(2012, 12, 27))
```

SUGGESTED TRANSFORMS

highlight *row*: (date(2012, 11, 7) <= **Access_Date**) && (**Access_Date** < date(2012, 12, 27))

delete *row*: (date(2012, 11, 7) <= **Access_Date**) && (**Access_Date** < date(2012, 12, 27))

keep *row*: (date(2012, 11, 7) <= **Access_Date**) && (**Access_Date** < date(2012, 12, 27))

SCRIPT

splitrows *col*: **column1** *on*: '\r\n'

split *col*: **column1** *on*: ',' *limit*: 12

header

split *col*: **Access_Time** *at*: 10,11

rename *col*: **column2** *to*: 'Access_Date'

BREAK

WORKING WITH TEXT

TEXT AS DATA

What might we want to do?

Find similar documents

    E.g., for document clustering

Find similarity between a document and a string

    E.g., for document search

Answer questions from documents

Assess document sentiment

Extract information from documents

Focus today: Given two pieces of text, how do we measure similarity?

TOKENIZATION

- A token is an instance of a sequence of characters

Input: “*Friends, Romans and Countrymen*”

Output: Tokens

  - *Friends*
  - *Romans*
  - *and*
  - *Countrymen*

- What are valid tokens?
- Typically, just words, but can be complicated

E.g., how many tokens is ***Lebensversicherungsgesellschaftsangestellter,*** *meaning* ‘life insurance company employee’ in German?

WHY TOKENIZE?

- Often useful to think of text as a bag of words, or as a table of words and their frequencies
- Need a standard way to define a word, and correct for differences in formatting, etc.
- LLMs are trained to consume and predict tokens

- Very common in information retrieval (IR) / keyword search
    - Typical goal: find similar documents based on their words or n-grams (length n word groups)

DOCUMENT SIMILARITY EXAMPLE

Suppose we have the following strings, and want to measure their similarity?

```
sen = [
    "Tim loves the band Korn.",
    "Tim adores the rock group Korn.",
    "Tim loves eating corn.",
    "Tim used to love Korn, but now he hates them.",
    "Tim absolutely loves Korn.",
    "Tim completely detests the performers named Korn",
    "Tim has a deep passion for the outfit the goes by the name of Korn",
    "Tim loves listening to the band Korn while eating corn."
]
```

BAG-OF-WORDS MODEL

- Treat documents as sets
- Measure similarity of sets

Standard set similarity metric: Jaccard Similarity

$$sim(s1, s2) = \frac{s1 \cap s2}{s2 \cup s2}$$

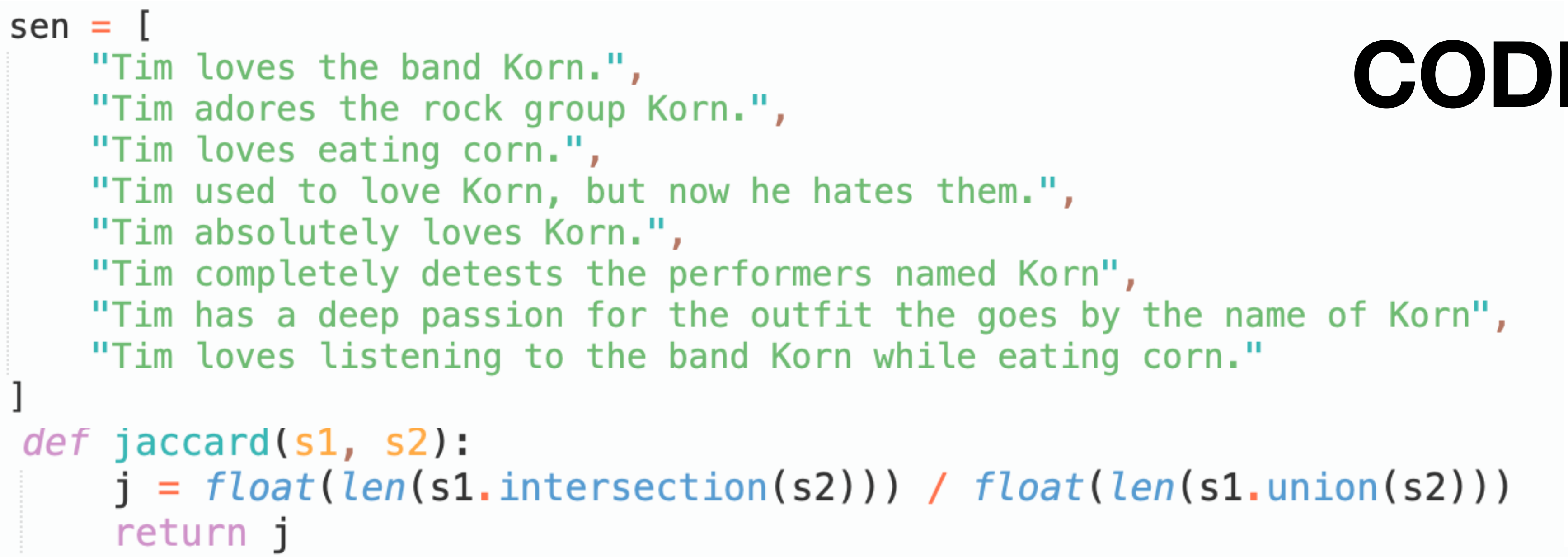
sim({tim,loves,korn}, {tim, loves, eating, corn}) = 2 / 5

sim({tim,absolutely,adores,the,band,korn}, {tim, loves, korn}) = 2 / 7

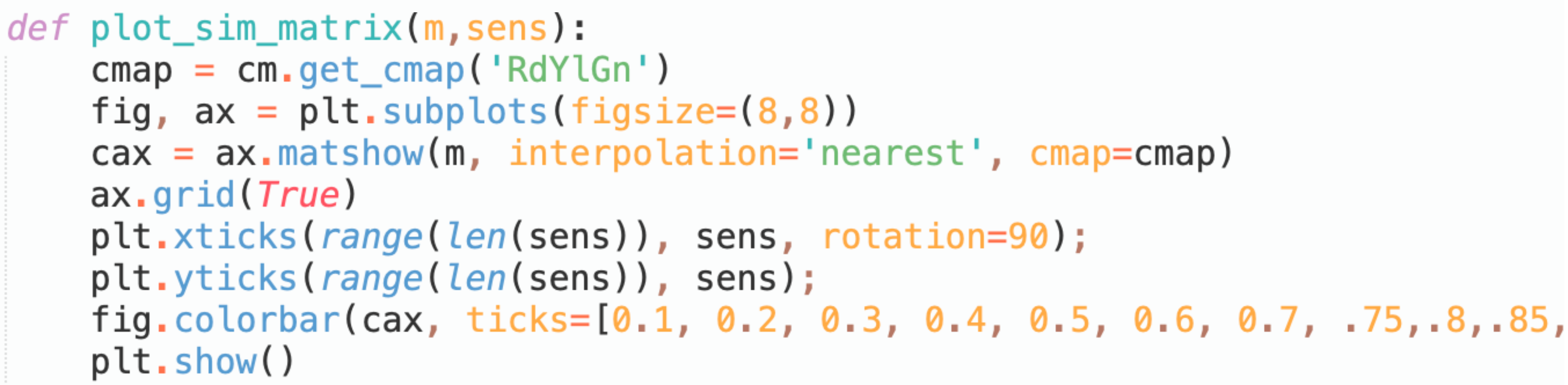
Problems:

All words weighted equally
Same word with different suffix treated differently (e.g., love & loves)
Semantic significance ignored (e.g., adores & loves are the same)
Duplicates are ignored ("Tim really, really loves Korn")

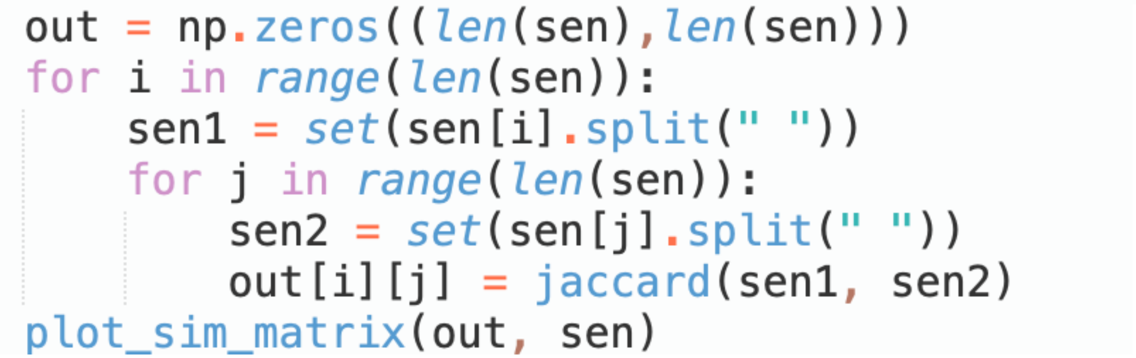
CODE



```
sen = [
    "Tim loves the band Korn.",
    "Tim adores the rock group Korn.",
    "Tim loves eating corn.",
    "Tim used to love Korn, but now he hates them.",
    "Tim absolutely loves Korn.",
    "Tim completely detests the performers named Korn",
    "Tim has a deep passion for the outfit the goes by the name of Korn",
    "Tim loves listening to the band Korn while eating corn."
]
 def jaccard(s1, s2):
     j = float(len(s1.intersection(s2))) / float(len(s1.union(s2)))
     return j
```

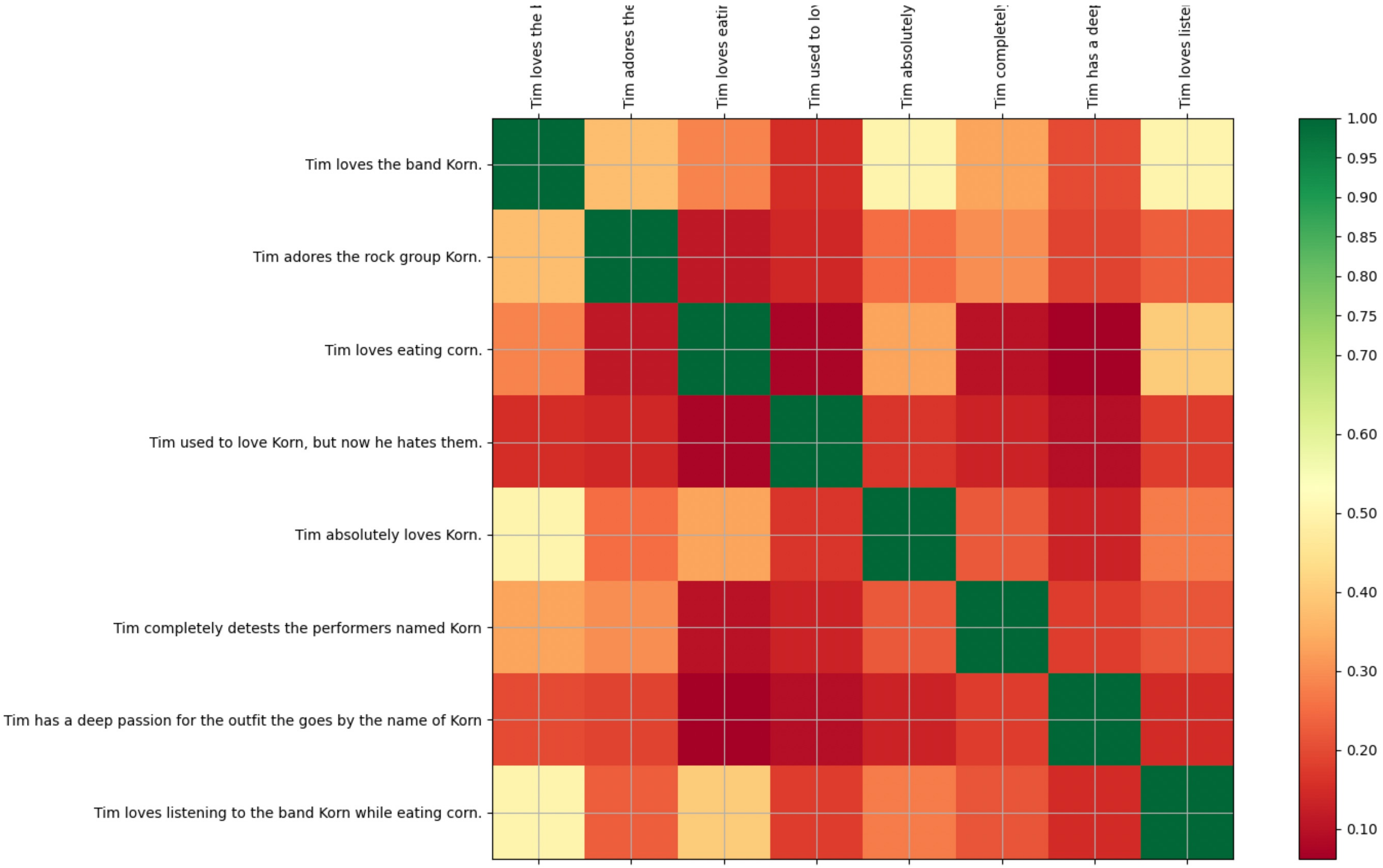


```
def plot_sim_matrix(m,sens):
    cmap = cm.get_cmap('RdYlGn')
    fig, ax = plt.subplots(figsize=(8,8))
    cax = ax.matshow(m, interpolation='nearest', cmap=cmap)
    ax.grid(True)
    plt.xticks(range(len(sens)), sens, rotation=90);
    plt.yticks(range(len(sens)), sens);
    fig.colorbar(cax, ticks=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, .75,.8,.85,
    plt.show()
```



```
 out = np.zeros((len(sen),len(sen)))
 for i in range(len(sen)):
     sen1 = set(sen[i].split(" "))
     for j in range(len(sen)):
         sen2 = set(sen[j].split(" "))
         out[i][j] = jaccard(sen1, sen2)
 plot_sim_matrix(out, sen)
```

EXAMPLE


Tim loves the
Tim adores the
Tim loves eati
Tim used to lo
Tim absolutely
Tim completel
Tim has a dee
Tim loves liste
Tim loves the band Korn.
Tim adores the rock group Korn.
Tim loves eating corn.
Tim used to love Korn, but now he hates them.
Tim absolutely loves Korn.
Tim completely detests the performers named Korn
Tim has a deep passion for the outfit the goes by the name of Korn
Tim loves listening to the band Korn while eating corn.
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.60
0.50
0.40
0.30
0.20
0.10

STOP WORDS

With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:

- They have little semantic content: *the, a, and, to, be*
- There are a lot of them: ~30% of postings for top 30 words

Sometimes you want to include them, as they affect meaning

- Phrase queries: “King of Denmark”
- Various song titles, etc.: “Let it be”, “To be or not to be”
- “Relational” queries: “flights to London”

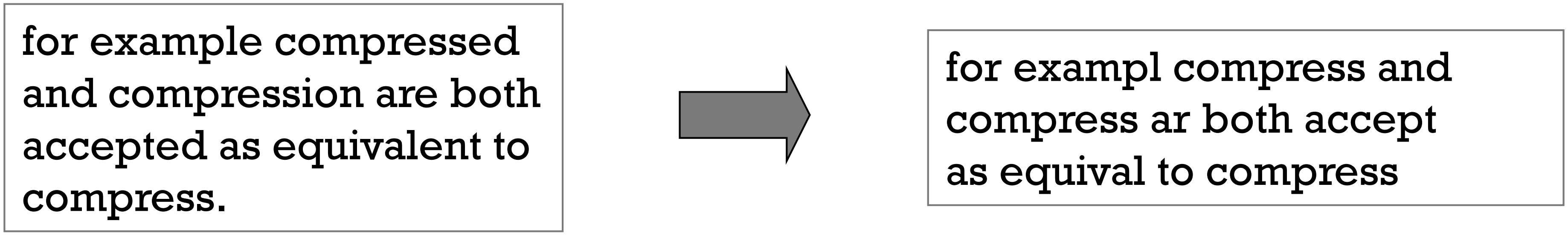
STOP WORDS IN PYTHON

```
from nltk.corpus import stopwords
print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've",
"you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',
'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them',
'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll",
'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has',
'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against',
'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to',
'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then',
'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each',
'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same',
'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should',
"should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',
"couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't",
'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't",
'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't",
'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

STEMMING

- Reduce terms to their “roots” before indexing
- “Stemming” performs crude affix chopping
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.



for example compressed and compression are both accepted as equivalent to compress.

for exampl compress and compress ar both accept as equival to compress

PORTER’S ALGORITHM

Most common algorithm for stemming English

- Other options exist, e.g., snowball

Conventions + 5 phases of reductions

- phases applied sequentially
- each phase consists of a set of commands
- sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

TYPICAL RULES IN PORTER

sses → *ss*

ies → *i*

ational → *ate*

tional → *tion*

Weight of word sensitive rules

(m>1) EMENT →

- *replacement* → *replac*
- *cement* → *cement*

STEMMING IN PYTHON

```python
import nltk.stem.porter

stemmer = nltk.stem.porter.PorterStemmer()
for w in sen[0].split(" "):
    print(stemmer.stem(w))
```

```
tim
love
the
band
korn
```

STEP WORDS + STEMMING

```
sen = [
    "Tim loves the band Korn.",
    "Tim adores the rock group Korn.",
    "Tim loves eating corn.",
    "Tim used to love Korn, but now he hates them.",
    "Tim absolutely loves Korn.",
    "Tim completely detests the performers named Korn",
    "Tim has a deep passion for the outfit the goes by the name of Korn",
    "Tim loves listening to the band Korn while eating corn."
]
```

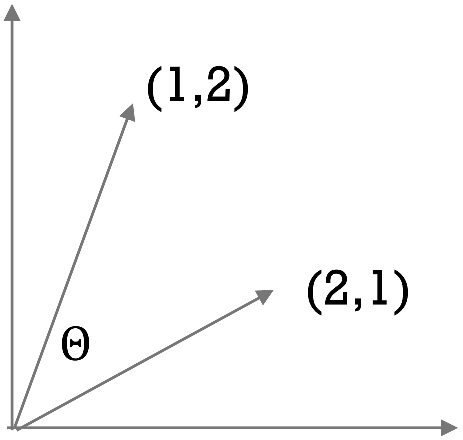
```
tim love band korn
tim ador rock group korn
tim love eat corn
tim use love korn hate
tim absolut love korn
tim complet detest perform name korn
tim deep passion outfit goe korn
tim love listen band korn eat corn
```

COSINE SIMILARITY

Given two vectors, a standard way to measure how similar they are

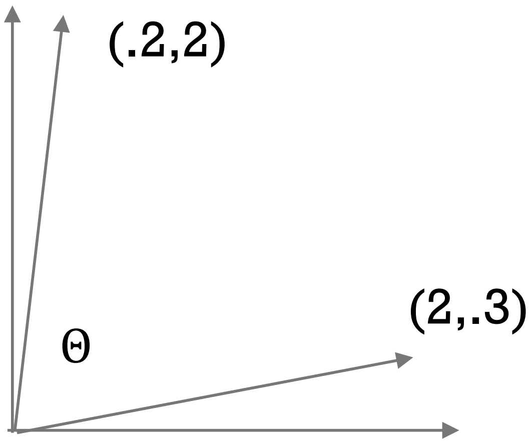
Cos(v1, v2) = closeness of two vectors (smaller is closer)

$\text{Cos}(\Theta) = V1 \bullet V2 / ||V1|| \times ||V2||$



$\text{Cos}(\Theta) = [1\ 2] \bullet [2\ 1] / (\text{sqrt}(5))\ \hat{}\ 2$

$\text{Acos}(4 / 5) = 36.8°$



$||V1|| = 2.01, \ ||V2|| = 2.02$

$\text{Cos}(\Theta) = [.2\ 2] \bullet [2\ .3] / 2.015$

$= 1/2.015$

$\text{Acos}(1/2.015) = 60.2°$

COSINE SIMILARITY OF WORD VECTORS

Cos(Θ) = V1 • V2 / ||V1|| x ||V2||

1 2 3
S1 = Tim loves Korn
4 5
S2 = Tim loves eating corn

V1 = 1 1 1 0 0
V2 = 1 1 0 1 1

V1 • V2 = 2
||V1|| = sqrt(3)
||V2|| = sqrt(4)

2 / sqrt(3) * sqrt(4) = .58

1 2 3
S1 = Tim loves Korn
4 5 6 7
S2 = Tim absolutely adores the band Korn

V1 = 1 1 1 0 0 0 0
V2 = 1 0 1 1 1 1 1

V1 • V2 = 2
||V1|| = sqrt(3)
||V2|| = sqrt(6)

2 / sqrt(3) * sqrt(6) = .47

Typically, when using cosine similarity, we don't take the acos of the values (since acos is expensive)

JACCARD VS COSINE

S1 = Tim loves Korn
S2 = Tim loves eating corn

CosSim(S1,S2) = .29
Jaccard(S1,S2) = .4

S3 = Tim absolutely adores the band Korn
CosSim(S1,S3) = .43
Jaccard(S1,S3) = .28

Jaccard more sensitive to different document lengths than CosSim

CosSim can incorporate repeated words (by using non-binary vectors)

CLICKER https://clicker.csail.mit.edu/6.s079/

Consider two setences:

Sam loves limp bizkit
Sam eats limp biscuits

What is their Jaccard similarity?

- A. 4/6
- B. 2/8
- [✓ C. 2/6]
- D. Something else

$$\frac{\{\text{Sam, limp}\}}{\{\text{Sam, loves, limp, bizkit, eats, biscuits}\}}$$

What is their Cosine similarity?

- A. 1/4
- [✓ B. 2/4]
- C. 4/6
- D. Something else

S1: 1 1 1 1 0 0
S2: 1 0 1 0 1 1
$S1 \cdot S2 = 2$
$||S1|| = ||S2|| = \text{sqrt}(4)$

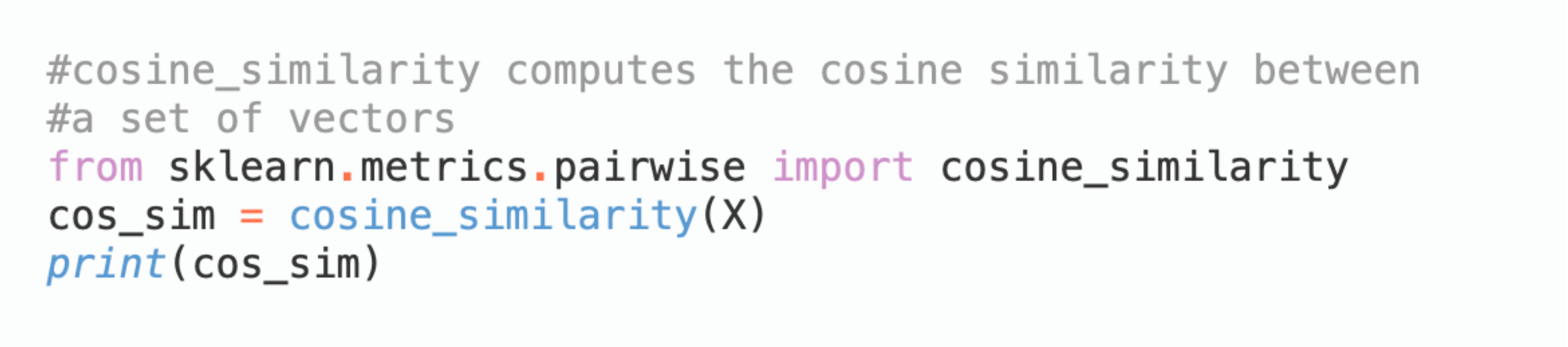
IMPLEMENTING COSINE SIMILARITY

```python
#Count vectorizer translates each document into a vector of counts
f = sklearn.feature_extraction.text.CountVectorizer()
X = f.fit_transform(sen)

print(X.toarray())
print(f.get_feature_names())
```

```
      band                       korn  love               tim
[[0  0  1  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  1  0]
 [0  1  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  1  1  0]
 [0  0  0  0  1  0  0  1  0  0  0  0  0  1  0  0  0  0  0  1  0]
 [0  0  0  0  0  0  0  0  0  0  1  1  0  1  0  0  0  0  0  1  1]
 [1  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  1  0]
 [0  0  0  1  0  0  1  0  0  0  0  1  0  0  1  0  0  1  0  1  0]
 [0  0  0  0  0  1  0  0  1  0  0  1  0  0  0  1  1  0  0  1  0]
 [0  0  1  0  1  0  0  1  0  0  0  1  1  1  0  0  0  0  0  1  0]]

['absolut', 'ador', 'band', 'complet', 'corn', 'deep',
'detest', 'eat', 'goe', 'group', 'hate', 'korn',
'listen', 'love', 'name', 'outfit', 'passion', 'perform',
'rock', 'tim', 'use']
```

IMPLEMENTING COSINE SIMILARITY

```
#Count vectorizer translates each document into a vector of counts
f = sklearn.feature_extraction.text.CountVectorizer()
X = f.fit_transform(sen)

print(X.toarray())
print(f.get_feature_names())
```

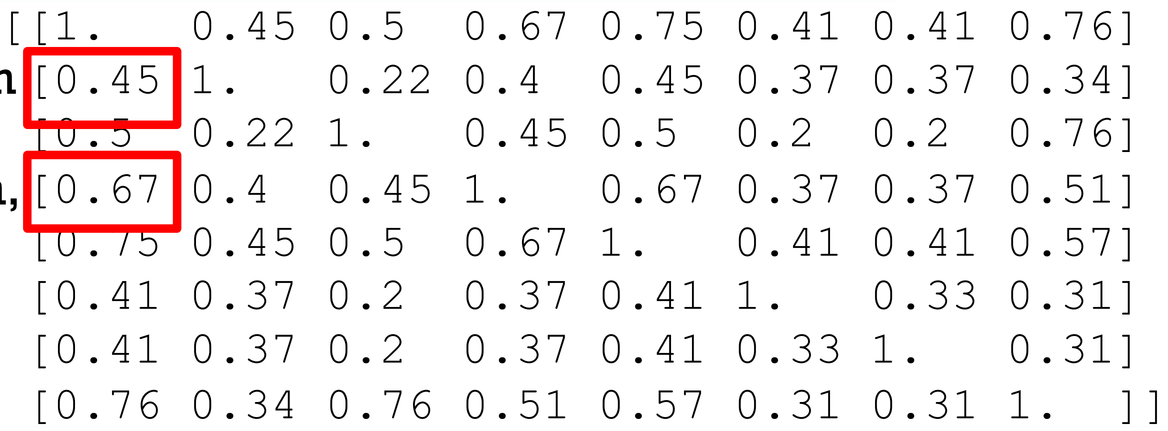


```
#cosine_similarity computes the cosine similarity between
#a set of vectors
from sklearn.metrics.pairwise import cosine_similarity
cos_sim = cosine_similarity(X)
print(cos_sim)
```

Tim loves the band Korn

Tim adores the rock group Korn

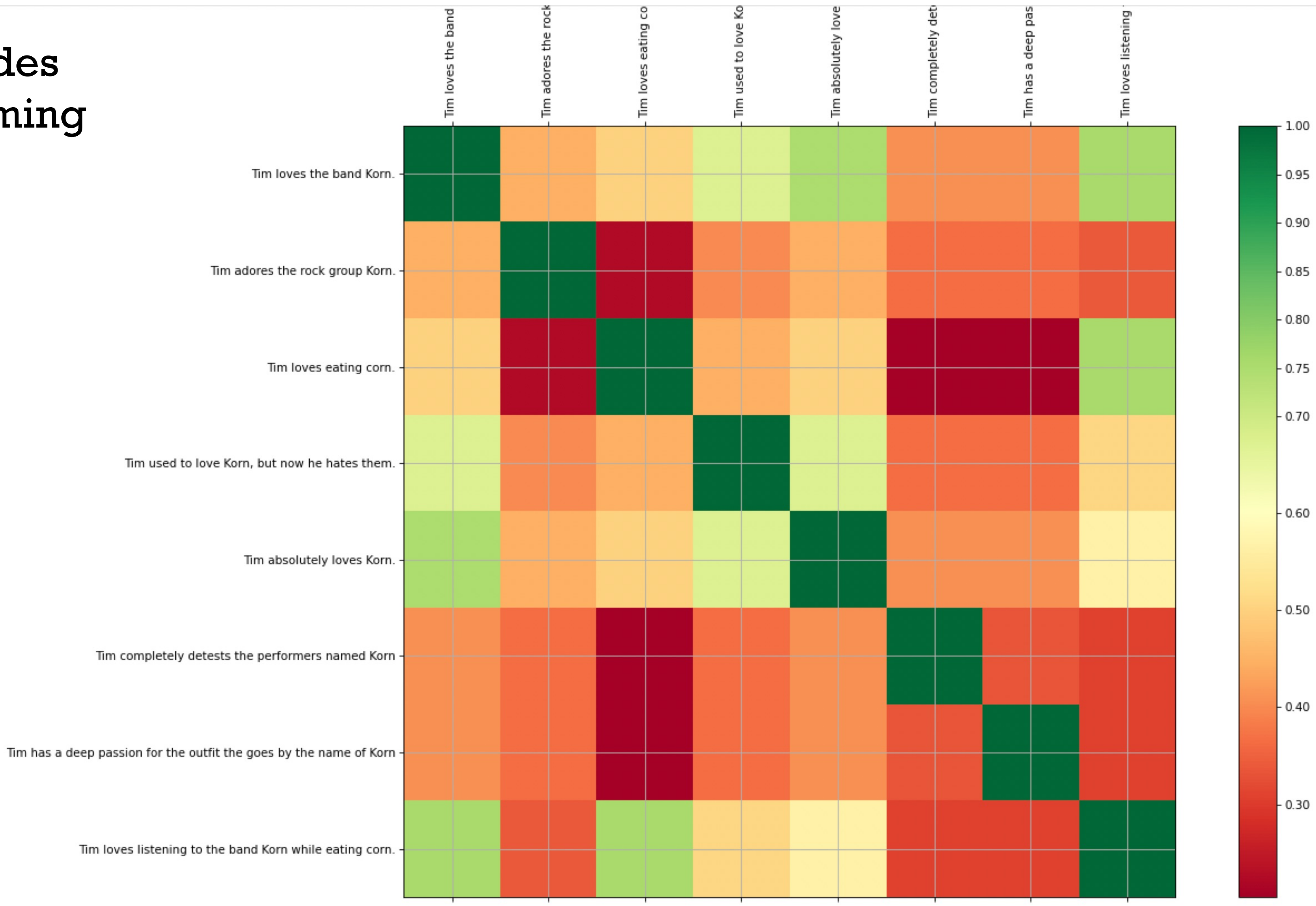
Tim used to love Korn,
but now he hates them



```
[[1.   0.45 0.5  0.67 0.75 0.41 0.41 0.76]
 [0.45 1.   0.22 0.4  0.45 0.37 0.37 0.34]
 [0.5  0.22 1.   0.45 0.5  0.2  0.2  0.76]
 [0.67 0.4  0.45 1.   0.67 0.37 0.37 0.51]
 [0.75 0.45 0.5  0.67 1.   0.41 0.41 0.57]
 [0.41 0.37 0.2  0.37 0.41 1.   0.33 0.31]
 [0.41 0.37 0.2  0.37 0.41 0.33 1.   0.31]
 [0.76 0.34 0.76 0.51 0.57 0.31 0.31 1.  ]]
```

COSINE SIMILARITY PLOT

Includes stemming



WHICH WORDS MATTER: TF-IDF

Problem: neither Jaccard nor Cosine Similarity have a way to understand which words are important

TF-IDF tries to estimate the importance of words based on

1) Their Term Frequency (TF) in a document
2) Their Inter-document Frequency (IDF), across all documents

Assumptions: If a term appears frequently in a document, it's more important in that document

If a term appears frequently in all documents, its less important

TF-IDF EQUATIONS

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Larger the more times document d uses term

t = t

d = document

$f_{t,d}$ = frequency of t in d

For each term t in d, *tf(t,d)* is the fraction of words in d that are t

$$idf(t,D) = log \frac{N}{|\{d \in D: t \in d|}$$

Approaches 0 as more documents use term

N = number of documents

D = set of all documents

$|\{d \in D: t \in d\}|$ = # documents which use term t

For each term t in all D, idf(t,D) is inversely proportional to the number of documents that use t

TF-IDF EQUATIONS

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \qquad idf(t,D) = log \frac{N}{|\{d \in D: t \in d|}$$

$$tf\text{-}idf(t,d,F) = tf(t,d) \bullet idf(t,D)$$

t = t

d = document

$f_{t,d}$ = frequency of t in d

N = number of documents

D = set of all documents

$|\{d \in D: t \in d\}|$= # documents which use term t

TF-IDF EXAMPLE

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

$$idf(t,D) = log \frac{N}{|\{d \in D: t \in d\}|}$$

S1 = Tim loves Korn

S2 = Tim loves eating corn

S1 = [0, 0, .23]
S2 = [0, 0, .17, .17]

Terms = Tim, loves, Korn, eating Korn

tf-idf(Tim,s1) = tf(Tim,s1) x idf(Tim) = 1/3 x log (2/2) = 0

tf-idf(loves,s1) = tf(loves,s1) x idf(loves) = 1/3 x log (2/2) = 0

tf-idf(Korn,s1) = tf(Korn,s1) x idf(Korn) = 1/3 x log (2/1) = 1/3 x .69 = 0.23

tf-idf(eating,s2) = tf(eating,s2) x idf(eating) = 1/4 x log(2/1) = 0.17

tf-idf(corn,s2) = tf(corn,s2) x idf(corn) = 1/4 x log(2/1) = 0.17

Words in all documents aren't helpful if we're trying to rank documents according to their similarity or do keyword search

TF-IDF IN PYTHON

These parameters make it match equations on previous slide

```
#TF-IDF using sklearn
f  = sklearn.feature_extraction.text.TfidfVectorizer(smooth_idf=False,norm='l1')
X = f.fit_transform(sen)
print(X.toarray())
cos_sim = cosine_similarity(X)
print(cos_sim)
```

Tim loves the band Korn
Tim adores the rock group Korn
Tim loves eating corn
Tim used to love Korn,
but now he hates them

```
[[1.   0.13 0.26 0.29 0.37 0.11 0.11 0.57]
 [0.13 1.   0.05 0.09 0.11 0.06 0.06 0.07]
 [0.26 0.05 1.   0.17 0.22 0.04 0.04 0.68]
 [0.29 0.09 0.17 1.   0.25 0.07 0.07 0.16]
 [0.37 0.11 0.22 0.25 1.   0.1  0.1  0.21]
 [0.11 0.06 0.04 0.07 0.1  1.   0.06 0.06]
 [0.11 0.06 0.04 0.07 0.1  0.06 1.   0.06]
 [0.57 0.07 0.68 0.16 0.21 0.06 0.06 1.  ]]
```

TF-IDF not a great choice for these sentences, because it downweights frequent words (Tim, Korn and loves)

MODERN ML TECHNIQUES

Modern deep learning has completely transformed text processing tasks like this

NLP models, e.g., BERT and GPT-3/4 trained to *understand* documents

Models are trained to predict missing words:

Tim loves the ____ Korn

Tim loves eating ____

We're going to try BERT, which is a slightly older model than GPT-3/4

Using billions of documents on the Web (training takes years of GPU time!!!)

Models take a window of text (e.g., 512 words) and produce an output vector (e.g., 768 floats) for each word

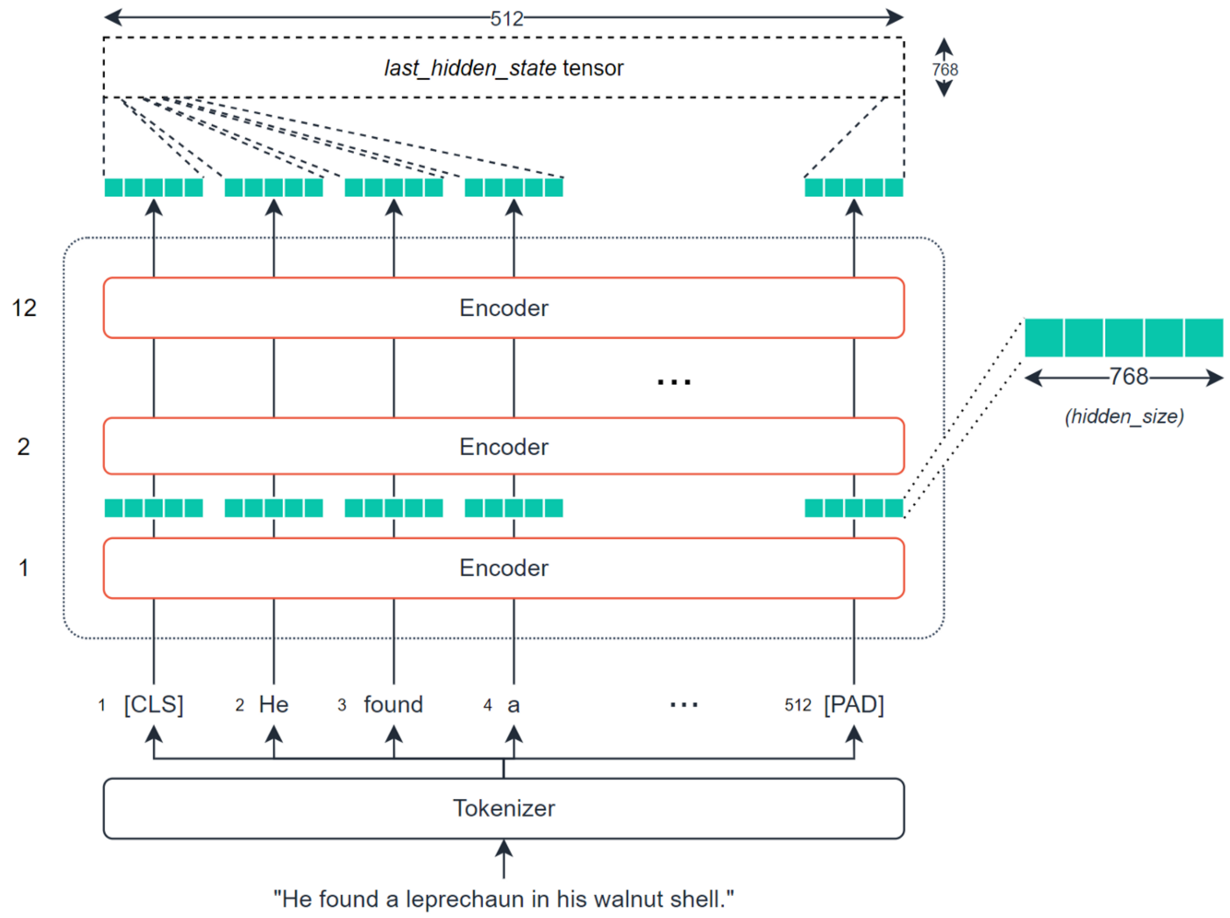
Vector represents the "meaning" of that word in **the context** of the natural language in which it appears

This vector can be used to predict the next word, or to measure the similarity of meaning of two words

BERT

Trained via mask & predict

Transformer Architecture



Each word in input assigned a 768 element output vector, that depends on its context (before and after)

(GPT3 is 175B!)

110M params

https://towardsdatascience.com/bert-for-measuring-text-similarity-eec91c6bf9e1

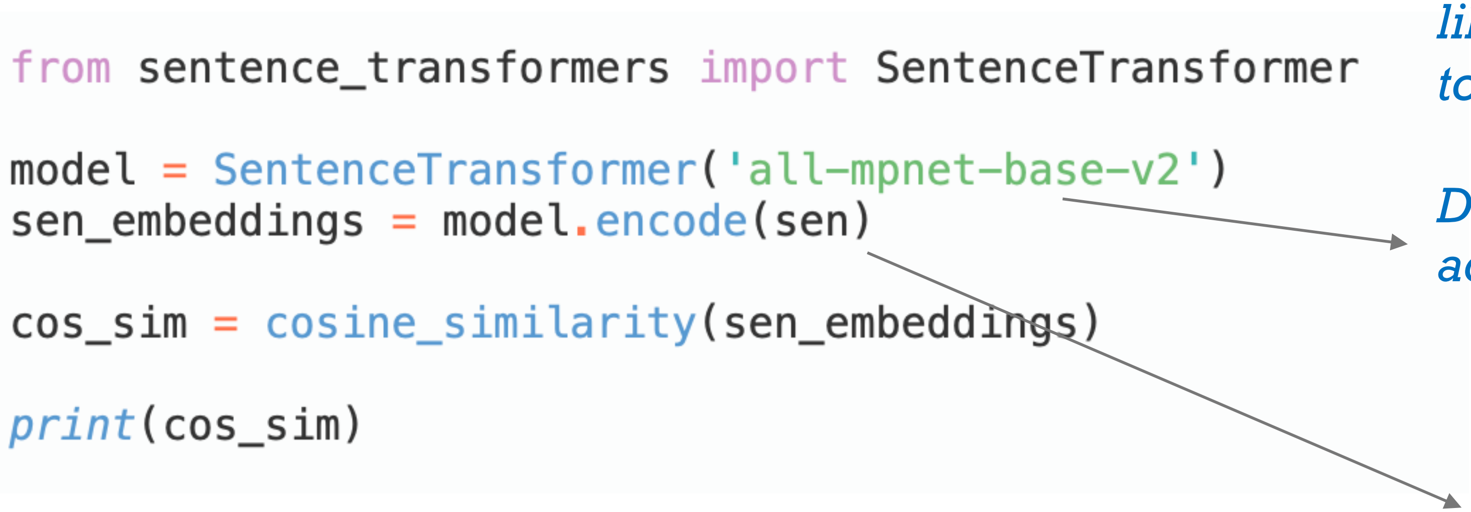
USING BERT VECTORS

Each word is represented by a set of 768-element outputs

Convert to a single element 768-vector for each sentence by averaging words in document

Compute similarity between vectors (e.g., using Cosine Similarity)

Python `sentence-transformers` package makes this trivial



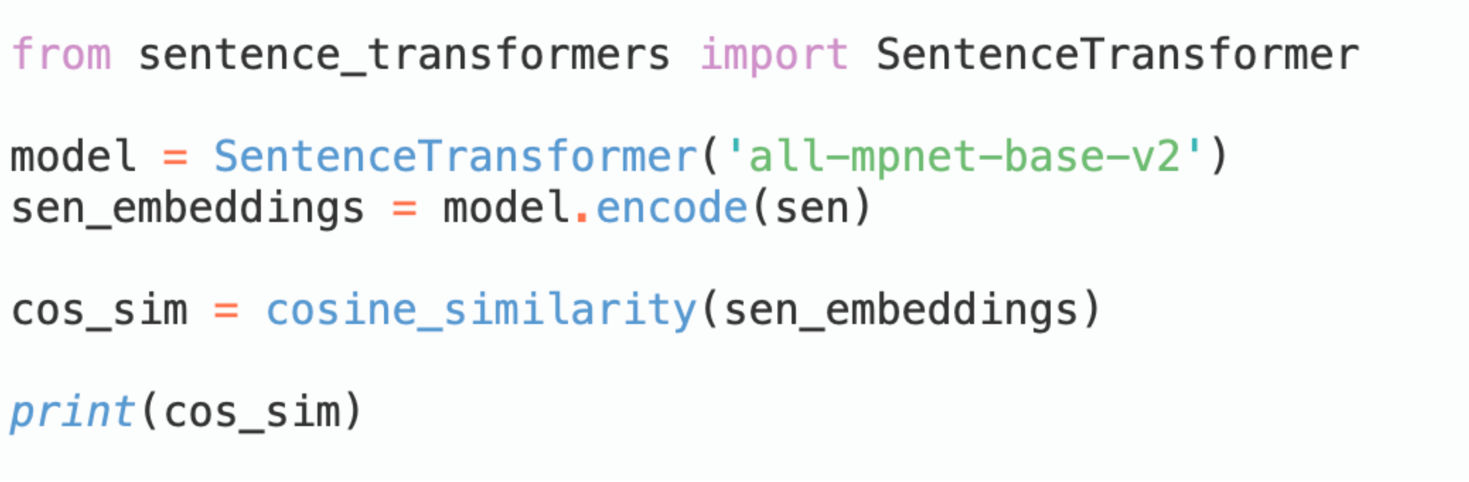
```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-mpnet-base-v2')
sen_embeddings = model.encode(sen)

cos_sim = cosine_similarity(sen_embeddings)

print(cos_sim)
```

A popular BERT-like model known to perform well

Does averaging across documents

Contains a 768-element vector for each document

USING BERT VECTORS



```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-mpnet-base-v2')
sen_embeddings = model.encode(sen)

cos_sim = cosine_similarity(sen_embeddings)

print(cos_sim)
```

Tim loves the band Korn
Tim adores the rock group Korn
Tim loves eating corn
Tim used to love Korn,
but now he hates them

```
[[1.   0.97 0.49 0.83 0.92 0.81 0.93 0.78]
 [0.97 1.   0.46 0.82 0.91 0.81 0.93 0.77]
 [0.49 0.46 1.   0.42 0.52 0.41 0.43 0.81]
 [0.83 0.82 0.42 1.   0.83 0.86 0.8  0.67]
 [0.92 0.91 0.52 0.83 1.   0.79 0.87 0.76]
 [0.81 0.81 0.41 0.86 0.79 1.   0.8  0.66]
 [0.93 0.93 0.43 0.8  0.87 0.8  1.   0.71]
 [0.78 0.77 0.81 0.67 0.76 0.66 0.71 1.  ]]
```

Captures meaning of sentences much better than other metrics

HEAT MAP


Tim loves the band K
Tim adores the rock g
Tim loves eating corn
Tim used to love Korr
Tim absolutely loves
Tim completely detes
Tim has a deep passi
Tim loves listening to
Tim loves the band Korn.
Tim adores the rock group Korn.
Tim loves eating corn.
Tim used to love Korn, but now he hates them.
Tim absolutely loves Korn.
Tim completely detests the performers named Korn
Tim has a deep passion for the outfit the goes by the name of Korn
Tim loves listening to the band Korn while eating corn.
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.60
0.50

Here are a bunch of sentences: "Tim loves the band Korn.", "Tim adores the rock group Korn.", "Tim loves eating corn.", "Tim used to love Korn, but now he hates them.", "Tim absolutely loves Korn.", "Tim completely detests the performers named Korn", "Tim has a deep passion for the outfit the goes by the name of Korn", "Tim loves listening to the band Korn while eating corn."

Which is most similar to the phrase "**Sam loves the band Korn**", and why?

ChatGPT

Based on the text provided in your message, the sentence most similar to "Sam loves the band Korn" is "**Tim loves the band Korn**.”

This is because both sentences express a positive sentiment about the subject's feeling towards the band Korn, with the only difference being the subject of the sentence (Sam vs. Tim). The structure and the main verb "loves" are the same, making them closely similar in both meaning and sentiment.

SUMMARY

Saw three classes of tools – grep, sed, and awk, based on regular expressions to transform data

Saw how tools like Wrangler try to automate this

Looked at text processing techniques

- Jaccard and Cosine similarity
- Tokenization, stemming, stop lists
- TF-IDF
- Embeddings using BERT

We will return to embeddings and GPT models in a few weeks