# **Lecture 15:** Trees

July 19th, 2022

Cooper Bedin

# Announcements

- Lectures are no longer linked directly from the website—apparently the link system is super buggy and the IT folks yelled at us because students were submitting a bunch of ticket requests
    - You should go to the bcourses media gallery directly to access lectures
- Lab 6: sections happening today, due tomorrow
- HW 4 released, due Thursday
- CATS due tomorrow, or tonight for EC, or by Thursday for 75% credit
- Next project (Ants) will be released once CATS is due
- We'll also be releasing an extra-credit mini-project called Magic: The Lambda-ing
- MT scores released very soon—when they are posted please check Piazza for how to submit regrade requests

# Trees

# Tree class definition

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = []

    def is_leaf(self):
        return self.branches == []

    # some more stuff also (__str__, __repr__, etc.)
```

`self.label` can be whatever you want—common data types you'll see in this class are integers and strings, but really it can be anything!

`self.branches` is *always* a list of `Tree` objects—this means that trees are recursive by definition :0

# Visualizing Trees

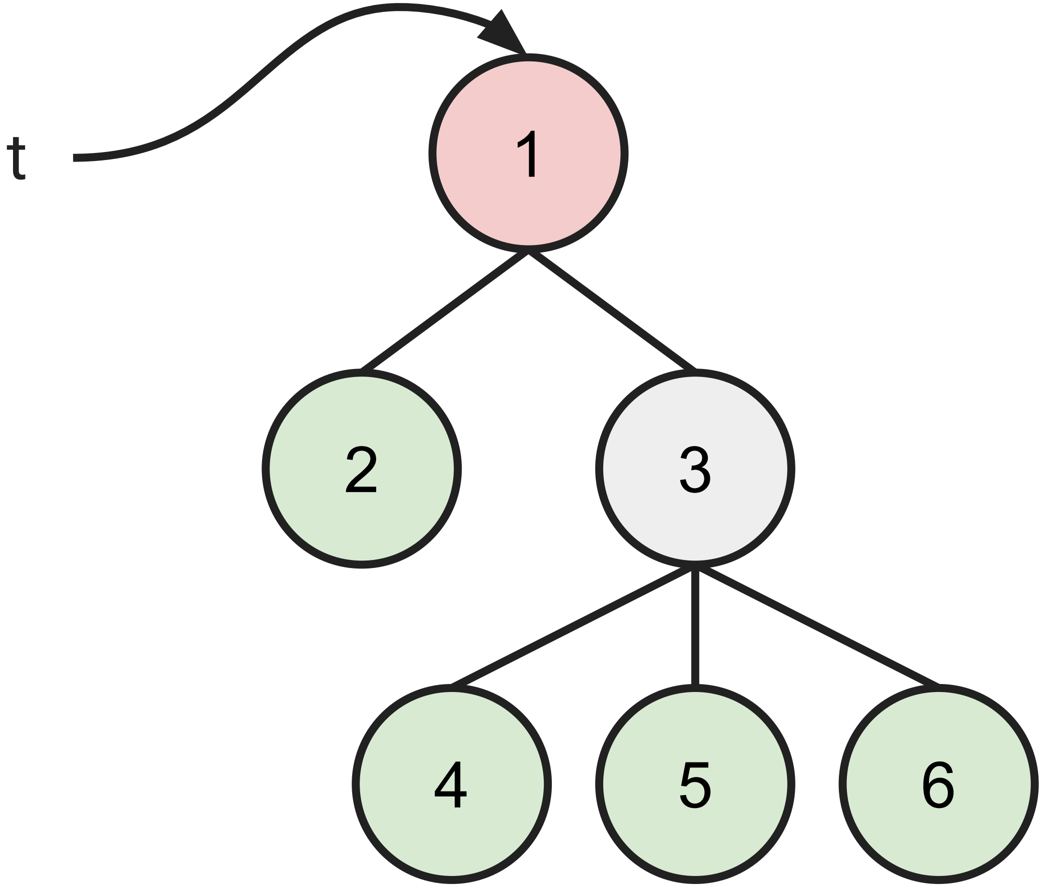
Demo

# Tree terminology

Let's say we have a variable `t` that is pointing at an instance of the `Tree` class

- The variable `t` is pointing at the **root** of our tree
- The class is called `Tree`, but we often refer to the entire, larger data structure as a **tree**, and refer to the individual `Tree` objects contained within it as **nodes**
- `t.label` allows us to access the **label** of `t`—the basic premise of trees as a data structure is that each individual node remembers a small piece of inforation
- Any of the trees stored in `t.branches` are called the **branches** or **children** of `t`
- `t` can also be called the **parent** of any of its immediate branches
- Any node or tree without any branches is called a **leaf**—we can verify whether a Tree instance is a leaf by using the `is_leaf` method

## Tree terminology

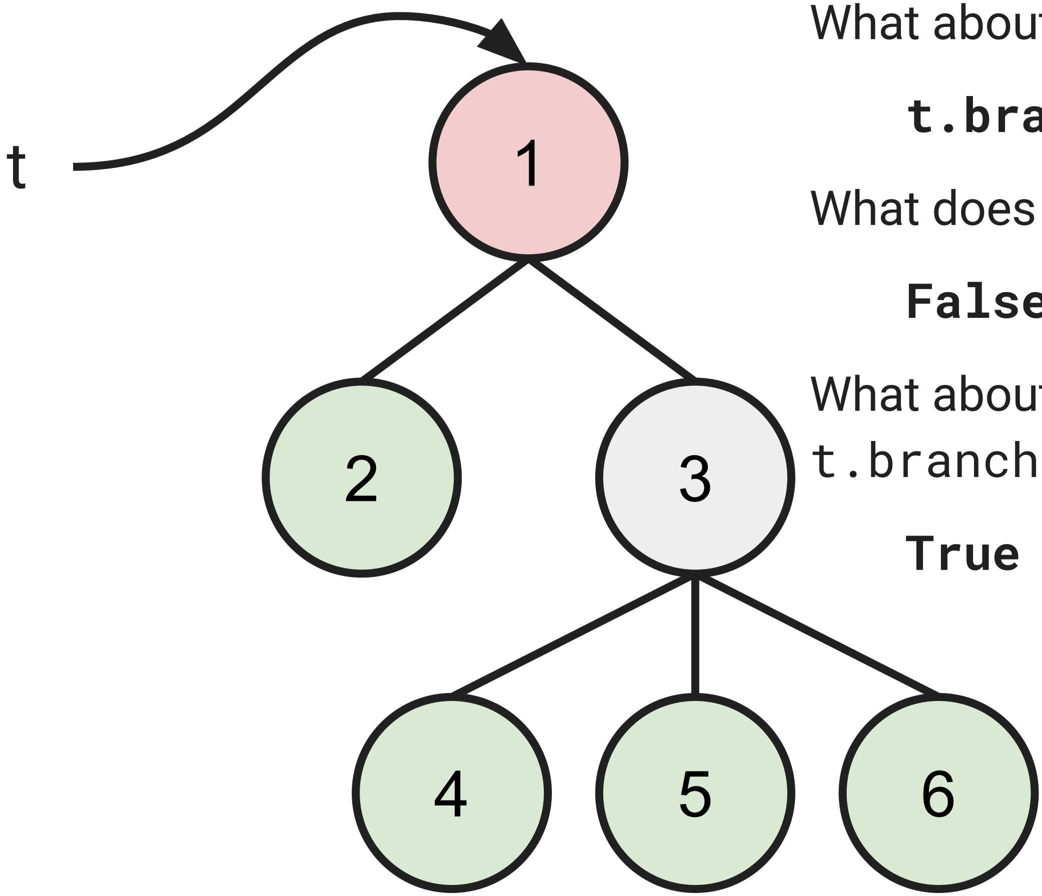


The `Tree` object whose label is 1 is the **root** of the tree stored at `t`

The `Tree` objects whose labels are 2 and 3 are the **branches** of that tree

The `Tree` objects whose labels are 2, 4, 5, and 6 are all **leaves**

# Sanity check



What is an expression that would evaluate to the Tree whose label is 2?

**`t.branches[0]`**

What about the tree whose label is 6?

**`t.branches[1].branches[2]`**

What does `t.is_leaf()` return?

**`False`**

What about `t.branches[1].branches[2].is_leaf()`?

**`True`**

# Tree Processing

## Tree processing

Trees are a natural recursive data structure!

Each `Tree` has:

- A `label`
- `0` or more `branches`, which are themselves `Trees`

This means that pretty much all of our algorithms to process trees are going to be recursive

(Especially, as the name suggests, tree-recursive)

Trees are simple to describe, but tricky to work with, so we're going to spend most of the rest of this lecture talking about different kinds of tree problems

# Count leaves

```
def count_leaves(t):
    """Returns the number of leaf nodes in T.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> count_leaves(t)
    2
    """
    if _____:
        __________
    else:
        __________
        # you can use as many lines as you want
```

# Count leaves - solution

```
def count_leaves(t):
    """Returns the number of leaf nodes in T.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> count_leaves(t)
    2
    """
    if t.is_leaf():
        return 1
    else:
        num_leaves = 0
        for b in t.branches:
            num_leaves += count_leaves(b)
        return num_leaves
```

# Count leaves - solution (one-liner recursive case)

```python
def count_leaves(t):
    """Returns the number of leaf nodes in T.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> count_leaves(t)
    2
    """
    if t.is_leaf():
        return 1
    else:
        return sum([count_leaves(b) for b
            in t.branches])
```

## List leaves

```
def list_leaves(t):
    """Returns a list of the leaf labels of T.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> list_of_leaves(t)
    [2, 3]
    """
    if _____:
        __________
    else:
        __________
```

# List leaves - solution

```
def list_leaves(t):
    """Returns a list of the leaf labels of T.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> list_of_leaves(t)
    [2, 3]
    """
    if t.is_leaf():
        return [t.label]
    else:
        list_of_lists = [list_leaves(b) for b
            in t.branches]
        return sum(list_of_lists, start=[])
```

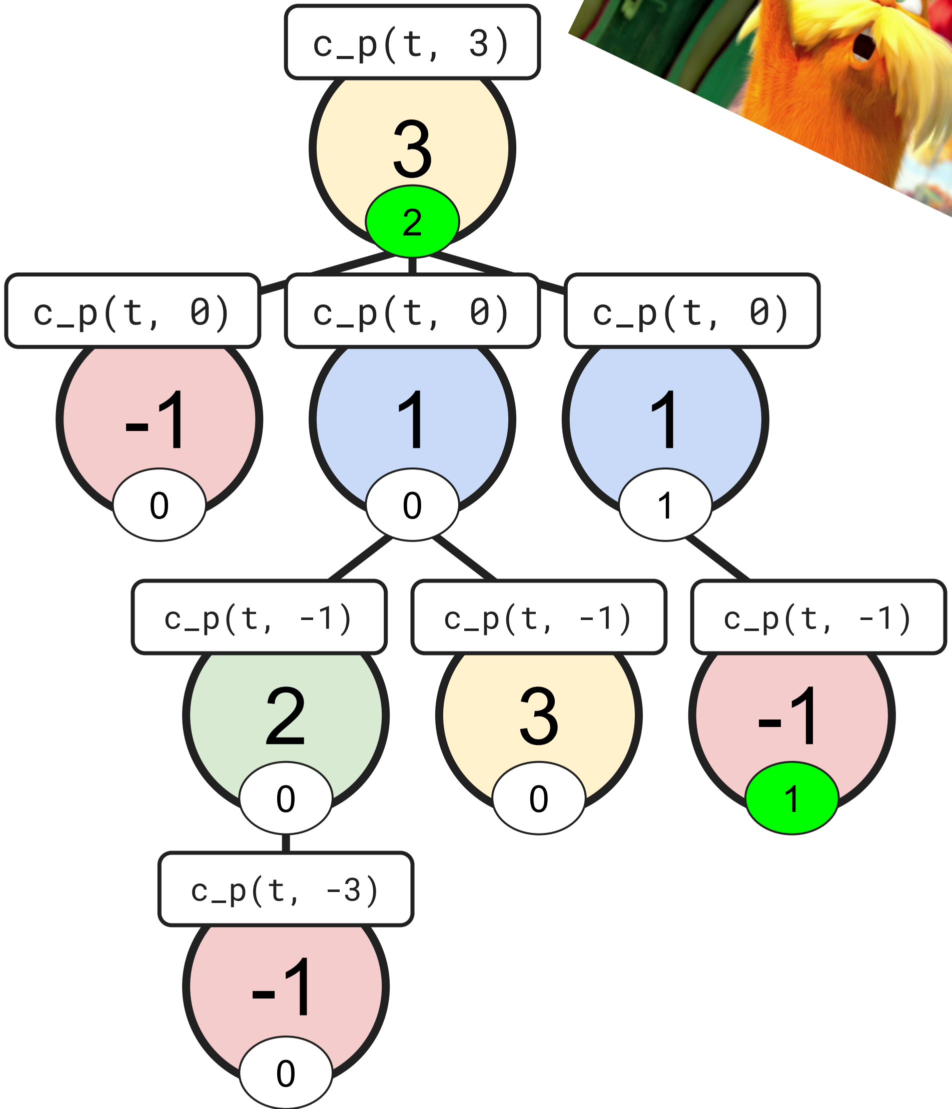
# Count paths

```
def count_paths(t, total):
    """Return the number of paths from the root to
    any node in T for which the labels along the
    path sum to TOTAL.
    >>> t = Tree(3, [Tree(-1), Tree(1, [Tree(2,
    [Tree(1)]), Tree(3)]), Tree(1, [Tree(-1)])])
    >>> count_paths(t, 3)
    2
    >>> count_paths(t, 4)
    2
    >>> count_paths(t, 5)
    0
    """
```

# Count paths - solution

```python
def count_paths(t, total):
    """Return the number of paths from the root to
    any node in T for which the labels along the
    path sum to TOTAL.
    """
    count = 0
    if total == t.label:
        count += 1
    for b in t.branches:
        count += count_paths(b, total - t.label)
    return count
```

# Visualizing count paths



c_p(t, 3)

3

2

c_p(t, 0)

-1

0

c_p(t, 0)

1

0

c_p(t, 0)

1

1

c_p(t, -1)

2

0

c_p(t, -1)

3

0

c_p(t, -1)

-1

1

c_p(t, -3)

-1

0

# Count paths - solution (with sum)

```
def count_paths(t, total):
    """Return the number of paths from the root to
    any node in T for which the labels along the
    path sum to TOTAL.
    """
    found = int(t.label == total)
    return sum([count_paths(b, total - t.label)
        for b in t.branches]) + found
```

# Aside: Why use list comprehensions? Or `sum`?

When programming, we have a couple main goals:

- Write code that solves the task
- Write code that is comprehensible to folks other than you
- Write code that runs as efficiently as possible

The first bullet point is obviously the most important, and we generally don't emphasize the other two much in this class. In general, which of the second or third bullet points matters most depends on the application and the context

Built-in Python constructs such as list comprehensions and functions like `sum` have some optimization strategies baked into them from the language itself (these are covered more in 61B and 61C)—this means we get an efficiency boost each time we're able to use them over a standard for- or while-loop

# Break

stival
16
#sundance
NDANCE TV
Festi
2016
sundanc
CANADA
#sund

# String Representation

# Print tree

```
def print_tree(t):
    """Prints the nodes of T with depth-based
    indent.
    >>> t = Tree(1, [Tree(2, [Tree(4), Tree(5)]),
    Tree(3)])
    >>> print_tree(t) # indents are two spaces
    1
      2
        4
        5
      3
    """
    # use as many lines as you want!
```

# Print tree

```
def print_tree(t):
    def helper(t, indent):
        print(indent * " " + str(t.label))
        for b in t.branches:
            helper(b, indent + 2)
    helper(t, 0)
```

Interesting things:

- We needed a helper function to carry information across calls—this is a common predicament
- There's no (explicit) base case! If `t` is a leaf, `t.branches` is empty, and the for-loop never executes. Our recursive calls are only inside the for-loop, so we never recurse on a leaf

# String representation for Trees

```
class Tree:
    # stuff we've already seen

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines
```

# Creating Trees

# Double

```
def double(t):
    """Returns a Tree identical to T, but with all
    labels doubled
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> double(t)
    Tree(2, [Tree(4), Tree(6)])
    """
    if _____:
        __________
    else:
        __________
```

# Double - solution

```
def double(t):
    """Returns a Tree identical to T, but with all
    labels doubled
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> double(t)
    Tree(2, [Tree(4), Tree(6)])
    """
    if t.is_leaf():
        return Tree(t.label * 2)
    else:
        return Tree(t.label * 2,
            [double(b) for b in t.branches])
```

# Double - solution (shorter!)

```
def double(t):
    """Returns a Tree identical to T, but with all
    labels doubled
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> double(t)
    Tree(2, [Tree(4), Tree(6)])
    """
    return Tree(t.label * 2,
        [double(b) for b in t.branches])
```

# Mutating Trees

# Double, mutatively

```
def double_mut(t):
    """Mutates T such that every label is doubled.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> double_mut(t)
    >>> t
    Tree(2, [Tree(4), Tree(6)])
    """
    # your code here!
```

# Double, mutatively - solution

```
def double_mut(t):
    """Mutates T such that every label is doubled.
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(4), Tree(6)])
    """
    t.label *= 2
    for b in t.branches:
        double_mut(b)
```

# Prune

```
def prune_n(t, n):
    """Prunes T by removing all subtrees whose
    label is N.
    >>> t = Tree(3, [Tree(1, [Tree(0), Tree(1)]),
    Tree(2, [Tree(1), Tree(1, [Tree(0),
    Tree(1)])])])
    >>> prune_n(t, 1)
    >>> t
    Tree(3, [Tree(2)])
    """
    # your code here!
```

# Prune - solution

```
def prune_n(t, n):
    """Prunes T by removing all subtrees whose
    label is N.
    """
    t.branches = [b for b in t.branches if
        b.label != n]
    for b in t.branches:
        prune_n(b, n)
```

## Summary so far

There are a couple of major types of Tree problems:

- Processing the whole tree and returning some kind of information about it
- Constructing a new tree—either based on an old one, or based on some other specification
- Mutating the labels of an existing tree
- Mutating the branches of an existing tree
- Sometimes we need helper functions to carry information across recursive calls

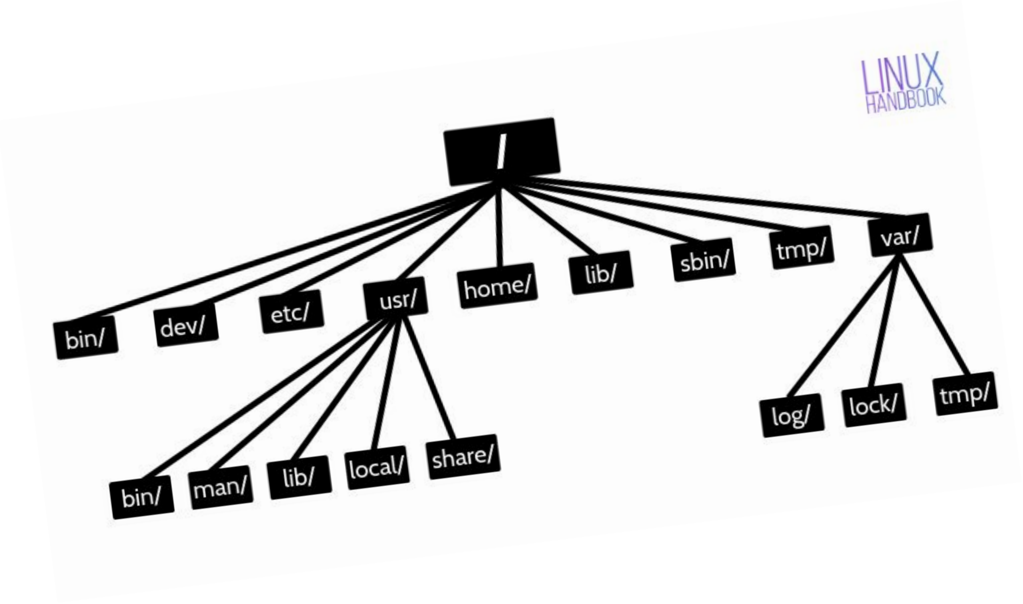
Tree algorithms use a lot of the same constructs over and over—`t.label`, `t.is_leaf()`, `for b in t.branches`, etc. It's harder than you might think to know when it's the right time to use each of those

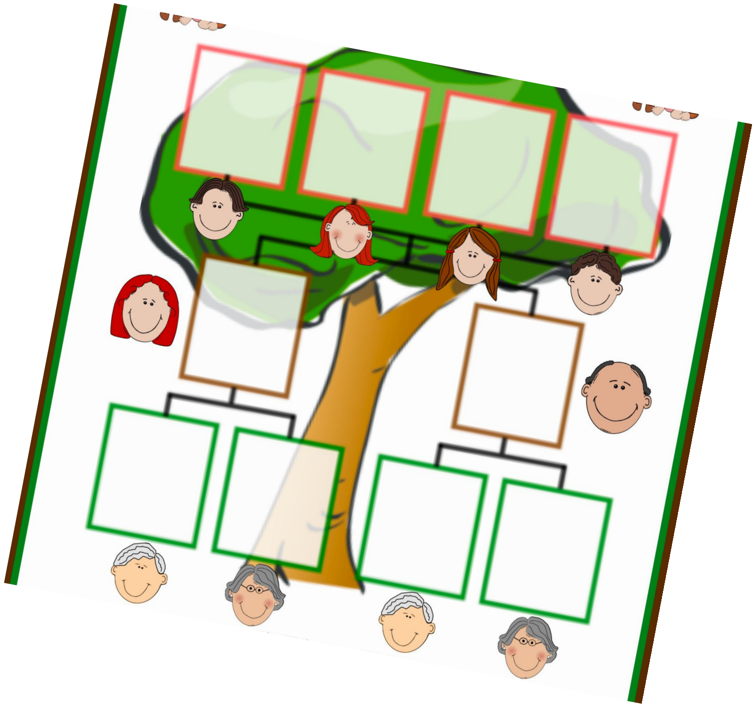
We just saw problems where we could use `is_leaf()`, but didn't need to!

# Syntax Trees

# Tree terminology

Trees are such a useful data structure because they can model things that are naturally recursive, and there's a ton of that both within and outside of CS!





## Syntax (the language kind)

Syntax is the branch of linguistics that studies sentence structure and word order—how these things are (un)constrained in various languages, and how they work together with words to create meaning

Consider this sentence:

The bar **was dirty**

The bar in Philadelphia **was dirty**

The bar in Philadelphia that Frank owned **was dirty**

In English, we can attach a (technically) arbitrary number of intervening clauses between a noun phrase and verb phrase, and speakers still agree what the verb phrase should be describing

It's really hard to describe this (and many other) phenomena using linear structure/direct word order—enter trees!

# Syntax trees

Syntax trees aim to model the structure of languages in a way that's easy to visualize (and, consequently, easy for computers to understand)

One of the first things we have to do to construct a syntax tree is classify words by their **part of speech**

| The | bar | in | Philadelphia | was | dirty |
|---|---|---|---|---|---|
| `DT` | `NN` | `IN` | `NNP` | `VBD` | `JJ` |

This is a kind of abstraction! There's a ton of words, but much fewer parts of speech, and many words behave in syntactically identical ways anyways (e.g. "The bar in Philadelphia was *run-down*")

Next, we want to use a tree to group these into **constituents**, which are basically phrasal units

| [The bar in Philadelphia] | [in Philadelphia was] |
|---|---|
| **`Constituent`** | **`Not a constituent`** |

# NLTK

The Natural Language Toolkit (NLTK - https://www.nltk.org/) is a pre-existing Python library with a ton of tools and databases for computational linguistics and natural language processing, including syntax trees!

NLTK comes with a built-in part-of-speech tagger that we can play around with! It's not perfect because even that task is relatively challenging

NLTK can also generate syntax trees (**parse trees**) from a sentence, when given a grammar—again, it's a very imperfect tool, but super neat!

Demo