RICE SOCIAL SCIENCES
Department of Linguistics

regular expressions & tokenization

LING 430 Computational Linguistics Fall 2026
lecture 4a
Sep 15 2026

Siyu Liang
siyu.liang@rice.edu

last time

- functions: `def`, parameters, `return`
- files: read, print, escape characters, write, append

agenda

- regular expressions
- tokenization

today's files

course website → week 4 → activity

- course website
  - `class04a.ipynb`
- Canvas → Files
  - `regex_playground.txt`

regular expressions

regular expression

why

- find every price on a page of listings
  - *$999.99, €450, 300 USD, etc.*
- find all emails on a page (that people sometimes use to spam)
  - everything that is something at something dot something
- find all phone numbers on a document
  - some numbers with optional dashes
- etc.

regular expression

- a **regular expression (regex)** is a notation for describing a set of strings
- a pattern contains
  - ordinary **characters** to search for
  - **operators** that combine them
  - reserved **symbols** such as “any digit” or “start of line”
- used in every programming language, in editors, in search boxes, and inside tokenizers

regex101

- open regex101.com
- paste `regex_playground.txt` into the test string box

regular expression

ordinary characters

- ordinary characters match **exactly** that sequence, **anywhere inside a word**
  - dog
    - matches all instances of ‘dog’

regular expression

character class

- `[]` square brackets mean **one item among the listed**
- `-` dash gives a **range**
- `^` caret first inside the brackets means **not**

type	matches
`[tT]he`	the, The
`b[iau]t`	bit, bat, but, but not bot or beat
`[0-9]`	every single digit
`[^a-z]`	everything that is not a lower-case letter: capitals, spaces, punctuation, digits

regular expression

disjunction

- `|` bar means **or**
- `()` parentheses limit the **reach of the bar**

type	matches
`cat|dog`	every cat and every dog
`pupp(y|ies)`	puppy, puppies
`puppy|ies`	puppy, and the ies of puppies

regular expression

optional

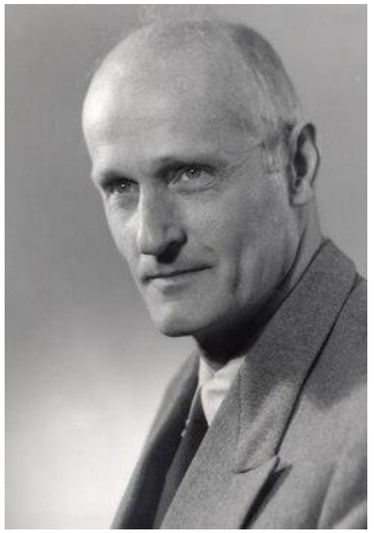
- `?` question mark makes the one thing right before it **optional**

type	matches
`colou?r`	color, colour, and the first five letters of colors and colours
`colou?rs?`	color, colour, colors, colours
`pup(py)?`	pup, puppy, and the pup inside puppies, pupil, puppet

regular expression

Kleene star and plus

- `*` **Kleene star** matches **zero or more** of the thing right before it
- `+` **Kleene plus** matches **one or more** of the thing right before it



Stephen Kleene

type	matches
`pizza-*time`	pizzatime, pizza-time, pizza--time; not pizza time
`ba+`	ba, baa, baaaa, and the ba in bat and bathe
`baa+!`	baa! and baaaa!
`ba*!`	those two, plus ba! and b!

regular expression

counters

- `{n}` means **exactly n** times
- `{m,n}` means **between m and n** times
- `{m,}` means **at least m** times
- `*`, `+`, `?` are basically `{0,}`, `{1,}`, `{0,1}`

type	what lights up
`a{2}`	aa
`a{2,3}`	aa or aaa
`[0-9]{4}`	four digits in a row, e.g. 1999, 2011, 2026

regular expression

the dot

- `.` matches **any one character** except a line break
  - `.*` matches any string, including nothing (empty string ε)
- matches are **greedy**, meaning the **longest match** will be kept

type	matches
`d.g`	dig, dog, dug; not dg, not dodge
`bread.*butter`	the longest line from the first bread to the last butter
`bread.*?butter`	anything that starts with bread and ends with butter

regular expression

parentheses

- `()` parentheses control the **order of operator application** in a nested pattern
  - similar to arithmetic
    - 10 + 2 × 3 = 16
    - (10 + 2) × 3 = 36
  - `ab|cd` is *ab* or *cd*, while `a(b|c)d` is *abd* or *acd*

type	matches
`pup(p(y|ies))?`	pup, puppy, puppies, and the pup of pupil and puppet
`(ba)+`	any repetition of ba

regular expression

anchors

- `^` is **start of line**
  - the caret has another function… what was it?
- `$` is **end of line**
- `\b` is a **word boundary**
  - anchors match positions, not characters, and use up nothing

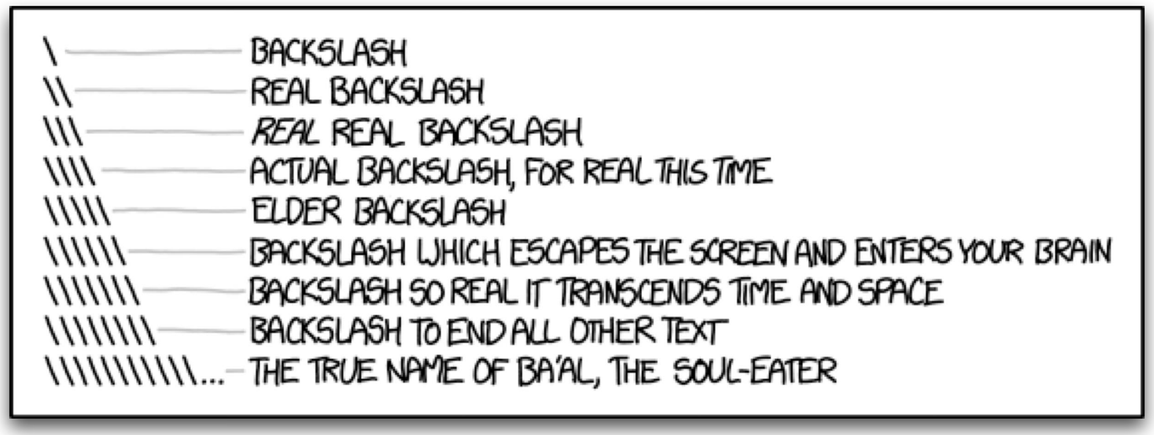
type	matches
un	the ‘un’ in unhappy, undo, tuna, untie, sun
`\bun`	‘un’ in the words that begin with it
`^un`	‘un’ in words beginning with it at beginning of line
`\b99\b`	99 in 99 bottles and $99, but not 299 or 1999

regular expression

escaping

- some characters mean something in regex, so they cannot be typed plainly
  - `. ? + * ( ) { } [ ] | ^ $ \`
- a **backslash** makes them literal
  - `\.` a period, `\$` a dollar sign, `\?` a question mark

type	matches
`\.$`	periods at the end of a line
`\$99`	the price, and only the price
`U\.?S\.?(A\.?)?`	US, U.S., USA, U.S.A.



regular expression

reserved symbols

- `\d` is any digit
- `\w+` is characters used to make up “words”
  - `\w+` is what a regex means by “a word”
- `\s` could be a space, tab, or newline

symbol	stands for
`\d`	`[0-9]`
`\w`	`[a-zA-Z0-9_]`
`\s`	`[ \t\n]`

but…

- every programming language has a **flavor** of regex
  - determines syntax, what counts in a group, etc.
- all characters we've seen are based on the Latin alphabet
  - what if you want to find only Greek letters, Chinese characters, Ge'ez letters, etc.?
- we'll talk about it on Thursday!

activity 1

regex patterns

- find a neighbor and describe each pattern in plain English
- then add some of the potential matches in the text to make sure they light up on regex101.com
  - `[Ii]-?[Pp]hones?`
  - `[A-Z].*(shire|cester|bury)`
  - `[^aeiou0-9]+`
  - `\d+-year-old`
  - `[ts]he\b.*`

activity 2

regex golf

- write one pattern that match all of:
  - *afoot*, *food*, *fool*
- and none of:
  - *foods*, *foolish*, *pool*, *foos*
- any operators you like: `? * + . [ ] ( | ) ^ $`

regex in Python

the re module

- you can use regex in Python as a module
- `import re`

call	returns
`re.search(pattern, s)`	the first match, or None
`re.match(pattern, s)`	the same, but only at the start of s
`re.findall(pattern, s)`	every match, as a list
`re.sub(pattern, replacement, s)`	a new string with every match replaced
`re.compile(pattern)`	the pattern itself, ready to reuse in a loop

regular expression

raw strings

- in an ordinary Python string a backslash starts an escape
  - `\t` becomes a tab before the pattern ever sees it
- patterns are full of backslashes that must reach `re` intact
- an `r` before the quote turns Python's escapes off
- **raw string** pattern: `r"..."`

```
print("a\tb")
→ a   b
print(r"a\tb")
→ a\tb
```

regular expression

re.search

- **pattern** first, **string** second
- returns the **first** match, with its position, or `None` if there is none
- so it works as a condition: `if re.search(pattern, s):`

```
re.search(r"Buttercup", "I'm called little Buttercup")
```

→ <re.Match object; span=(18, 27), match='Buttercup'>

```
re.search(r"Daisy", "I'm called little Buttercup")
```

→ None

regular expression

the match object

- `.group()` returns the **matched text**
- if no match, you'll get an error

```
m = re.search(r"b([aeiou])", "brambular")
m.group(), m.group(1), m.span()
```

→ ('bu', 'u', (4, 6))

```
re.search(r"c", "xyz").group()
```

→ AttributeError: 'NoneType' object has no attribute 'group'

regular expression

re.findall

- `findall` returns as **a list of all matched strings**
- basically what regex101 highlights
- `(?:)` means group, but do not capture in Python to return whole matches, not the pieces

```
re.findall(r"colou?rs?", playground)
```

→ ['color', 'colour', 'colors', 'colours']

```
re.findall(r"pupp(?:y|ies)", playground)
```

→ ['puppy', 'puppies']

regular expression

re.sub

- `re.sub` works like `.replace()`
  - a replacement of `""` deletes the match

```
re.sub(r"colou?r", "color", "colour or color")
```

→ 'color or color'

regular expression

re.sub

- `\1`, `\2`, `\3` in the replacement put the captured pieces back, in any order or any number of times
  - the parentheses does the capturing and is required
  - `\b(\w+) \1\b` finds a repeated word

```
re.sub(r"(b[aeiou])", r"\1\1", "bombastic")
```

→ 'bobombabastic'

```
re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\2-\1-\3", "The date is 10/15/2011")
```

→ 'The date is 15-10-2011'

regular expression

looking back

- last week, we had a cell to clean “love,” “love.” to be just “love” in Shakespeare
- `re.sub(r"^[\W_]+|[\W_]+$", "", "love,")`

  → 'love'

piece	meaning
`^`	at the start
`[\W_]`	a character that is not a word character, or an underscore
`+`	one or more of them
`|`	or
`[\W_]+$`	the same run, at the end
`""`	replaced with nothing

list comprehension

- `[f(x) for x in items]` builds a list in one line
- the same two lines as loops

```
cleaned = [re.sub(r"^[\W_]+|[\W_]+$", "", w) for w in
text.lower().split()]
```

```
cleaned = []
for w in text.lower().split():
    cleaned.append(re.sub(r"^[\W_]+|[\W_]+$", "", w))
```

tokenization

tokenization

- *tokenization*
  - segmenting text into tokens
  - the first step of all language processing task
- why is it hard?
  - punctuation as separate tokens, but kept inside *m.p.h.*, *Ph.D.*, *AT&T*, *cap'n*
  - prices, dates, URLs, hashtags and email addresses kept whole or standardized: *$45.55*, *01/02/06*, *#grwm*
  - numbers: *555,500.50* in the US; *555 500,50* in many European countries
  - *clitics* expanded: *what're* → *what are*; French *j'ai*, *l'homme*
  - multiword expressions could be one token: *New York*, *rock 'n' roll*

tokenization

a first tokenizer

- a regex pattern is already a tokenizer
- both cut *lawyer's* and *wasn't* at the apostrophe, which is the wrong analysis; a real tokenizer is this idea plus a list of exceptions

```
s = "The lawyer's group wasn't amused, was it?"
re.findall(r"\w+", s)
```

→ ['The', 'lawyer', 's', 'group', 'wasn', 't', 'amused', 'was', 'it']

```
re.findall(r"\w+|[^\w\s]", s)
```

→ ['The', 'lawyer', "'", 's', 'group', 'wasn', "'", 't', 'amused', ',', 'was', 'it', '?']

tokenization

a regex tokenizer

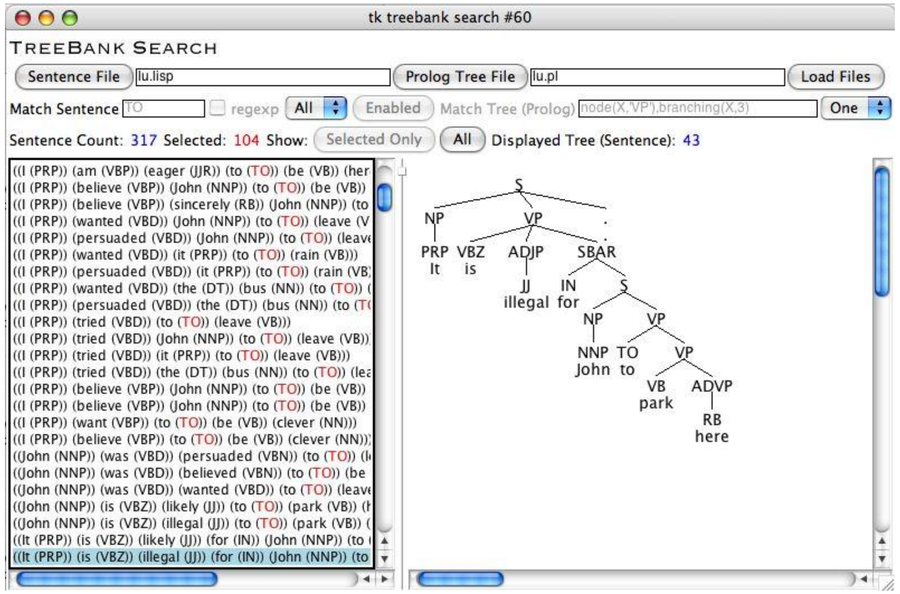
- you can define more complex tokenizer with regex using nltk.regexp_tokenize

```
text = "That U.S.A. poster-print costs $12.40..."
pattern = r"""(?x)
    (?:[A-Z]\.)+     # abbreviations, e.g. U.S.A.
  | \w+(?:-\w+)*     # words with optional internal hyphens
  | ...
"""
nltk.regexp_tokenize(text, pattern)
```

→ ['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']

Treebank

- words have **part of speech (POS)**
  - noun, verb, adjective, etc.
- POS tagging was considered a fundamental problem in natural language processing
  - people have spent a lot of time building the data to train and test computers on it
- **Penn Treebank** is the most famous one
  - available in many languages, supposedly universal
- we'll talk more about them in week 9



tokenization

the Penn Treebank standard

- the tokenization standard behind the Penn Treebank & many other corpora
- clitics split off: *doesn't* becomes *does* and *n't*; if *not* is a word, *n't* is one too
- hyphenated words kept together
- all punctuation separated, including the *$*

```
input:   "The San Francisco-based restaurant," they said,
"doesn't charge $10".
```

```
output:  " The San Francisco-based restaurant , " they said , "
does n't charge $ 10 " .
```

tokenization

word_tokenize

- nltk's tokenizer implements the Treebank standard

```
from nltk import word_tokenize
s = "The lawyer's group wasn't amused, was it?"
s.split()
```

→ ['The', "lawyer's", 'group', "wasn't", 'amused,', 'was', 'it?']

```
word_tokenize(s)
```

→ ['The', 'lawyer', "'s", 'group', 'was', "n't", 'amused', ',', 'was', 'it', '?']

tokenization

the period

- a period ends a sentence, marks an abbreviation, or both at once (*Inc.* at the end of a sentence)
- *Dr.* and *p.m.* were kept whole; *al.* was not: the abbreviation list did not know it

```
word_tokenize("Dr. Niedzielski et al. arrived at 5 p.m. today.")
```

→ ['Dr.', 'Niedzielski', 'et', 'al', '.', 'arrived', 'at', '5', 'p.m.', 'today', '.']

tokenization

tokenizers compared

tokenizer	wasn’t	$2.44	it?
`.split()`	wasn’t	$2.44	it?
`re.findall(r"\w+|[^\w\s]")`	wasn · ’ · t	$ · 2 · . · 44	it · ?
`wordpunct_tokenize`	wasn · ’ · t	$ · 2 · . · 44	it · ?
`word_tokenize`	was · n’t	$ · 2.44	it · ?

activity

break the tokenizer

- work in pairs to write sentences that to test the limit of the tokenizers
- print several tokenizers output side by side
- does output make sense to you?
- suggestions:
  - abbreviations, URLs, email addresses, emoji, hyphens, other scripts
- discussion: how would you do differently?

next time

- Thursday
  - Unicode and subword tokenization
  - Jurafsky & Martin, Ch. 2, §§2.3–2.4
  - optional: Python Unicode HOWTO
- **HW 2 due Thursday, 11:59 pm**