Ling 131A
Introduction to NLP with Python

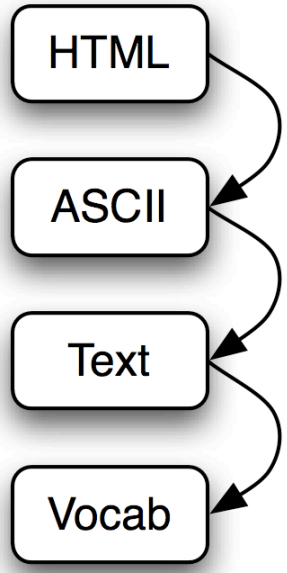
Regular Expressions

Marc Verhagen, Fall 2017

Today

- Python lab
  - Monday 1pm and Wednesday 1pm
- Assignment 3
- Assignment 4
  - WordNet and POS tagging
- NLTK Chapter 3
  - Regular Expressions

Reading HTML




```
html = urlopen(url).read()
raw = nltk.clean_html(html)
raw = raw[750:23506]

tokens = nltk.wordpunct_tokenize(raw)
tokens = tokens[20:1834]
text = nltk.Text(tokens)

words = [w.lower() for w in text]
vocab = sorted(set(words))
```

Download web page, strip HTML if necessary, trim to desired content

Tokenize the text, select tokens of interest, create an NLTK text

Normalize the words, build the vocabulary

Note: this is an image from the book, but it is not quite right

Reading HTML

The example in the book uses a URL that does not exist anymore

```
from urllib import request

url = "http://www.gutenberg.org/files/158/158-0.txt"
response = request.urlopen(url)
raw = response.read().decode('utf8')
tokens = nltk.word_tokenize(raw)
```


The example in the book uses nltk.clean_html(), but that does not exist anymore. Instead you can use BeautifulSoup.


```
$ pip3 install bs4
```

```
from urllib import request
from bs4 import BeautifulSoup

response = request.urlopen("http://nltk.org/”)
html_content = response.read().decode('utf8')
text_content = BeautifulSoup(html_content).get_text()
```

Finite State Processing

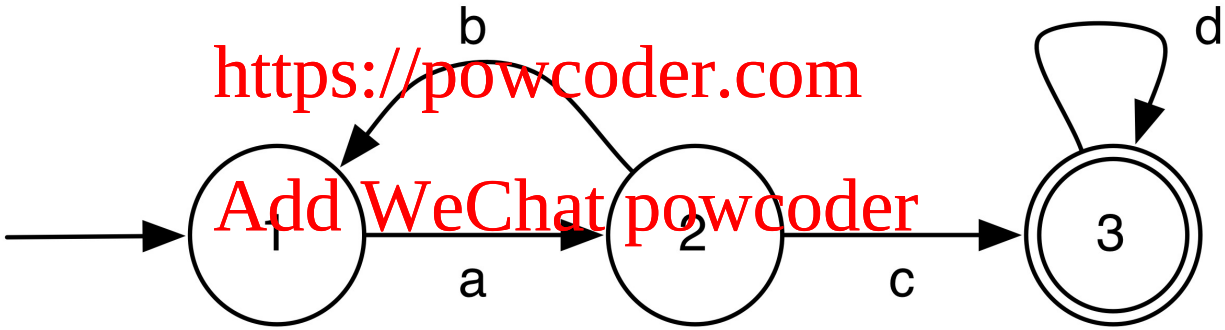
- Chomsky Hierarchy of languages/grammars

Grammar	Language	Automaton	Rules
Type-0	Recursively enumerable	Turing machine	x → y
Type-1	Context sensitive	LBA	xAy → xBy
Type-2	Context free	Pushdown automaton	A → x
Type-3	Regular	Finite state automaton	A → a; A → aB

- Difference is in complexity of grammar and power of processing automaton
- We will look at context free grammars later, but now focus just on regular grammars

Finite State Processing

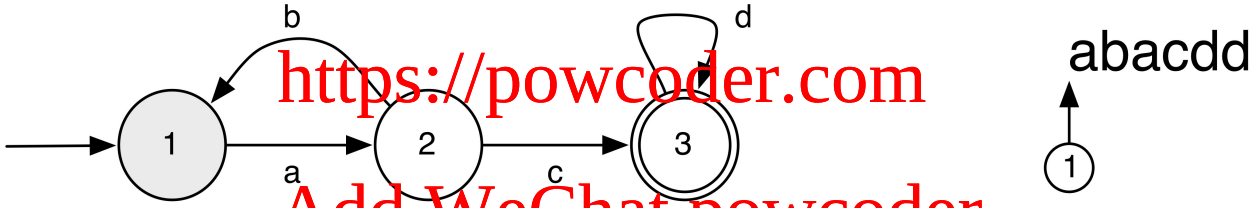
Behold a finite state automaton (FSA)



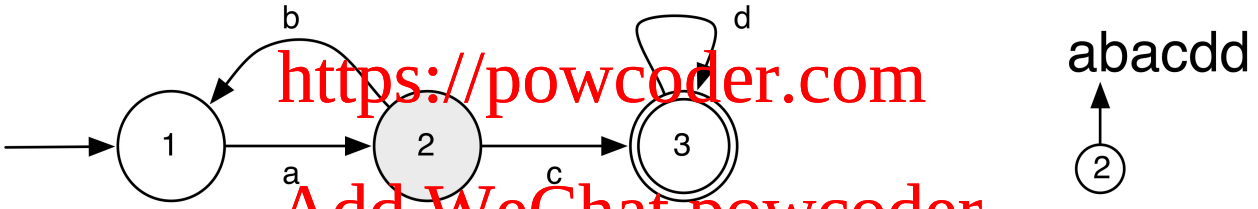
States, start state, final state(s), transitions, transition labels, alphabet of labels

Accepts all and only strings for some regular language

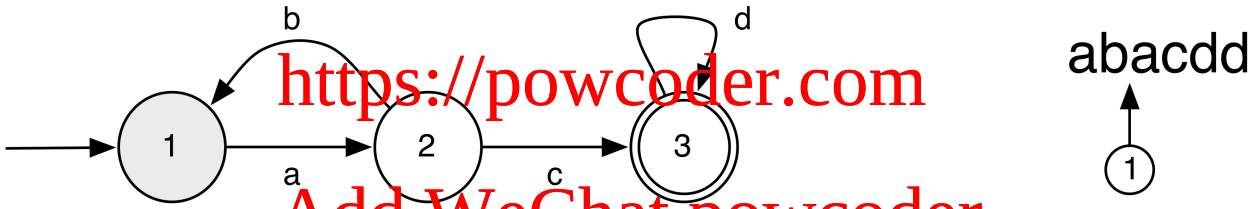
Finite State Processing



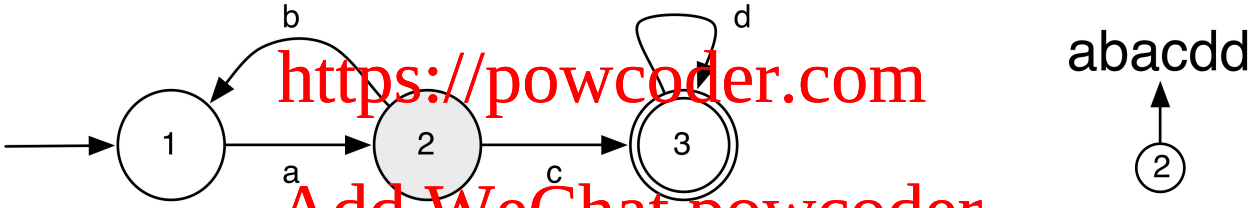
Finite State Processing



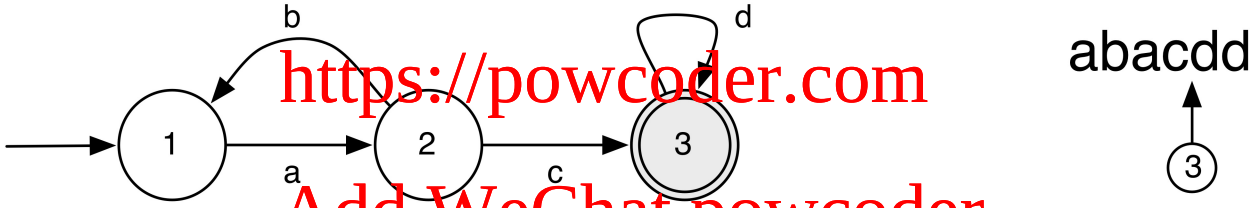
Finite State Processing



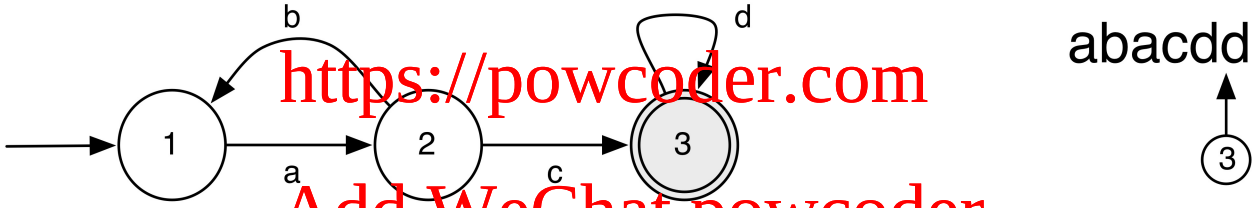
Finite State Processing



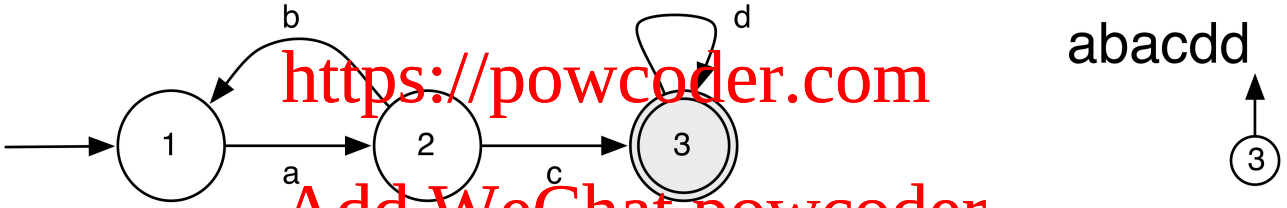
Finite State Processing



Finite State Processing



Finite State Processing



At end of string plus in a final state → success

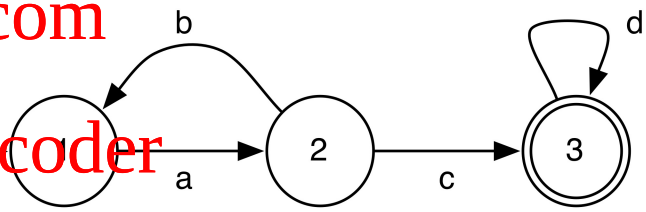
Finite State Processing

- Characteristics
  - It is finite (only a finite number of states)
  - It is fast: usually speed is to the order of the length of the input, that is O(n).
  - It only works for regular languages
  - It is the basis for regular expressions
    - In fact, you can proof that for each FSA there is an equivalent regular grammar and vice versa.
  - It is still powerful enough for many NLP tasks

Regular Expression Basics

Say you have an alphabet { a, b }

- a and b are regular expressions
- If x and y are regular expressions
  then
  - xy is a regular expression
  - x|y is a regular expression
  - x* is a regular expression
  - x? is a regular expression
  - x+ is shorthand for xx*
  - (x) is a regular expression



a(ba)*cd*

Regular Expressions

- Always defined with reference to some alphabet (Σ)
  - Generally, ASCII characters, A-Z
  - Could be smaller (Σ = {a, b, !}) or larger (Unicode)

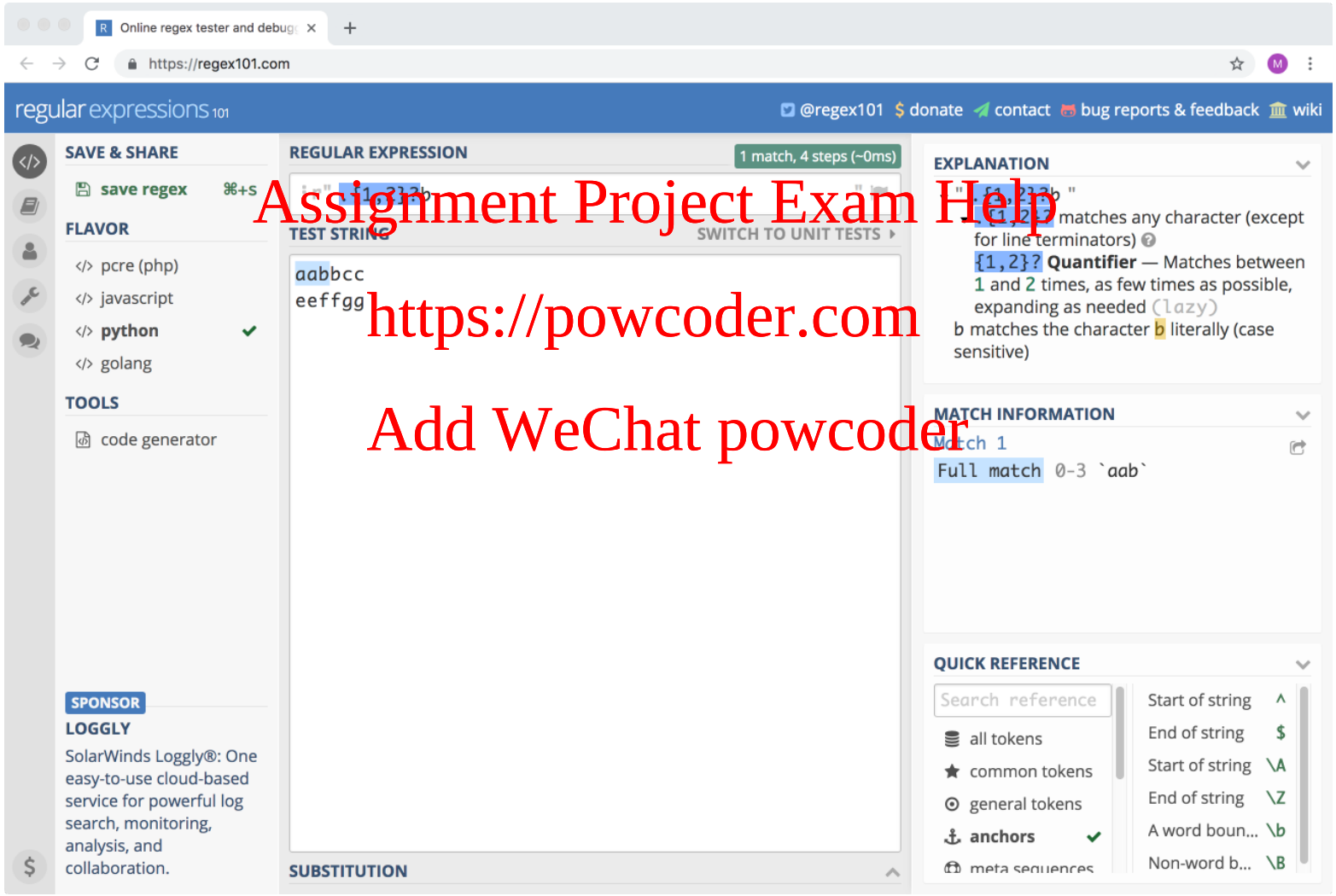
Regular expressions

- Programming language-independent
  - Has good support in Python, Perl, Java
- Practical for text processing
- Specifies members of a set, for example:
  - All strings that contain 3 capital H's
  - All strings that contain “Lemma"
  - All strings that contain an upper-case vowel
  - All strings that begin with a b, d, or g
  - All strings that end with VBZ

Applications of regular expressions

- Finding words with certain patterns
  - Can you find in nltk.corpus.treebank.words():
    - All words that end with dollar signs?
    - All words that indicate a year (e.g., 1988)?
- Low-level language processing tasks:
  - Tokenization
  - Stemming
  - Normalization

https://regex101.com/



Regular expressions

- **Basics**
  - characters and character classes
  - anchors
  - quantifiers
- Python
  - re.search() and friends
  - match object
  - groupings
  - flags

Character classes

```
[AEIOU]
```

Any one of A, E, I, O, or U


```
H[aeiou]d
```


H, followed by any one of a, e, i, o, or u, followed by d

Character classes: complements

- Any character that's not a vowel
- [^aeiouAEIOU]

In this context, ^ means "not"

Character classes

- […] matches any character contained in the list.
  - [abc] means one occurrence of either a, b, or c
- [^…] matches any character not contained in the list.
  - [^abc] means one occurrence of any character that is not an a, b, or c,
- [ABCDEFGHIJKLMNOPQRSTUVWXYZ] one upper-case unaccented letter
- [0123456789] means one digit.
- [0123456789]+\.[0123456789]+ matches decimal numbers.
- [Cc]omputer [Ss]cience
  - matches Computer Science, computer Science, Computer science, computer science.

Character classes: ranges

- All upper-case, all lower-case, all letters, any digit from zero to 9…
- [A-Z] = [ABCDEFGHIJKLMNOPQRSTUVWXYZ]
- [a-z]
- [A-Za-z] or [A-z]
- [0-9]

Pre-defined character classes

- `\d` digit, equivalent to [0-9]
- `\D` non-digit: [^0-9]
- `\s` whitespace, equivalent to [\t\n\r\f\v]
- `\S` non-whitespace, equivalent to [^ \t\n\r\f\v]
- `\w` alphanumeric character, equivalent to [a-zA-Z0-9_]
- `\W` non-alphanumeric character
- `.` Anything except for \n, or anything in multiline mode

Anchors

- `^T` line that begins with T
- `T$` line that ends with T
- `\bT` word that begins with T
- `T\b` word that ends with T
- `\AT` string that starts with T
  (similar to ^ unless in multiline mode)
- `T\Z` string that ends with T

Quantifiers

- `a{n}` `n` occurrences of “a”
- `a{n,m}` between `n` and `m` occurrences of “a”
- `a{,m}` at most `m` occurrences of “a”
- `a{n,}` at least `n` occurrences of “a”
- `a+` one or more “a”s, equivalent to `a{1,}`
- `a*` zero or more “a's”, equivalent to `a{0,}` or `a{,}`
- `a?` one “a”, or nothing equivalent to `a{0,1}` or `a{,1}`

Disjunction and Scope

- `a|b`: disjunction
- `a(b|c)+` parenthesis indicating scope of operators

What do the following do?

- [a-zA-Z]
- [A-Z][a-z]*
- P[aeiou]{,2}t
- \d+(\.\d+)?
- ([^aeiou][aeiou][^aeiou])*
- \w+|[^\w\s]+

Regular expressions

- Basics
  - characters and character classes
  - anchors
  - quantifiers
- **Python**
  - re.search() and friends
  - match object
  - groupings
  - flags

Using regular expressions in Python

- https://docs.python.org/3/howto/regex.html
- import re

Regular expression methods

- re.search(REGEXP, STRING)
  - Does STRING match REGEXP? Returns a match object
- re.match(REGEXP, STRING)
  - Does the beginning of STRING match REGEXP? Returns a match object
- re.findall(REGEXP, STRING)
  - Returns all matches of REGEXP in STRING in a list
- re.sub(REGEXP, REPLACEMENT, STRING)
  - Replaces all occurrences of REGEXP with REPLACEMENT in STRING
- re.finditer(REGEXP, STRING)
  - Return an iterator of match objects
- re.split(REGEXP, STRING)
  - Split STRING on REGEXP, return a list of substrings

Compiling regular expressions

- Up to now:
  - re.search(“[aeiou]”, “this is a string”)
  - re.findall(“[aeiou]”, “this is a string”)
- Other option: make a regular expression object


```
import re

>>> myreg = re.compile(“[aeiou]”)
>>> myreg.search(“this is a string”).group()
‘i’
>>> myreg.findall(“this is a string”)
['i', 'i', 'a', 'i’]
```

Compiling regular expressions

- search() and findall() exist
  - as functions of the re module
  - as methods of the regular expression class
- Make a regular expression object using re.compile("…")
- When should you make a compiled regular expression object?
  - Same regular expression used many times: regular expression object is faster
  - Regular expression with clear semantics: making an object as documentation

```
word = re.compile(r"\b\w+\b")
```

  - Otherwise, you might as well use the re module functions

Grouping

```
>>> line = "the 10-12 class"
>>> p = re.compile(r"the ((\d+)-(\d+))")
>>> m = p.search(line)
>>> m.group()
'the 10-12'
>>> m.group(0)
'the 10-12'
>>> m.group(1)
'10-12'
```

Grouping

```
>>> m.group(2)
'10'
>>> m.group(3)
'12'
>>> m.group(4)
IndexError
>>> m.group(2, 3)
('10', '12')
```

More methods on Match Object

```
>>> m.groups()
('10-12', '10', '12')
>>> m.start(3)
7
>>> m.end(3)
9
>>> m.span(3)
(7, 9)
```

Give names to groups

```
>>> line = "Arthur has arrived!"
>>> p = re.compile(r'\b(?P<name>\w+)\b')
>>> m = p.search(line)
>>> m.group(1)
'Arthur'
>>> m.group('name')
'Arthur'
```

Back references

```
>>> p = re.compile(r'(\b\w+\b).+(\1)')

>>> line = "test this is a test"
>>> p.search(line).groups()
('test', 'test')

>>> line = "test this is a pest"
>>> p.search(line).groups()
AttributeError
```

Search is greedy (till you tell it not to)

```
>>> line = "Sentence One. Sentence Two."
>>> p1 = re.compile(r".*\.")
>>> m1 = p1.search(line)
>>> m1.group()
'Sentence One. Sentence Two.'
>>> p2 = re.compile(r'.+?\.')
>>> m2 = p2.search(line)
>>> m2.group()
'Sentence One.'
```

Use the raw string, always

```
>>> s = "\section"
>>> m = re.search("\\section", s)
>>> m.group()
AttributeError
>>> m = re.search("\\\\section", s)
>>> m.group()
'\\section'
>>> m = re.search(r"\\section", s)
>>> m.group()
'\\section'
```

Flags to change the behavior of re

```
>>> s = "1st of November”
>>> m = re.search("""
^\s*                    # space at beginning of string
\d+(th|st|nd|rd)        # ordinal number (no commas)
\s+                     # at least one space
\w+                     # then a word
""", s, re.VERBOSE)

>>> m.group()
'1st of'
```

Flags to change the behavior of re

- re.VERBOSE, re.X:
  - whitespace in the regular expression ignored, except in bracket expressions or as \s
  - comments at end of lines, starting with #
  - longer regular expression strings, using triple quotes
  - Highly recommended for complex expressions
    - This lets you write regular expressions that are actually readable

Flags to change the behavior of re

- re.IGNORECASE, re.I:
  - Ignore case. [A-Z] is the same as [a-z]
- re.MULTILINE, re.M:
  - Use this on multiline strings, i.e. strings that contain newline.
  - ^ matches beginning of string and beginning of each line
  - $ matches end of string and end of each line

Case-sensitive and -insensitive match

```
>>> line = "Python is not python"
>>> re.findall('python', line)
['python']
>>> re.findall('python', line, re.I)
['Python', 'python']
>>> re.findall('(?i)python', line)
['Python', 'python']
```