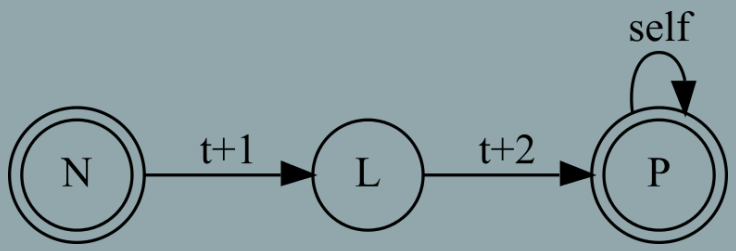
LING-362

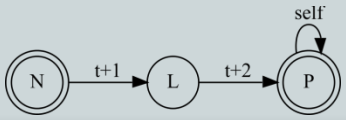
Introduction to Natural Language Processing

Tokenization and Regular Expressions I



Quick review

- if …:
  - print(x)
- elif…:
  - print(y)
- elif …:
  - print(z)
- else:
  - Do something else

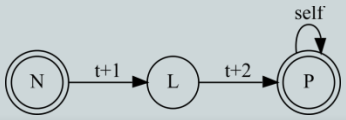


name.py

```
my_name = "Linda"
if my_name[-1] == "a":
   gender_guess = "F"
   print("ends in -a, probably female")
else:
   gender_guess = "M"
   print("does not end in -a, guess male")
```

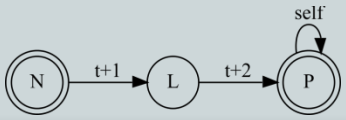
Scope of **if**

Scope of **else**



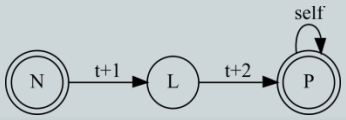
argparse – common variants

```
# Unnamed, mandatory positional argument (or 'parameter')
parser.add_argument("filename", help="file to process")
# Optional arguments (='parameters')
parser.add_argument('-v', '--verbose', action="store_true")
parser.add_argument('-l', '--lang', action="store", default="eng")


options = parser.parse_args()

filename = options.filename
if options.verbose:
        print("Processing " + filename)
        print("In language: " + options.lang)
```

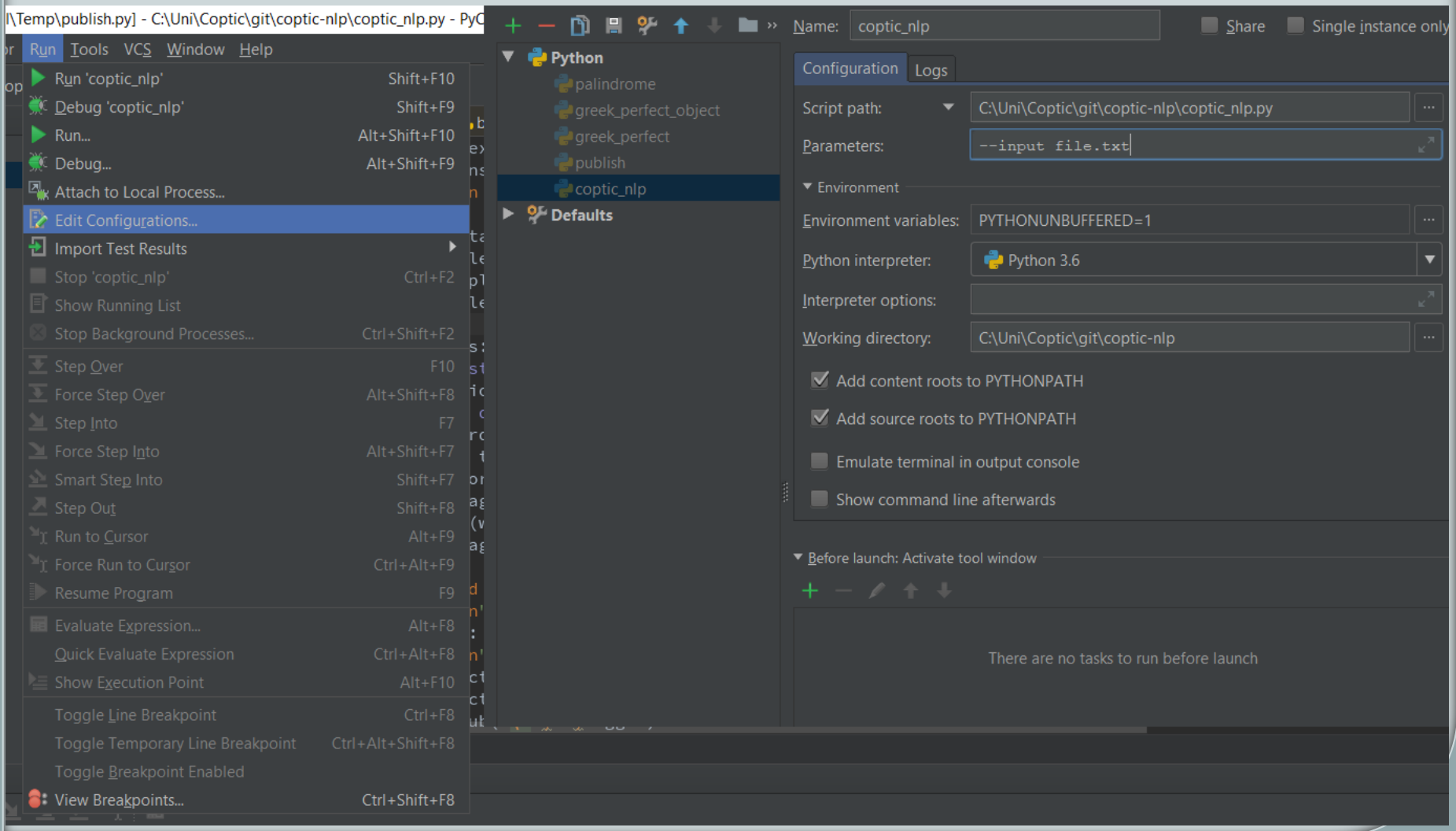


argparser – help

```
> python palindrome.py -h
usage: palindrome.py [-h] [-b] input

positional arguments:
  input           Word or phrase to test for being a palindrome

optional arguments:
  -h, --help      Show this help message and exit
  -b, --boolean   Use for boolean output, otherwise answers in plain English
```

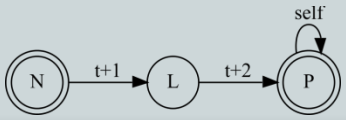


CLI arguments in PyCharm



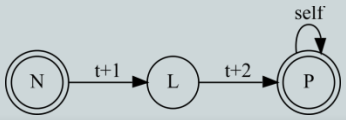
More questions from last time?

?



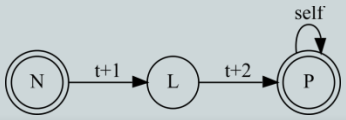
Today

- Working with words:
  - From text to tokens
  - Lists in Python
  - DIY tokenization vs. NLTK
  - Time allowing: start learning about finite-state pattern matching with **regular expressions**



Working with words

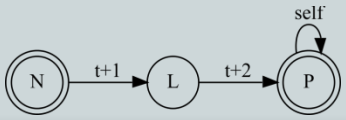
- The NLTK Text Class stores some sort of words:
  - .concordance() method displays whole words
  - .similar() method checks context in words
- How does NLTK know where words start and end in Moby Dick?



How many words are there here?

- Random Tweet:
  - *This month you can catch a rare sight in pre-dawn sky. Here are 5 things to know this week: http://nasa.tumblr.com/post/138041535009/solar-system-5-things-to-know-this-week …*

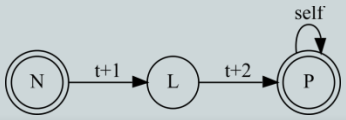
- What does your answer do for functions like **text1.similar("dawn")**?
- Implications for other applications?



Tokenization

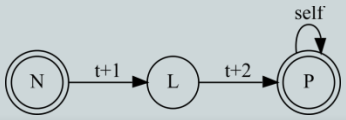
- One of the first steps in dealing with a text is dividing it into **minimal units**
  - These are the basic units for our analyses
  - All text we want to deal with is assigned to some unit
- Tempting to think of as 'words', but notice:

```
>>> text1.concordance("whale",25)
Displaying 25 of 1226 matches:
at name a whale - fish is
SWEDISH . WHALE , ICELAND
 piggledy whale statement
```



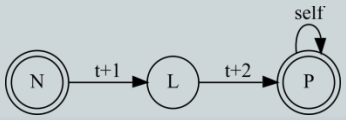
Tokenization

- NLTK treats punctuation marks the same as independent words
- Reason – once we separate out the words, something is left over:
  - "Thank you, Ambassador (Kantor),
  - → " Thank you , Ambassador (Kantor) ,
- We call all of these units **tokens**



A non-trivial form of analysis

- Tokenization is a non-trivial problem
  - Easier in languages with spaces, like English
  - Attempt to split by spaces, then split off **leading/trailing** punctuation
- Still needs disambiguation:
  - *Oranges etc. would be …*
  - *whether rivers are nearby, etc.*
- And dealing with errors:
  - *… at the same time.That we do …*

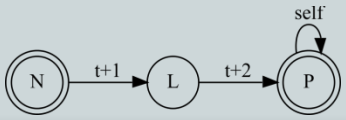


A non-trivial form of analysis

- Morpho-phonology also poses challenges:
  - *it was**n't** as bad as the pizza*
- If ***not*** is a word, should ***n't*** be one too?
  - De facto answer: yes
  - Note that this has some funny consequences:

    *I wo **n't** do it*

    → in NLP, English has a modal verb ***wo*** ☺

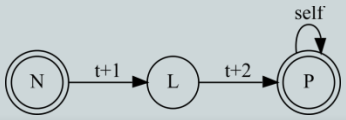


Tokenization in practice

- As a very simple **baseline**, we can use Python's **split()** method to tokenize
- Split is a method available to all strings:

```
>>> example = "This is an example"
>>> example.split()
['This', 'is', 'an', 'example']
```

- What are those brackets?



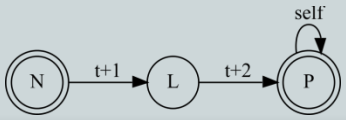
Lists

- Python uses square brackets to surround **lists**
- Variables can contain list of values:

```
>>> shopping_list = ["bread", "parsley", "beans"]
>>> number_list = [0, 1, 1, 2, 3, 5]
>>> mixed_list = ["why", "not", 10]
```

- You can also get the nth member of a list **(zero indexed!)**

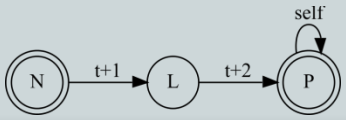
```
>>> shopping_list[1]
'parsley'
```



Lists – positions with [n:m]

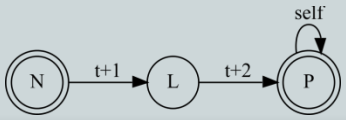
- Actually, our string character extraction used lists as well – strings contain lists of characters:

```
>>> "parsley"[0]
'p'
>>> "parsley"[0:2]
'pa'
>>> shopping_list[1]
'parsley'
>>> shopping_list[1][0]  # Char position in list position…
'p'
```



Lists – append() and remove()

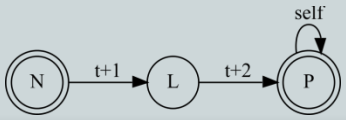
```
>>> shopping_list.append("broccoli")
>>> shopping_list
['bread', 'parsley', 'beans', 'broccoli']
>>> shopping_list.remove("bread")
>>> shopping_list
['parsley', 'beans', 'broccoli']
>>> shopping_list.remove("celery")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```



Back to split()

- .split() gives us a list of strings from a single string
- By default the separator is assumed to be space, but it can be specified:

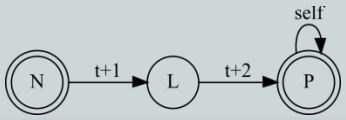
```
>>> sentence = "This is an example"
>>> sentence.split()
['This', 'is', 'an', 'example']

>>> server = "corpling.uis.georgetown.edu"
>>> server.split(".")
['corpling', 'uis', 'georgetown', 'edu']
```



Not enough for tokenization

Let’s try using .split() on this string variable:

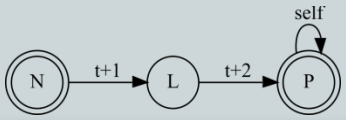
sent = **"the lawyer's group, chaired by Lefcourt, will deal with this."**



Not enough for tokenization

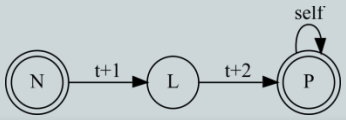
```
>>> sent = "the lawyer's group, chaired by Lefcourt, will deal with this."
>>> sent.split()
['the', "lawyer's", 'group,', 'chaired', 'by', 'Lefcourt,', 'will', 'deal', 'with', 'this.']
```

- We need some way to define more complex patterns to split off
- These will be language (genre?) dependent



Existing tools

- We will learn how automatic tokenization can be coded from scratch in just a bit!
- But first let's look at an off-the-shelf tokenizer
  - NLTK comes with a built-in tokenizer for English
  - Doesn't always get things right (when?)
  - But a pretty good approximation



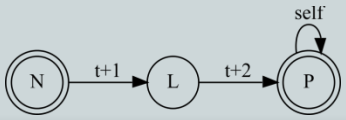
nltk.word_tokenize()

- You can use the tokenization function as a method of the nltk package:

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

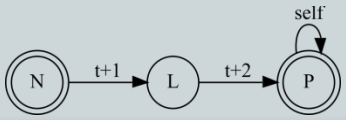
- Or import it directly – both work the same:

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



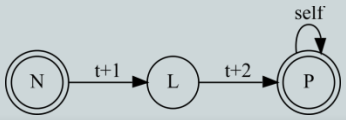
Let's try this out!

- Go online and see how good NLTK is!
  - Pick some potentially messy content like:
    - News
    - Twitter/FaceBook/Reddit…
    - Weather report (numbers, degrees and all)
    - …
  - Paste the text into a new script file in PyCharm:
    - Surround it with **triple** quotes and assign to a variable:
      - my_text = """some really long text… """
    - import **nltk** and use **word_tokenize()** on that variable
    - Print the result



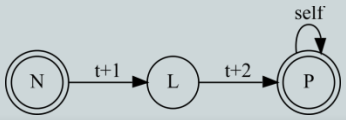
Warning 1: Why triple quotes?

- You can use double or single quotes to protect quotes in Strings:
  - sentence1 = 'I said "hello" next'
  - sentence2 = "This is Ben's cousin"
- But what if you have both? → Triple quotes:
  - Sentence3 = """This one can contain "normal" use of quotes even if you don't avoid single quotes"""



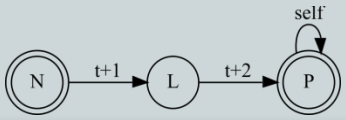
Examples

- Weather:
  - Elev 90 ft 38.91 °N , 77.01 °W | Updated 4 min ago Overcast Overcast 49.4 °F Feels Like 49 °F N1.0 Wind from South Gusts 6.0 mph Tomorrow is forecast to be MUCH COOLER than today .
- Tweet:
  - This month you can catch a rare sight in pre-dawn sky . Here are 5 things to know this week : http : //nasa.tumblr.com/post/138041535009/solar-system-5-things-to-know-this-week …



Outputting lists more readably

- If you just print a list, you get a rather unreadable output
- And possibly no support for Unicode in your terminal:
  - ['Elev', '90', 'ft', '38.91', '\xc2\xb0N', ',', '77.01', '\xc2\xb0W', '|', 'Updated', '4', 'min', 'ago', 'Overcast', …]

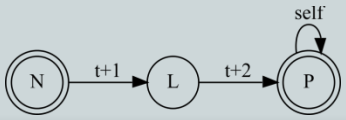


Basic **for** loop

- A nicer way is to **loop** through the list and output each token in a separate line:

```
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**

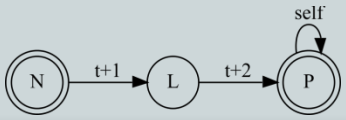


(Code also in Canvas > Files > Code > tokenization

How can we catch the errors?

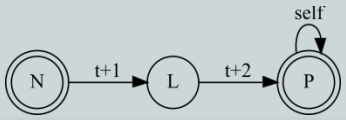
- What would you need to know to do tokenization right?
  - Lists (e.g. abbreviations) – bonus challenge in Canvas!
  - Heuristics (e.g. capitalization)
  - Patterns (contractions, URLs, …)

- We can check heuristics like capitalization (if word[0]…) – but how do we use "patterns"?



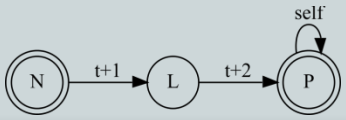
Patterns using regular expressions

- A general way to define simple textual patterns (for tokenization or other purposes):
  - Suppose you want to find all prices in a collection of computer hardware descriptions
  - Maybe you want to split sum and currency…
  - But prices come in many shapes:
    - $999.99
    - €450
    - 300 Dollars
    - …
  - Regular expressions (RegEx) can capture these!



Regular expressions

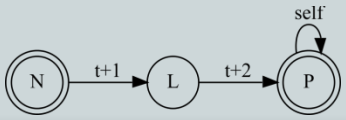
- A regular expression can contain:
  - Characters to search for
  - Operators – special symbols
  - Reserved symbols (e.g. 'any digit' or 'beginning of line')
- By convention RegEx is often given between slashes (Perl syntax – not actual Python):
  - "hello" A string of characters, exactly 'hello'
  - /hello/ A RegEx matching only the pattern 'hello'



Kleene star * and plus +

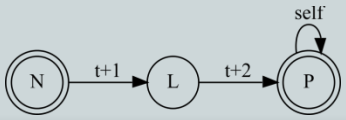
Basic operators:

- 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: .

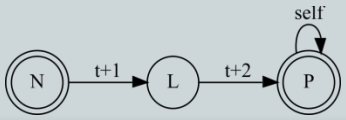
- The . stands for any character
  - /d.g/ matches: dig, dug, dog …
- Can combine with other operators:
  - /.*tion/ matches words in -tion
  - /bread.*butter/ matches: bread & butter, bread n' butter, breadbutter … (note: RegEX doesn't care about words here!)



Optional stuff: ? and |

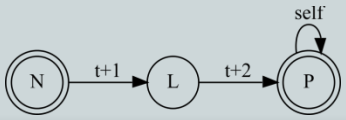
- **?** marks a previous character as optional
  - /colou?r/ matches **color** and **colour**
- **|** marks alternatives:
  - /cat|dog/ matches **cat** and **dog**
- **[]** marks a range of possible characters:
  - /b[iau]t/ matches **bit, bat, but** (not **bot**)

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



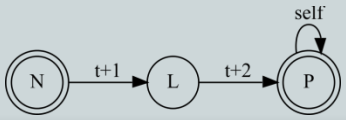
Range expressions and negation

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

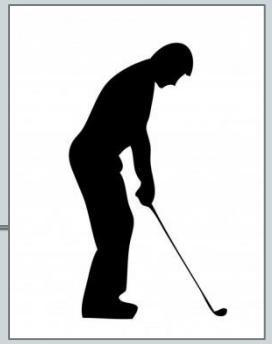


Applying operators to part of a string

- 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**

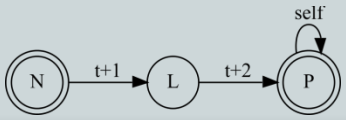


Regex Golf

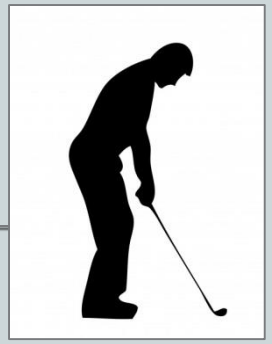


- A game – match **entire** string of these:
  - ***afoot, foody, fool***
- Do not match: ***forest, afluent, pool, foos***
- Use as few characters as possible (par: 13)

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



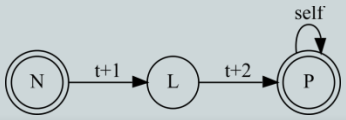
Regex Golf



- 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)

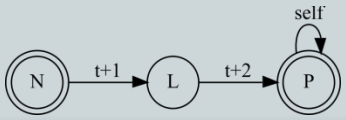
```
? * . + [] (|)
```

- 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