

Lecture 15: RE in Python

Ling 1330/2330 Intro to Computational Linguistics
Na-Rae Han, 10/19/2023

Outline

- ▶ Midterm review, Exercise 7 review
- ▶ *Language and Computers*, Ch.4 Searching
 - ◆ 4.4 Searching semi-structured data with **regular expressions**
 - ◆ 4.41 Syntax of regular expressions
- ▶ Learning regular expressions
 - ◆ regex101 (real-time regex tester):
 - ◆ <https://regex101.com/>
 - ◆ Python Regex syntax reference:
<https://docs.python.org/3/library/re.html>
 - ◆ Regex tutorial:
http://gnosis.cx/publish/programming/regular_expressions.html
 - ◆ Na-Rae's Python 3 Notes on Regex:
<http://www.pitt.edu/~naraehan/python3/re.html>

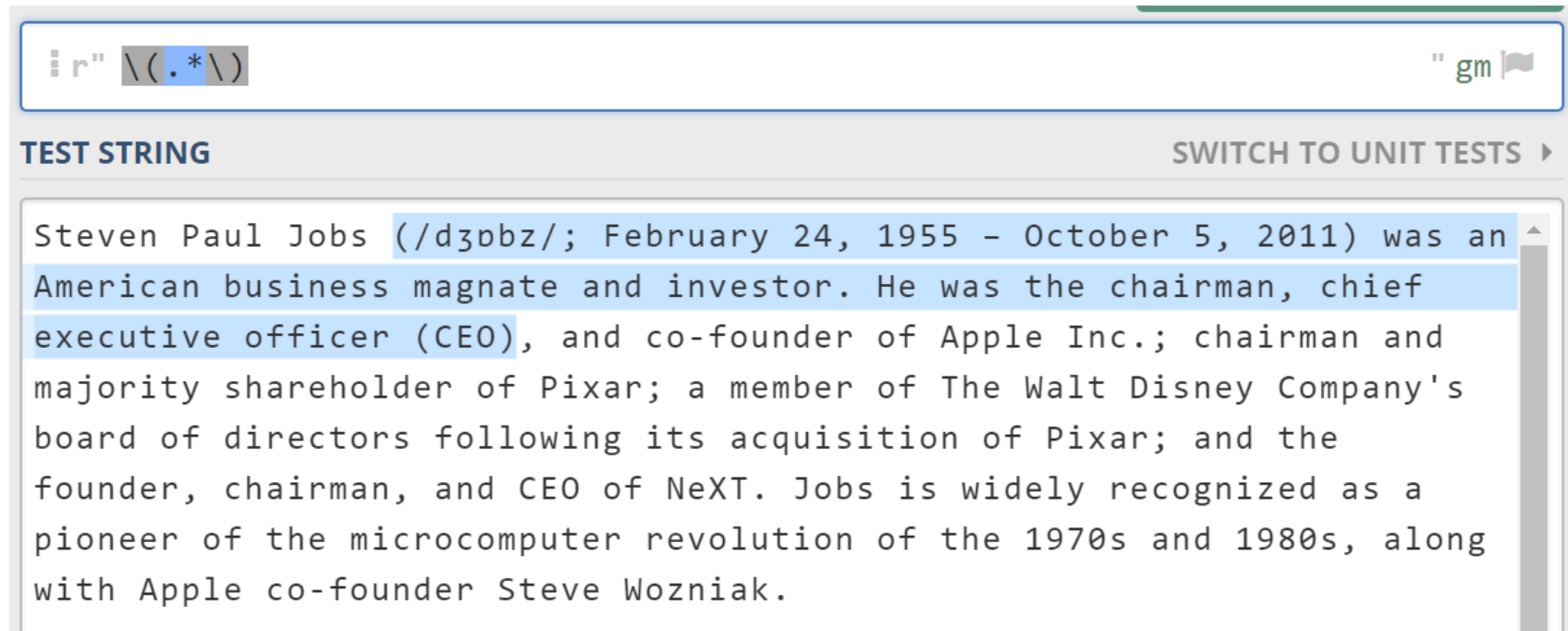
Exercise 7: Regex vs. Jobs

- ▶ `[x|X]` ❌ `[xX]` ✓ `(x|X)` ✓
 - ← Within `[...]`, all characters are already considered forming a set, i.e., alternation. Example: `[aeiou]` (any "vowel" character) Corollary: a `[...]` match is always of width 1 (=single character)!
- ▶ `[A-z]` ❌ `[A-Za-z]` ✓
 - ← `[A-z]` character range goes by ASCII code points: some symbols get included. If you are doing `[A-Za-z]`, see if `\w` will work for your purpose (includes digits and `_`).
- ▶ Hyphenated words
 - ← `\w+-\w+` does not match 'state-of-the-art' as a whole
 - ← `\w+(-\w+)+` does!
- ▶ 'the ... of' construction, 1-4 ... words, allow punctuation
 - ← `the(\w+){1,4} of` works, but punctuation not allowed
 - ← `the(\S+){1,4} of` allows punctuation!
- ▶ Parentheses
 - ← `\(.*\)` has a critical flaw due to 'greedy matching'

Greedy match

▶ + and * are *greedy*:

- ◆ Matches the *longest* string it can. `\(.*\)` matches:

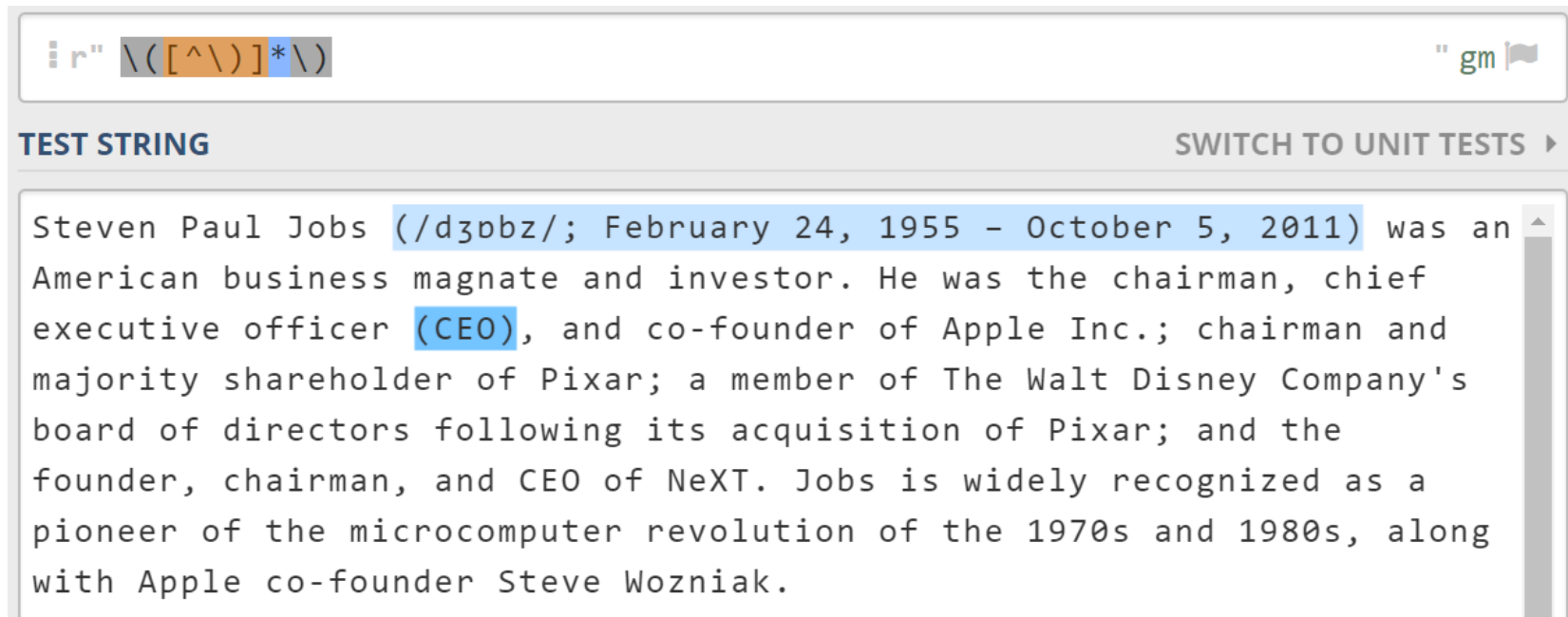


The screenshot shows a regex testing interface. At the top, the input field contains the regex `ir" \(.*\)` and the flags `" gm`. Below the input field, there are two tabs: "TEST STRING" and "SWITCH TO UNIT TESTS". The "TEST STRING" tab is active, and the text "Steven Paul Jobs (/dʒɒbz/; February 24, 1955 - October 5, 2011) was an American business magnate and investor. He was the chairman, chief executive officer (CEO), and co-founder of Apple Inc.; chairman and majority shareholder of Pixar; a member of The Walt Disney Company's board of directors following its acquisition of Pixar; and the founder, chairman, and CEO of NeXT. Jobs is widely recognized as a pioneer of the microcomputer revolution of the 1970s and 1980s, along with Apple co-founder Steve Wozniak." is displayed. The text is highlighted in blue, indicating a successful match.

Greedy match

▶ + and * are *greedy*:

- ◆ Matches the *longest* string it can.
- ◆ SOLUTION 1:
 - ◆ Instead of `.*`, exclude `)` in the middle portion with `[^\)]*`



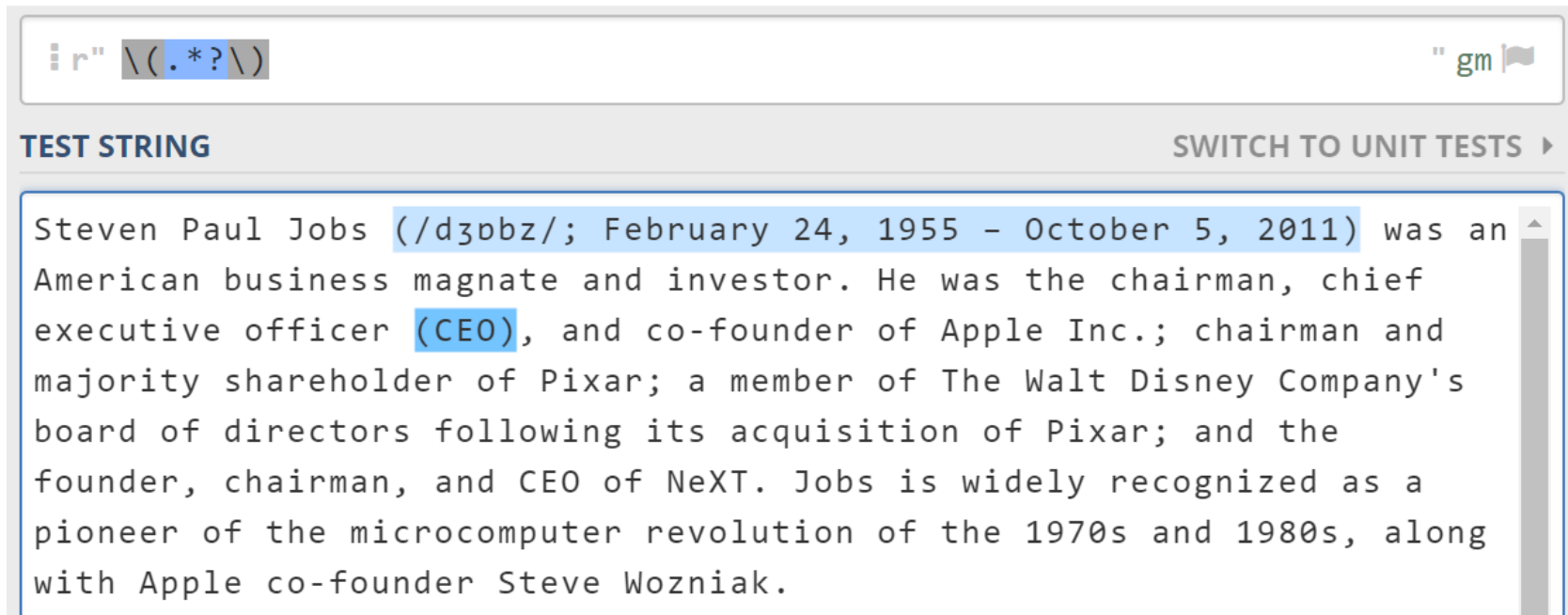
The screenshot shows a regex testing interface. The search bar contains the regex `\([^\)]*\)` with a search button labeled "gm". Below the search bar, there are two tabs: "TEST STRING" (selected) and "SWITCH TO UNIT TESTS". The test string area contains the text: "Steven Paul Jobs (/d3pbz/; February 24, 1955 - October 5, 2011) was an American business magnate and investor. He was the chairman, chief executive officer (CEO), and co-founder of Apple Inc.; chairman and majority shareholder of Pixar; a member of The Walt Disney Company's board of directors following its acquisition of Pixar; and the founder, chairman, and CEO of NeXT. Jobs is widely recognized as a pioneer of the microcomputer revolution of the 1970s and 1980s, along with Apple co-founder Steve Wozniak." The text `(/d3pbz/; February 24, 1955 - October 5, 2011)` is highlighted in blue, indicating a successful match.

Un-greedy match

▶ **+** and ***** are *greedy*:

- ◆ Matches the *longest* string it can.
- ◆ SOLUTION 2:

- ◆ To turn **+** and ***** into un-greedy, suffix **'?'** → **.+?** and **.*?**



The screenshot shows a regex testing interface. The search bar contains the regex `\/d3pbz/; February 24, 1955 - October 5, 2011`. The test string is a paragraph about Steven Paul Jobs. The search results show a match for the date range: `\/d3pbz/; February 24, 1955 - October 5, 2011`.

```
\/d3pbz/; February 24, 1955 - October 5, 2011
```

TEST STRING SWITCH TO UNIT TESTS ▶

Steven Paul Jobs (\/d3pbz/; February 24, 1955 - October 5, 2011) was an American business magnate and investor. He was the chairman, chief executive officer (CEO), and co-founder of Apple Inc.; chairman and majority shareholder of Pixar; a member of The Walt Disney Company's board of directors following its acquisition of Pixar; and the founder, chairman, and CEO of NeXT. Jobs is widely recognized as a pioneer of the microcomputer revolution of the 1970s and 1980s, along with Apple co-founder Steve Wozniak.

Using regex in Python

▶ Na-Rae's tutorials:

- ◆ <https://sites.pitt.edu/~naraehan/python3/re.html>
- ◆ [https://sites.pitt.edu/~naraehan/python3/more list comp.html](https://sites.pitt.edu/~naraehan/python3/more_list_comp.html)

▶ Official reference

- ◆ Python Regex syntax reference:
<https://docs.python.org/3/library/re.html>

▶ 're' is Python's regular expression module

- ◆ Like any other module, start by importing it:

```
>>> import re
>>>
```

Using `re.findall()`

▶ `re.findall(pattern, string)`

- ◆ Returns all matches as a list

```
>>> chom = 'Colorless green ideas sleep furiously.'  
>>> re.findall(r'e+', chom)  
['e', 'ee', 'e', 'ee']  
>>> re.findall(r'\d', chom)  
[]
```

use 'r' prefix to mark your regular expression as a **raw string** (otherwise you have to escape your \)

- ◆ 'Ignore case' option:

```
>>> foo = 'This and that and those'  
>>> re.findall(r'th\w+', foo)  
['that', 'those']  
>>> re.findall(r'th\w+', foo, re.IGNORECASE)  
['This', 'that', 'those']
```

Use this flag to match both upper and lower case
Also works: **re.I**

re.findall() and group extraction

- ▶ Use `()` to capture only a specific portion of the match

```
>>> foo = 'walked, studied and stopped.'  
>>> re.findall(r'\w+ed', foo)  
['walked', 'studied', 'stopped']
```

```
>>> re.findall(r'(\w+)ed', foo)  
['walk', 'studi', 'stopp']
```

If there is `(x)` in the expression, `.findall()` returns a list of `x`'s

```
>>> re.findall(r'(\w+)(ed)', foo)  
[('walk', 'ed'), ('studi', 'ed'), ('stopp', 'ed')]
```

```
>>> re.findall(r'((\w+)ed)', foo)  
[('walked', 'walk'), ('studied', 'studi'), ('stopped', 'stopp')]
```

Multiple `()`s: returns a list of `tuples`.

re.findall() and group extraction

- ▶ Be careful when you *need* to use ():

```
>>> foo = 'bless this mess'
>>> re.findall(r'(bl|m)ess', foo)
['bl', 'm']

>>> re.findall(r'(?:(bl|m)ess', foo)
['bless', 'mess']
```

Using () to override precedence, but group is captured

Solution:
(?:) avoids creating unwanted group capture

Regex substitution

- ▶ Use `re.sub()` method to replace matched portions with a new string.

```
>>> foo = 'walked, studied and stopped.'  
>>> re.sub(r'\w+ed', 'Xed', foo)  
    'Xed, Xed and Xed.'  
  
>>> tale = """It was the best of times, it was the worst of times,  
    it was the age of wisdom, it was the age of foolishness,  
    ...  
    being received, for good or for evil, in the superlative degree  
    of comparison only."""  
>>> print(re.sub(r'\w+ of \w+', 'CREAM of MUSHROOM', tale))  
    It was the CREAM of MUSHROOM, it was the CREAM of MUSHROOM,  
    it was the CREAM of MUSHROOM, it was the CREAM of MUSHROOM,  
    it was the CREAM of MUSHROOM, it was the CREAM of MUSHROOM,  
    it was the CREAM of MUSHROOM, it was the CREAM of MUSHROOM,  
    it was the CREAM of MUSHROOM, it was the CREAM of MUSHROOM,  
    we had everything before us, we had nothing before us,
```

Referencing group matches: \1 \2

- ▶ Once groups have been captured using `(...)` ... `(...)`, they can be referenced as `\1`, `\2`, etc.

```
>>> tale = """It was the best of times, it was the worst of times
it was the age of wisdom, it was the age of foolishness,
...
being received, for good or for evil, in the superlative degree
of comparison only."""
>>> print(re.sub(r'(\w+) of (\w+)', r'\2 of \1', tale))
It was the times of best, it was the times of worst,
it was the wisdom of age, it was the foolishness of age,
it was the belief of epoch, it was the incredulity of epoch,
it was the Light of season, it was the Darkness of season,
it was the hope of spring, it was the despair of winter,
we had everything before us, we had nothing before us,
```

Lots of "X of Y".
Change them all
to "Y of X"?

Try it yourself

3 minutes



```
>>> foo = 'This and that and those'
>>> re.findall(r'th\w+', foo)
['that', 'those']
>>> re.findall(r'th\w+', foo, re.IGNORECASE)
['This', 'that', 'those']
```

```
>>> foo = 'walked, studied and stopped.'
>>> re.findall(r'\w+ed', foo)
['walked', 'studied', 'stopped']
>>> re.findall(r'(\w+)ed', foo)
['walk', 'studi', 'stopp']
>>> re.findall(r'(\w+)(ed)', foo)
[('walk', 'ed'), ('studi', 'ed'), ('stopp', 'ed')]
>>> re.findall(r'((\w+)ed)', foo)
[('walked', 'walk'), ('studied', 'studi'), ('stopped', 'stopp')]
>>> re.findall(r'(?:\w+)ed', foo)
['walked', 'studied', 'stopped']
>>> re.sub(r'\w+ed', 'Xed', foo)
'Xed, Xed and Xed.'
```

Too short/simple?
Try with tale string

Also: how to double-
up vowel letters
using \1?

Zero-width matches: AVOID

```
>>> re.findall(r'x+', 'abc')
[]
>>> re.findall(r'x*', 'abc')
['', '', '', '']
>>> re.sub(r'b+', '-', 'abc')
'a-c'
>>> re.sub(r'b*', '-', 'abc')
'-a--c-'
>>> re.sub(r'b*', '-', 'ac')
'-a-c-'
>>>
```

Zero-width matches
can catch you
off-guard.

Different implementations
of regex may handle these
differently, even between
Python 3.6 and 3.7!

Take care to **AVOID**
composing regular
expressions that could
produce **zero-width matches**.



Compiling regular expression objects

- ▶ Constructing a regular expression (\rightarrow FSA) is computationally expensive.
- ▶ If you will be matching a regex repeatedly, pre-compiling a **regular expression object** lightens processing load.

```
>>> myre = re.compile(r'\w*e\w*')
```

myre is compiled as a **regular expression object**; `.findall()` method is directly called on it.

```
>>> myre.findall('Colorless green ideas sleep furiously.')
['Colorless', 'green', 'ideas', 'sleep']
>>> myre.findall('bless this mess')
['bless', 'mess']
>>> myre.findall('Mary had a little lamb')
['little']
```

.search()

- ▶ Sometimes, we are dealing with a whole lot of strings, and only interested in whether there *is* a match, and not in identifying all the matching parts.

- ◆ Ex: Find all lines in Jane Austen novels that contain 'so ...ly'

- ← `.findall()` is an overkill for this purpose.

- ← Use `.search()` instead.

- ▶ Using `.search()`

```
>>> chom = 'Colorless green ideas sleep furiously.'  
>>> re.search(r'e+', chom)  
<_sre.SRE_Match object; span=(6, 7), match='e'>  
>>> re.search(r'e+', chom).group()  
'e'
```


.search() method

- ◆ .search() only finds the first match and then quits.
- ◆ If successful, .search() returns a "match object" instead of a list.
 - ← The matched portion is available through .group()
- ◆ If there's no match, .search() returns None: *nothing*.

```
>>> chom = 'Colorless green ideas sleep furiously.'
>>> re.search(r'e+', chom)
<_sre.SRE_Match object; span=(6, 7), match='e'>
>>> re.search(r'e+', chom).group()
'e'
>>> re.search(r'\d', chom)
>>> ←.....
>>> re.search(r'\d', chom).group()
Traceback (most recent call last):
  File "<pyshell#146>", line 1, in <module>
    re.search(r'\d', chom).group()
AttributeError: 'NoneType' object has no attribute 'group'
```

No digit in chom;
returns None

If a match found, ... else, ...

- ▶ For obvious reasons, regular expression matches are often coupled with *if ... else*:

```
>>> chomwds = 'Colorless green ideas sleep furiously'.split()
>>> chomwds
['Colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> for c in chomwds:
...     mat = re.search(r'e+', c)
...     if mat: print('YES', mat.group(), 'in', c)
...     else: print('NO', c)
```

```
YES e in Colorless
YES ee in green
YES e in ideas
YES ee in sleep
NO furiously
```

But *mat* is a regex match object and not a True/False type. How could it work in *if ... test*?

non-Boolean "False/True" values

```
>>> if ['a'] : print('YES')
YES
>>> if [] : print('YES')
>>>
>>> if 3 : print('YES')
YES
>>> if 0 : print('YES')
>>>
>>> if -3 : print('YES')
YES
>>> if 'lala' : print('YES')
YES
>>> if '' : print('YES')
>>>
>>> if {'a':9} : print('YES')
YES
>>> if {} : print('YES')
>>>
```

- ▶ For *if* ... testing, certain non-Boolean type values are also considered **False**.
 - ◆ None (when no object is returned)
 - ◆ Number zero
 - ◆ Any empty sequence: [], (), ""
 - ◆ An empty dictionary
- ▶ Conversely, the following are considered **True**.
 - ◆ Any returned object
 - ◆ Any number other than zero
 - ◆ A non-empty sequence or dictionary

Searching in text

- ▶ Searching through a text typically proceeds line-by-line.
- ▶ Since a regex will be repeatedly matched, pre-compiling it before the iterated search is a MUST.

```
f = open('austen-emma.txt')
elines = f.readlines()
f.close()

myre = re.compile(r'(have|has|had) been', re.I)
for l in elines:
    mat = myre.search(l)
    if mat :
        print(mat.group(), '--', l, end='')
```

```
have been -- very well considering, it would probably have been better if
had been -- she felt, that pleased as she had been to see Frank Churchill,
had been -- but having once owned that she had been presumptuous and silly,
have been -- would have been a stain indeed.
have been -- of his son-in-law's protection, would have been under wretched
>>>
```

Tokenization through `re.split()`, `re.findall()`

```
>>> sent = "It's 5 o'clock somewhere. Why don't we drink a martini."
>>> sent.split()
["It's", '5', "o'clock", 'somewhere.', 'Why', "don't", 'we', 'drink',
'a', 'martini.']
>>> re.split(r'\s+', sent)
["It's", '5', "o'clock", 'somewhere.', 'Why', "don't", 'we', 'drink',
'a', 'martini.']
>>> re.split(r'[ eo]', sent)
["It's", '5', '', "'cl", 'ck', 's', 'm', 'wh', 'r', '.', 'Why', 'd',
'n't", 'w', '', 'drink', 'a', 'martini.']
>>> re.split(r'\W', sent)
['It', 's', '5', 'o', 'clock', 'somewhere', '', 'Why', 'don', 't',
'we', 'drink', 'a', 'martini', '']
>>> re.split(r'\W+', sent)
['It', 's', '5', 'o', 'clock', 'somewhere', 'Why', 'don', 't', 'we',
'drink', 'a', 'martini', '']
>>> re.findall(r'\w+', sent)
['It', 's', '5', 'o', 'clock', 'somewhere', 'Why', 'don', 't', 'we',
'drink', 'a', 'martini']
```

Regular-expression based tokenization

- ▶ Remember NLTK's plain-text corpus reader was using a different word tokenizer than `nltk.word_tokenize()`:

```
>>> import nltk
>>> help(nltk.corpus.reader.PlaintextCorpusReader)
Help on class PlaintextCorpusReader in module nltk.corpus.reader.plaintext:

class PlaintextCorpusReader(nltk.corpus.reader.api.CorpusReader)
 |   PlaintextCorpusReader(root, fileids, word_tokenizer=WordPunctTokenizer(pattern='\\w+|[^\w\s]+', gaps=False, discard_empty=True, flags=re.UNICODE|re.MULTILINE|re.DOTALL), sent_tokenizer=<nltk.tokenize.punkt.PunktSentenceTokenizer object at 0x0000020B5D61A940>, para_block_reader=<function read_blankline_block at 0x0000020B5D63D9D0>, encoding='utf8')
```

`\\w+|[^\w\s]+`

What sort of tokens does this produce?

Wrapping up

- ▶ Office hours change: Tianyi **Wed 1-3pm** (no morning hours)
- ▶ HW5 out: due Tuesday
- ▶ Next week: Morphology and FST
 - ◆ Jurafsky & Martin (2nd Ed!) Ch.3 Words and Transducers
 - ◆ Hulden (2011) Morphological analysis with FST
- ▶ What class to take in Spring? → Next slide
- ▶ PyLing!
 - ◆ Next Wednesday, 6pm, 2818 CL
 - ◆ "From Bayes to BERT: Classification Approaches in NLP" by Alejandro Ciuba



Coming soon (hopefully):

Computational Linguistics Certificate

► Pre-reqs (LING & CS shared):

- ◆ LING 1578 (phonetics), LING 1777 (syntax), LING 1682 (semantics) or LING 1267 (sociolinguistics)
- ◆ COMPINF 401 (intermediate Java), CS 445 (algorithms and data structures 1)
- ◆ STAT 1000 (applied statistics) or equivalent (such as LING 1810)

► Required content courses:

LING & CS shared:	
LING 1330 Intro to Computational Linguistics CS 1684 Bias and Ethical Implications in AI (or CS 590 for LING majors)	
LING majors/minors:*	CS majors/minors:
LING 1340 Data Science for Linguists LING 1810 Stats <i>or</i> LING 1269 Variation & Change 1 elective 1 capstone (2-3 credits)	CS 1671 Human Language Technologies CS 1571 Intro to AI <i>or</i> CS 1675 Intro to ML 1 elective 1 capstone

* Maximum of 8 credit overlap allowed with LING major/minor