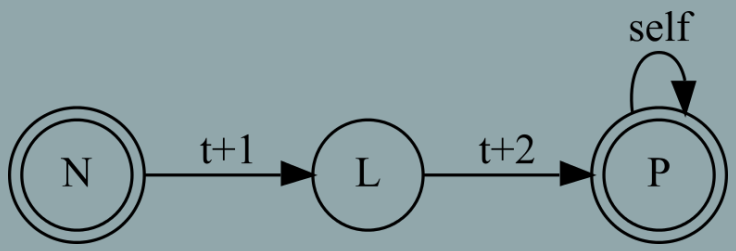
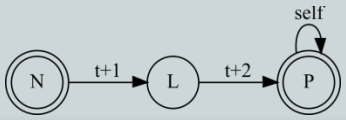
LING-362

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

Python & NLTK basics II

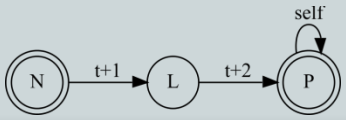


Questions from last time



Quick review

- Math:
  - 4 + 3
  - 5 ** 2
- Variable type conversions:
  - int(4.0) → 4
  - float(4) → 4.0
  - str(4) → "4"



Quick review

- Strings:
  - "hello" + " " + "world"

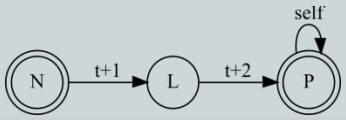
  'hello world'
  - "bye" * 2

  'byebye'
  - "bye"[0]

  'b'
  - "bye"[-1]

  'e'
  - "bye"[0:-1]

  'by'

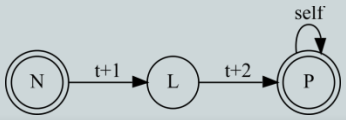


Quick review

- Booleans:
  - 6 > 5

  True
  - 6 == 5

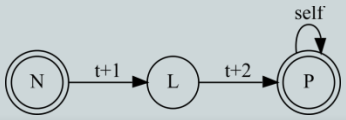
  False



NLTK – a quick taste

- With the resources installed, we can play with some texts:

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```



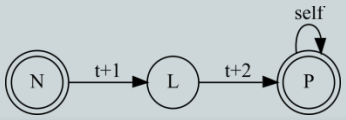
Imports

- You can import an installed library like this:

```
>>> import nltk  # Now we can access the nltk object
>>> nltk.__version__  # Get the __version__ attribute
'3.6.2'
```

- We can also import contents of specific submodules:

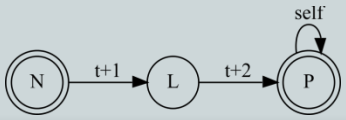
```
>>> from nltk.book import *  # all contents of nltk.book
```



Imports

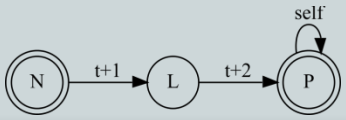
- Or even specific objects and functions:

```
>>> from nltk.book import text1
>>> print(text1.name)
'Moby Dick by Herman Melville 1851'
>>> print(text2.name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'text2' is not defined
```



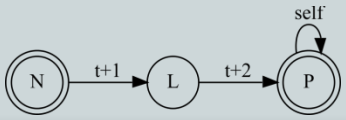
NLTK – a quick taste

- What are these texts we're importing?
  - Objects of the type or **Class *Text***
  - A 'customized' data type – we will learn a lot about these
  - Objects represent encapsulated, somewhat autonomous pieces of code with specified functionality
- Objects have:
  - **Properties** or **attributes** text1.name
  - **Functions** or **methods** text1.concordance("ship")



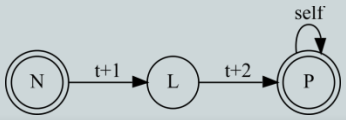
Methods and arguments

```
>>> text1.concordance("harpoon",10,10)
Displaying 10 of 76 matches:
risk a harpoon
And that harpoon
a tall harpoon
…

>>> text1.concordance("harpoon",lines=100)
Displaying 76 of 76 matches:
hen they were nigh enough to risk a harpoon from the bowsprit ? Now having a ni
n a sunrise and a sunset . And that harpoon -- so like a corkscrew now -- was f
 over the fire - place , and a tall harpoon standing at the head of the bed . B
```



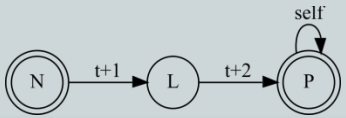
First notions about OOP

- A major line of thought for Object Oriented Programming (OOP)
  - Objects are encapsulated
  - They do their job and expose methods
  - We don't know (and don't want to know) ***how***
- Advantages:
  - Others can import our objects without studying our code
  - Possible to improve our objects without altering their **interface** to the outside



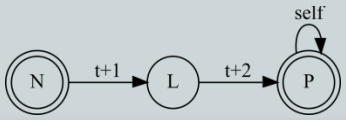
Procedural vs. OOP

- Our Greek perfect program is an example of a **procedural program** (procedural ≠ OOP)
  - Functions or 'procedures' run in sequence
  - Not bundled into opaque 'objects'
- What kind of object would we use to contain our Greek verb?
  - What properties would the object have?
  - How would we get the perfect / present form?
  - What else could it do?



Procedural vs. OOP

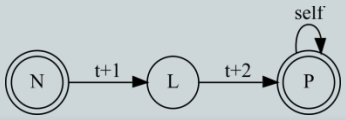
- Bonus exercise: (intermediate pythonistas!)
  - Look at ***greek_perfect_object.py*** in Canvas
- This is a toy implementation of a GreekVerb class (a custom data type for Greek verbs!)
  - There are a lot of things here we haven't learned yet…
  - Try stepping through the code in the debugger
- Point for further thinking:
  - This is a lot more code than ***greek.py***
  - Why is this useful? Is it worth it?



Another example: *.similar(word)*

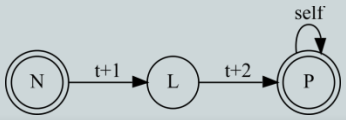
- The Class Text has a *similar* function with the **signature**: .similar(word, num=20)
- It gives you ***num*** distributionally similar words
  - How?
  - Who cares: The Text Class takes care of this for us
  - If we have a better method to do this next week, we'll release a new version of the *Text* Class

- This is nice and correct in principle… but stay vigilant!



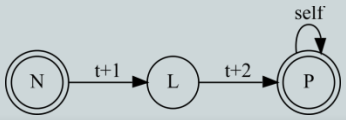
Another example: *.similar(word)*

```
>>> text1.similar("boat",num=4)
whale ship head sea
>>> text1.similar("Ahab",num=4)
it he that queequeg
>>> text1.similar("crew",num=4)
whale ship head boat
>>> text1.similar("harpoon",num=4)
whale boat ship sea
```



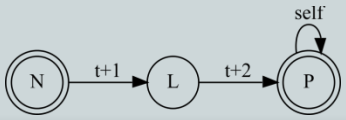
A (slightly) more serious program

- For our next exercise, we will build a program to check whether our input is a **palindrome:**
  - dud
  - kayak
- Or not:
  - bud
  - magic



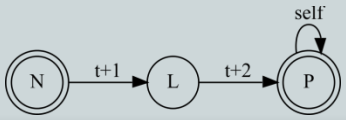
Palindrome checker

- Thinking about input and output:
  - IN: a string of characters
  - OUT:
    - An answer in English (String)
    - True or False (Boolean)



Some starter code - *if*

```
test = "kayak"

# Ideally we'd want something like this:
if test_is_a_palindrome:
        print("The input '" + test + "' is a palindrome")
else:
        print("The input '" + test + "' is not a palindrome")


# We need to learn more…
```

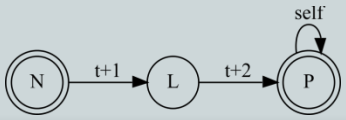


A word about indentation

- To know what to do '*only* ***if*** X' Python uses indentation:

```
x = 5                    # Not indented, always do this part
y = user_input           # Also not indented
if x > y:  # Not indented, since this check always happens
   print("it's bigger!") # This is indented – only do if x>y
```

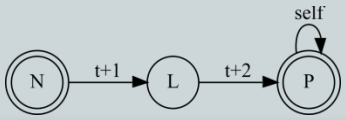
- In other words, indentation determines the **scope** of the **if** statement



If, else and elif

- You can also test multiple alternatives:

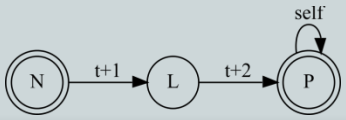
```
x = 5
y = user_input
if x > y:
   print("it's bigger!")
elif x < y:
   print("it's smaller!")
else:
   print("it's the same!")
```



Quick exercise – imagine it's snowing!

- Hurray! Snow!!
- What will this code print?

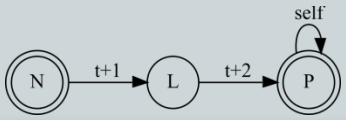
```
snow_inches = 40                # Current snow level
campus_open_max = 55            # Level at which campus closes
tomorrow_min = 10               # Minimum projected snowfall tomorrow
tomorrow_max = 20               # Maximum project snowfall tomorrow

if snow_inches + tomorrow_max < campus_open_max:
    print("campus will definitely be open")
elif snow_inches + tomorrow_min < campus_open_max:
    print("campus might be open")
else:
    print("campus will definitely be closed")
```



Another word about indentation

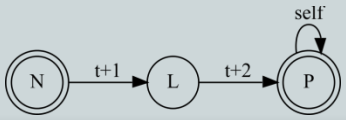
- Python accepts two ways of indenting:
  - Initial spaces, often 4 (sometimes 2 are used)
  - Tabs
- **PEP8** recommends 4 spaces
- But many developers use tabs (esp. outside US)
- I will accept either, but no mixing!

- What is PEP?



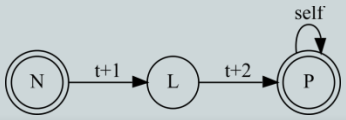
Conventions and names

- It is a good idea to use informative names for variables and functions
- To document your code inside your scripts
  - Helps others work with your code
  - Helps you to remember what you were doing
  - Allows creation of automatic documentation
- High quality code is easy to maintain – but what conventions should we use?



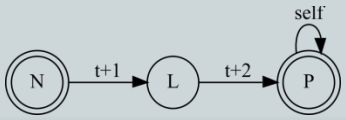
PEP

- Python is developed using the Python Enhancement Proposal (PEP) process
  - Enhancements to the language in newer versions (e.g. adding new operators, built in functions…)
  - Various **recommendations**
- Crucial for maintaining readable code:
  - PEP 0008: Style Guide for Python Code https://www.python.org/dev/peps/pep-0008/
  - PyCharm automatically checks PEP8 compliance
  - We will follow PEP8 in our assignments



Some PEP8 basics

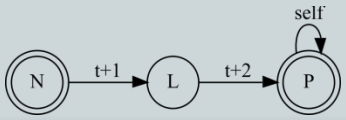
- Variables and functions should have informative, lower case names:
  - ✓ word_count ✗ wdct
  - ✓ find_nouns() ✗ findnn(), FindNouns()
- White space around operators:
  - ✓ count = previous + 1
  - ✗ count=previous+1
- Line length should be 79 characters maximum
- Nice overview: https://realpython.com/python-pep8/



Indentation and hierarchy

- Note that indentation is hierarchical:

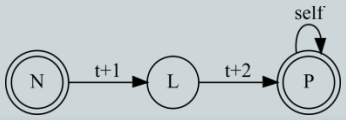
```
if x > y:                # Not indented, always happens.
→if x > y * 2:           # Indented, happens if x > y
→→print("it's a lot bigger!")     # Indented twice!
→else:
→→print("it's a bit bigger")     # Indented twice!
else:
→print("it's smaller")
```



Exercise – checking things

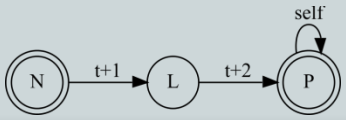
- Suppose we have a variable *my_name*, which holds a name
- We want to guess gender based on the name and use a very simple heuristic:
  - Name ends in -a: *gender_guess* = "F"
  - Otherwise: *gender_guess* = "M"

- How would the code look to check *my_name* with the value "Linda"?
  - Remember how to look up the last character in a string…
  - Remember the difference between = and ==
  - *(Solution also in Canvas)*



Checking things

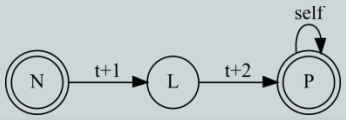
```
my_name = "Linda"
if my_name[-1] == "a":
   gender_guess = "F"
   print("ends in -a, probably female")
else:
   gender_guess = "M"
   print("does not end in -a, guess male")
```



How can we check the palindrome?

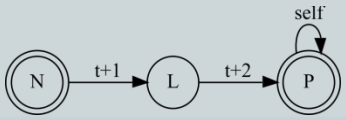
- If something is a palindrome, then it is identical to its reverse
- How can we reverse a string?

```
>>> 'hello world'[::-1]
'dlrow olleh'
>>> # This means: Go through entire string: 'bla'[:]
>>> # Do it in steps of -1: 'bla'[::-1] == 'alb'
```



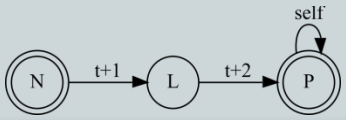
Full palindrome checker V1

```
test = "kayak"

# Get the reverse of the input string
reversed_test = test[::-1]

if test == reversed_test:
    print("The input '" + test + "' is a palindrome")
else:
    print("The input '" + test + "' is not a palindrome")
```



How to give our program parameters?

- Changing the variable ***test*** in code every time you want to check input is not an option
- Users must be able to run the program without altering code
- We need **input parameters** or **'arguments'**



Unnamed arguments

- When we run a Python script from the command line we can get anything written after the script name like this:

Command line terminal:

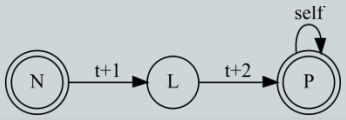
> python my_script.py arg1 arg2 arg3

Script:

```
import sys
print(sys.argv)
```

Output:

```
['my_script.py', 'arg1', 'arg2', 'arg3']
```

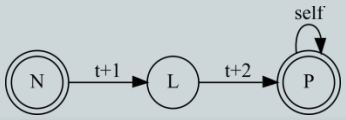


Unnamed arguments

- But there is no structure to these:

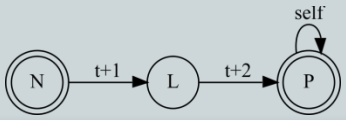
  ['test.py', 'arg1', 'arg2', 'arg3']

  - They don't have names
  - Must be in a specific order
  - Must all be present
  - No helpful message for the user what input is allowed
  - We could make a better way… or **import** one!



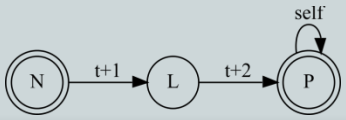
Building block: argparser

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-i', '--input', default="kayak", help="text
to check")

options = parser.parse_args()

word_to_test = options.input
```



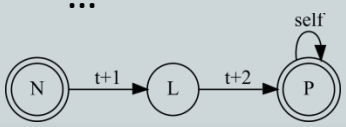
argparser – other options

```
# Unnamed, mandatory positional argument

parser.add_argument("filename", help="file to process")


options = parser.parse_args()


filename = options.filename
```



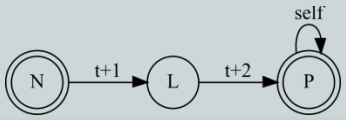
argparser – other options

```
# Boolean options (neater than saving ‘yes’ or ‘true’
# Compare:
parser.add_argument("--parse","-p", action="store_true",
        help="file to process")
parser.add_argument("--doparse","-d", action="store",
        default="yes",help="file to process")

options = parser.parse_args()

if options.parse:
 …

if options.doparse == "yes":
 …
```



argparser – help

```
> python heb_pipe.py -h
usage: python heb_pipe.py [OPTIONS] files

positional arguments:
  files                 File name/pattern of files to process (e.g. *.txt)

optional arguments:
  -h, --help            show this help message and exit

standard module options:
  -w, --whitespace      Perform white-space based tokenization
  -t, --tokenize        Tokenize word forms into morphological segments
  -p, --pos             Do POS tagging
  -l, --lemma           Do lemmatization
  -m, --morph           Do morphological tagging
  -d, --dependencies    Parse with dependency parser
  -e, --entities        Add entity spans and types
  -c, --coref           Add coreference annotations
  -s {auto,none}, --sent {auto,none}
                        XML tag to split sentences, e.g. sent for <sent ..>
                        (otherwise automatic sentence splitting)
```



Homework – due next Wednesday

- You will find the rudimentary ***palindrome.py*** in Canvas
  - In the comments you will find 4 tasks:
    - Add 2 parameters for input and True/False output mode
    - Add handling for capitalization
    - Add handling for spaces in input
  - The comments will guide you and provide some sample inputs
    - If your inputs go through correctly – all is well!
    - If not – try debugging in PyCharm first!
    - Ask Janet and me for help, and come to office hours!