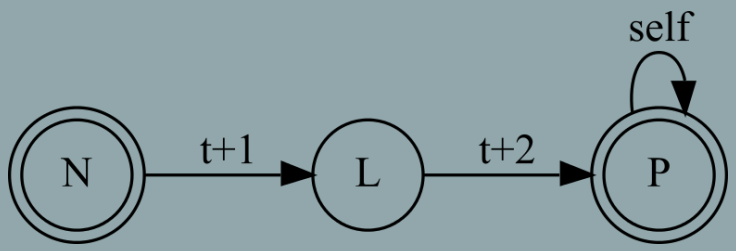
LING-362

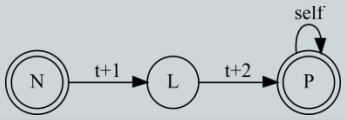
Introduction to **N**atural **L**anguage **P**rocessing

Regular Expressions



Homework assignment - overview

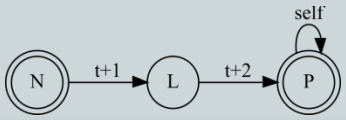
- We learned to:
  - Take some input from the user (via command line)
  - Manipulate the input (lower case, remove spaces)
  - Check some things (palindrome?)
  - Return output in some format:
    - "This is a palindrome"
    - True
- These are the makings of many/most NLP tool workflows!



About task 2

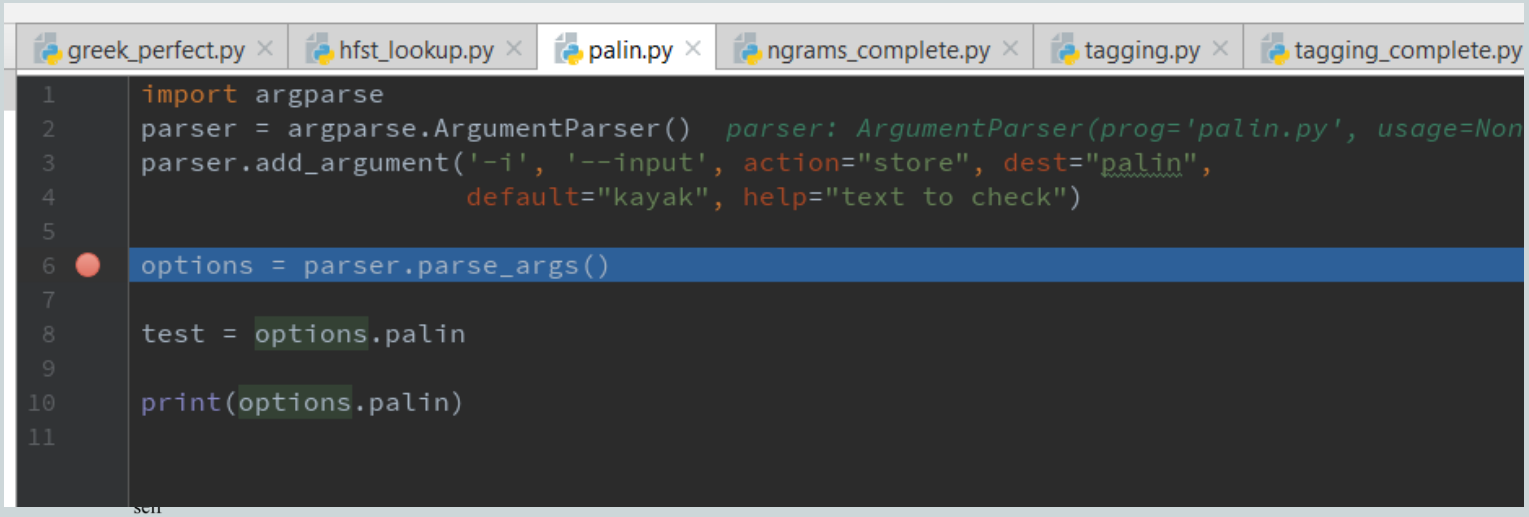
- Why is there an 'output type' argument?
- In many scenarios we'll want different output formats
  - Human readable output – good for inspecting results
  - Machine readable output – useful to plug our script into a bigger workflow as a module

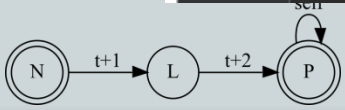
- Maybe better to think of this argument as:
  - -f / --format: "sentence" or "yes_no"



Using the debugger

- Using the debugger is a great way to see what's going on in your code
  - If you need parameters: run -> edit configurations
  - Then run -> debug will use those parameters

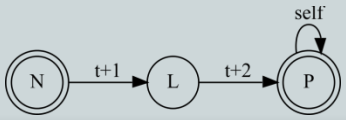




Quick review: tokenization

- You can do very rudimentary tokenization with .split():

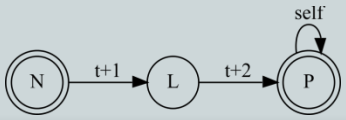
```
>>> tokens = "A Santa lived as a devil at NASA".split(" ")
>>> print(tokens)

['A', 'Santa', 'lived', 'as', 'a', 'devil', 'at', 'NASA']
```



Quick review: tokenization

- Better to use the tokenizer from the NLTK package:

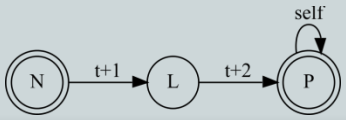
```
from nltk import word_tokenize
word_tokenize("Some words.")
```



Basic **for** loop

```
tweet = """This month you can catch a rare sight in pre-dawn sky"""

tokenized = word_tokenize(tweet)
token_count = 0
for token in tokenized:
    print(token)
    token_count = token_count + 1
print("FINISHED PRINTING " + str(token_count) + " TOKENS")
```

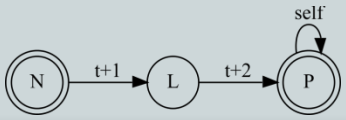
Scope of **for**



How does tokenization work?

- In part:
  - Lists of abbreviations (don't split e.g.)
  - heuristics (capital → previous period was sentence end, not an abbreviation?)

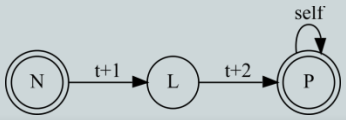
- But most importantly:
  - **Patterns:**
    - Anything with a **XXXX's**: split the genitive **s**!
    - Split/don't split hyphenated words
    - Don't split all caps acronyms with punctuation: M*A*S*H
    - How can we define these patterns?



If you already know RegEx

- Feel free to try out this challenge instead:
  - https://corpling.uis.georgetown.edu/etherpad/p/stemmer

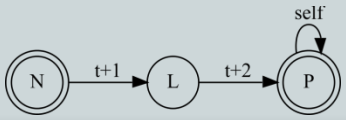
- Or, try it at home after class!



Kleene star * and plus +

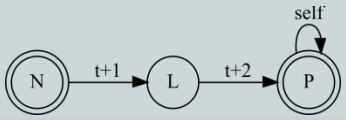
Basic operators:

- Kleene star: *
  - The previous character any number of times
  - /pizza-*time/ matches:
    - pizzatime, pizza-time, pizza--time, pizza---time …
- Kleene plus: +
  - The previous character, at least once
  - /ba+/ matches the sheep language:
    - ba, baa, baaa, baaaa …
  - Does **not** match just a single 'b'



The dot wildcard: .

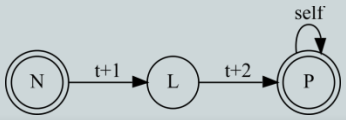
- The . stands for any character
  - /d.g/ matches: dig, dug, dog …
- Can combine with other operators:
  - /.*tion/ matches words in -tion
  - /bread.*butter/ matches: bread & butter, bread n' butter, breadbutter … (note: RegEX doesn't care about words here!)



Optional stuff: ? and |

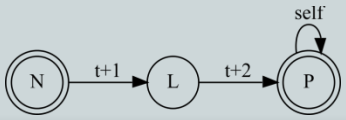
- **?** marks a previous character as optional
  - /colou?r/ matches **color** and **colour**
- **|** marks alternatives:
  - /cat|dog/ matches **cat** and **dog**
- **[]** marks a range of possible characters:
  - /b[iau]t/ matches **bit, bat, but** (not **bot**)

- Combines with **+** and ***** :
  - /[0-9]+/ a sequence of digits



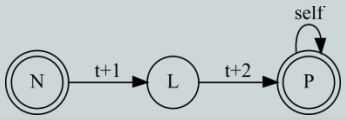
Range expressions and negation

- Some ranges can be abbreviated:
  - [a-z] any lower case character
  - [A-Z] any upper case character
  - [A-Za-z] both
  - [0-9] any digit
- Negative ranges begin with ^:
  - [^a-z] anything **other** than a lower case character
  - [^aeiou] **not** a vowel

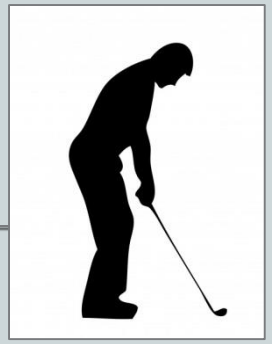


Applying operators to part of a string

- You can use **parentheses** to apply an operator to part of a string:
  - /pupp(y|ies)/ finds **puppy** or **puppies**
  - /puppy|ies/ finds **puppy** or **ies**
- Applies to other operators too:
  - /pup(py)?/ finds **pup** and **puppy**
- You can nest brackets:
  - /pup(p(y|ies))?/ finds **pup, puppy, puppies**

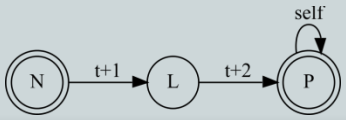


Regex Golf

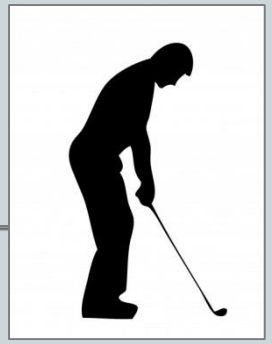


- A game – match **entire** string of these:
  - ***afoot, foody, fool***
- Do not match: ***forest, afluent, pool, foos***
- Use as few characters as possible (par: 13)

```
? * . + [] (|)
```



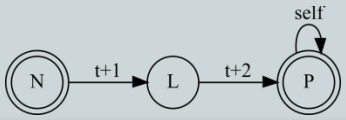
Regex Golf



- A game – match **entire** string of these:
  - ***afoot, foody, fool***
- Do not match: ***forest, afluent, pool, foos***
- Use as few characters as possible (par: 13)

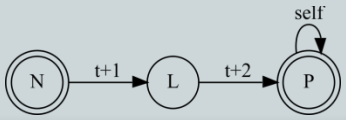
```
? * . + [] (|)
```

- Some solutions:
  - /a?foo[tdl]y?/ 12
  - /a?foo(t|l|dy)/ 13
  - /a?foo([tl]|dy)/ 14



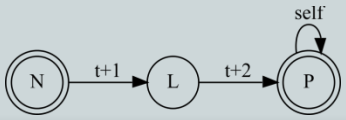
Anchoring

- You may want to find a regex only if it's the beginning or end of a string:
  - **^** - line begin
  - **$** - line end
- Examples:
  - /pup/ matches **pup**, but also **pup**il
  - /^pup$/ matches exactly **pup**
  - /^un.*/ words beginning with **un**



Escape sequences

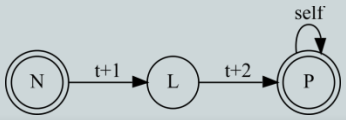
- Some characters can't be represented easily:
  - What if we want to search for an actual '?' or '.'?
- **Escape** operator characters with a backslash \
  - /\./ finds periods
  - What does this do? /U\.?S\.?(A\.?)?/
- Other special characters:
  - \n a new line symbol
  - \t a tab character



Quick review: Basic regex operators

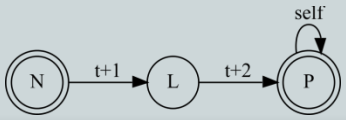
.	[ ]
+	(|)
*	[^]
?	

- What do these do?
  - [A-Z].*(shire|cester|bury)
  - [Ii]-?[Pp]hones?
  - [^aeiou0-9]+



Reading for Wednesday

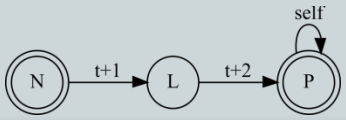
- Regular Expressions:
  - Jurafsky & Martin (2017), Section 2.1, in Canvas (Readings folder)
  - Mostly overlaps this session, but good to review
  - Note any syntax or options we haven't discussed



Using RegEx in Python

- To use RegEx we have to import a module: **re**
- Function: re.search()

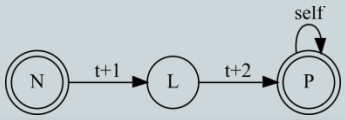
```
import re
my_data = "brambular"

# Let's go looking for sheep language…
# Is there a bV syllable in there?
match1 = re.search("(b[aeiou])", my_data)
```



Using RegEx in Python

- If the match is successful, you'll get a Match object, otherwise you get nothing, or **None**
- You can check whether the result is the special value **None**
- Use **is** and **is not** to test for **None**:

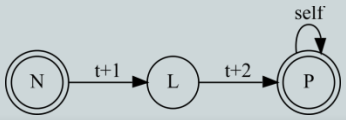
```
if match1 is not None:
  print(match1.group()) # This prints out the matching text
else:
  print("No match found")
```



Using RegEx in Python

- Can we print .group() if there's no match?

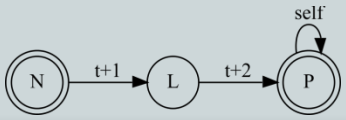
```
>>> import re
>>> match1 = re.search("(b[aeiou])","brambular")
>>> match1.group()
'bu'
>>> match2 = re.search("(b[aeiou])","bramblicious")
>>> match2.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
```



Matching groups and Python

- Why use the round brackets here?
  - match1 = re.search("(b[aeiou]).*", my_data)

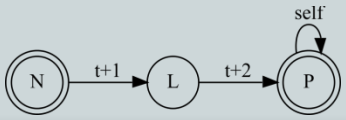
- Suppose you're interested not only in **whether** this is a sheep syllable
- You want to know **which** syllable it was
- Parts in round brackets get **saved**
- We call these: **matching groups**



Matching groups and Python

- Example: Let's say we want just the vowel:

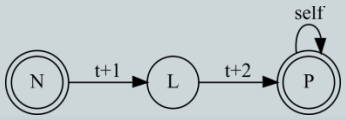
```
>>> import re
>>> match1 = re.search("b([aeiou])","brambular")
>>> match1.group() # returns whole match
'bu'
>>> match1.group(1) # return contents of first ( ... )
'u'
```



Regex substitution

- We can also replace what we find using **re.sub()**
  - Suppose we're using brackets to represent syntax
  - Can't have brackets in our string…
  - re.sub() Works like .replace() – but with full regex power!

```
>>> # Let's get rid of brackets in the input
>>> data = "A sentence (with brackets)"
>>> data = re.sub(r'\(', '-LRB-', data)
>>> data = re.sub(r'\)', '-RRB-', data)
>>> data
'A sentence -LRB-with brackets-RRB-'
```

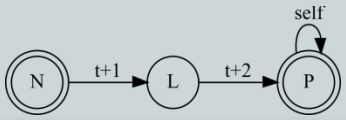


Excursus – raw strings

- What does the 'r' do?

```
>>> data = re.sub(r'\(','-LRB-',data)
```

- Remember that '**(**' is a regex operator
- Has to be escaped with **\(**
- But Python also uses **** for escaping… to use a literal backslash in the regex, we must write \\
- To search for a backslash in regex we'd write: \\\\

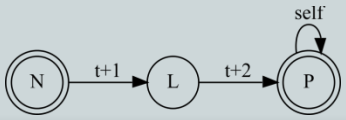


Excursus – raw strings

- This is all very cumbersome
- If we want Python to treat backslashes literally in our string and pass them on to regex, we can use **raw strings**
  - Prefixed by **r**
  - These two are the same:

```
'\\('       # Send regex a backslash to mark ( as literal
r'\('       # Same, but Python doesn't look inside
```

➢TLDR: put r before regex pattern strings

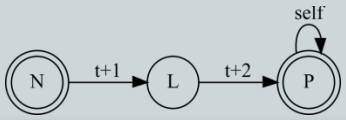


Regex substitution

- We can grab stuff out of matches with **\1**, **\2**..
- Note the raw string (otherwise: \\1, \\2)

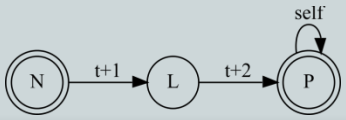
```
# Let's duplicate the b[aeiou] syllables!
>> word = "bombastic"
>> word = re.sub(r'(b[aeiou])', r'\1\1', word)
>> word
'bobombabastic'
```

- The number \1 always refers to the first part of the pattern **in brackets** (\2 is the second brackets, if available, etc.) – like group(1) etc.



Now tokenization for real

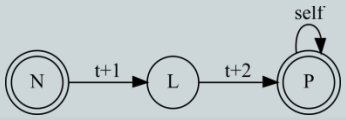
- How does NLTK's tokenizer work?
  - Actually you can almost read the code right now!
  - Here's the important stuff



nltk.tokenize.treebank (adapted)

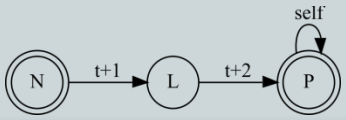
- Note the use of **def** some_function(arg1,arg2,…):

```
import re

def tokenize(text, CONTRACTIONS2, CONTRACTIONS3):

    #starting quotes
    text = re.sub(r'^\"', r'``', text)
    text = re.sub(r'(``)', r' \1 ', text)
    text = re.sub(r'([ (\[{<])"', r'\1 `` ', text)
```



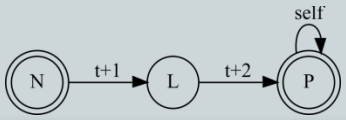
nltk.tokenize.treebank (adapted)

```
#punctuation
text = re.sub(r'([:,])([^\d])', r' \1 \2', text)
text = re.sub(r'([:,])$', r' \1 ', text)
# ... 6 more


#parentheses, brackets, etc.
text = re.sub(r'([\]\[\(\)\{\}\<\>])', r' \1 ', text)
text = re.sub(r'--', r' -- ', text)


#add extra space to make things easier
text = " " + text + " "
```



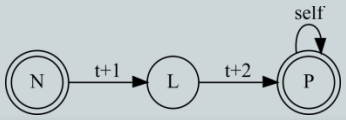
nltk.tokenize.treebank (adapted)

```
# Ending quotes
text = re.sub(r'"', " '' ", text)
# ...


# Contractions
text = re.sub(r"([^' ])('[sS]|'[mM]|'[dD]|') ", r"\1 \2 ", text)
text = re.sub(r"([^' ])('ll|'LL|'re|'RE|'ve|'VE|n't|N'T) ", r"\1 \2",
text)
```



nltk.tokenize.treebank (adapted)

```
for regexp in CONTRACTIONS2:
    text = re.sub(regexp, r' \1 \2 ', text)
for regexp in CONTRACTIONS3:
    text = re.sub(regexp, r' \1 \2 ', text)

return text.split()
```

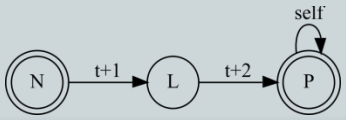


Not easy to follow, but…

- This is actual working code for entry level tokenization
- Just uses RegEx!!
- Full code here:

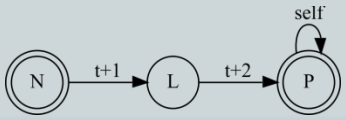
  http://www.nltk.org/_modules/nltk/tokenize/treebank.html

- More professional options out there too (notably Stanza, SpaCy, the TreeTagger tokenizer in Perl or in Python here)
- Special tools for more challenging languages



Exercise: Phone number scraper

- Download *phones.py* from Canvas
  - Fix the script so it finds all phone numbers in the text
  - Some of the steps have been done for you
  - Bonus question: can you print out all the numbers in a uniform format?



Homework

- There is a new homework script to practice RegEx in Canvas (pig_latin.py), due **Monday**
  - Instructions are in the script file/Canvas assignment