Computational Structures in Data Science

Data Structures:
Linked Lists

Guest Lecture (Rebecca Dang)

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

A note on the midterm…

- Staff aiming to release midterm grades by end of day tomorrow (Tue 3/19) but no guarantees
- Your grades don't define you. Seriously!
  - From Berkeleytime historical data, ~40% of students get an A- or higher
- Focus on learning the material, stay on track, don't lose sight of the big picture
- If you need support or have questions/concerns please reach out :D

Announcements

Lab 8 and HW 8

- Released this week
- Due the week we come back from spring break
- Nothing due this week 🎉

Fun Video: CGP Grey Rock Paper Scissors

- How many rounds of Rock Paper Scissors is a 1 in 1,000,000,000 chance of winning?
- Each video leads to another set of videos.
- This is technically a *tree*, but we'll come back to that later.

Where We're Going

- For now – we've learned *most* of the basics of Python!
  - There are plenty of Python we don't see in CS88
- We'll be applying OOP principles to explore new topics.
- We're going to focus on storing / organizing data
  - Lists, Tuples, and Dictionaries: Data Structures you already know!
- BUT: How do we build our own?
  - We'll build our own lists first, then talk about trees and other ways of organizing data
- Last few lectures: Switch to SQL

Why "Data Structures"? (Next Few lectures)

- Data Structures
  - OOP helps us organize our *programs*
  - Data Structures help us organize our data!
    - Can be implemented using OOP
  - You already know lists and dictionaries!
  - We'll see a new one today
- Enjoy this stuff? Take CS 61B!
- Find it challenging? Don't worry! It's a different way of thinking.

Computational Structures in Data Science

Linked Lists

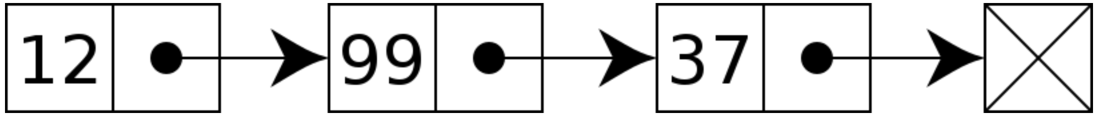
Berkeley
UNIVERSITY OF CALIFORNIA

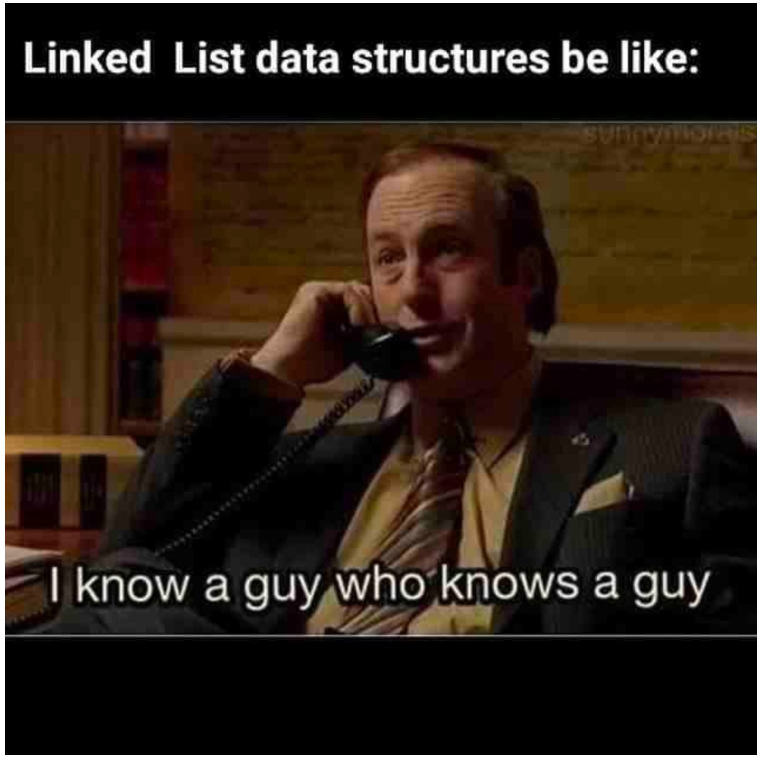
Data Structures

- A data structure is a way to organize or group a bunch of independent pieces of data.
  - Lists (arrays)
  - Dictionaries
  - Tuples
- A class, on its own, is ***not*** necessarily a data structure, it represents a new data type.
  - a "car" or a "person" is an instance of that data type.
  - Lists, Dicts, etc are also data types; their goal is to organize other data.
- These are common patterns that can be used to solve a wide variety of problems.
- Sometimes we're giving structure to make it easier as a programmer, sometimes we're trying to be fast or efficient.

Linked Lists

- A Recursive List, sometimes called a "rlist"
- Linked lists contain other linked lists
- A series of items with two pieces:
  - A value, usually called `"first"`
  - A “pointer” to the `rest` of the items in the list.





- We’ll use a very small Python class “Link” to model this.
- `Link(12, Link(99, Link(37, Link.empty)))`

What's Needed For a Linked List?

- `first`
- `rest`
- An idea of "empty"
- Nothing else is *necessary*
- `__repr__`, `__len__` methods are all useful shortcuts and useful recursion practice.

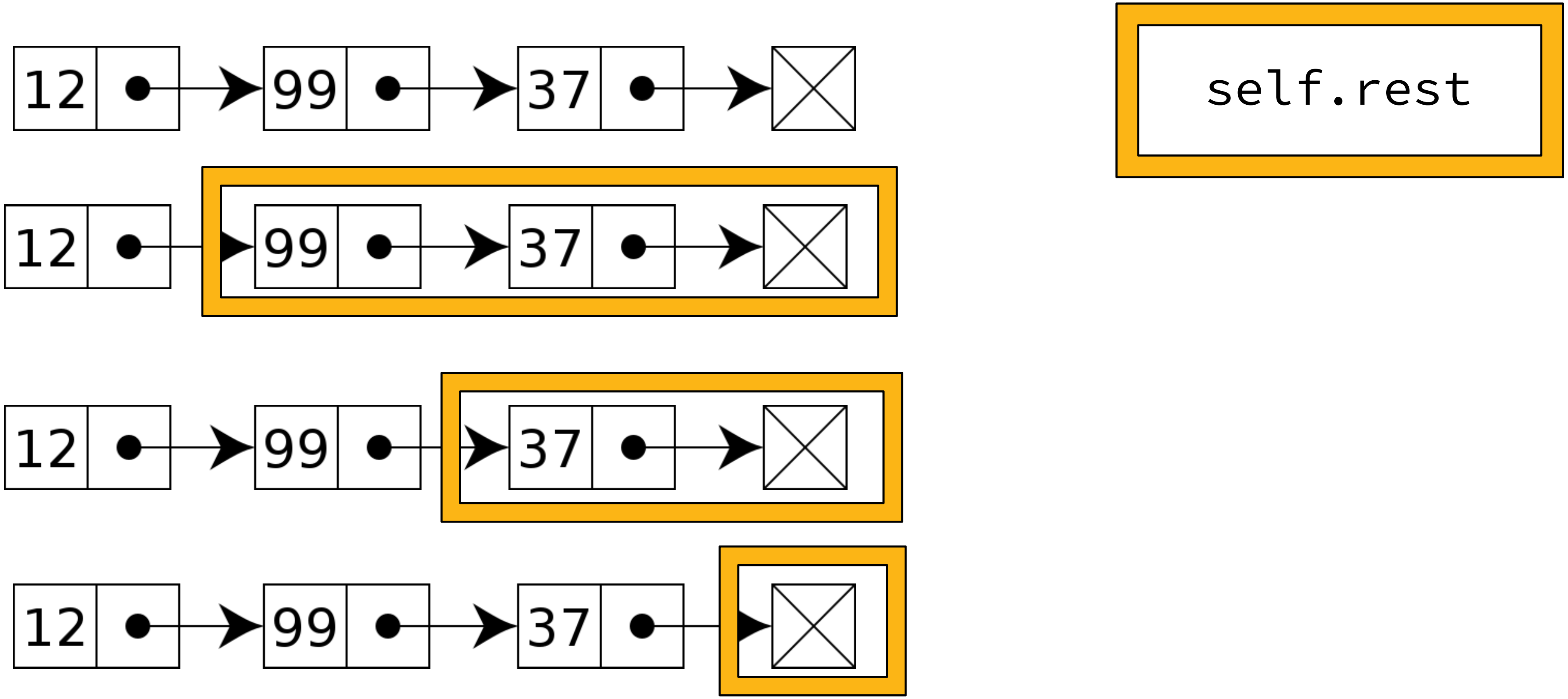
The Link Class

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

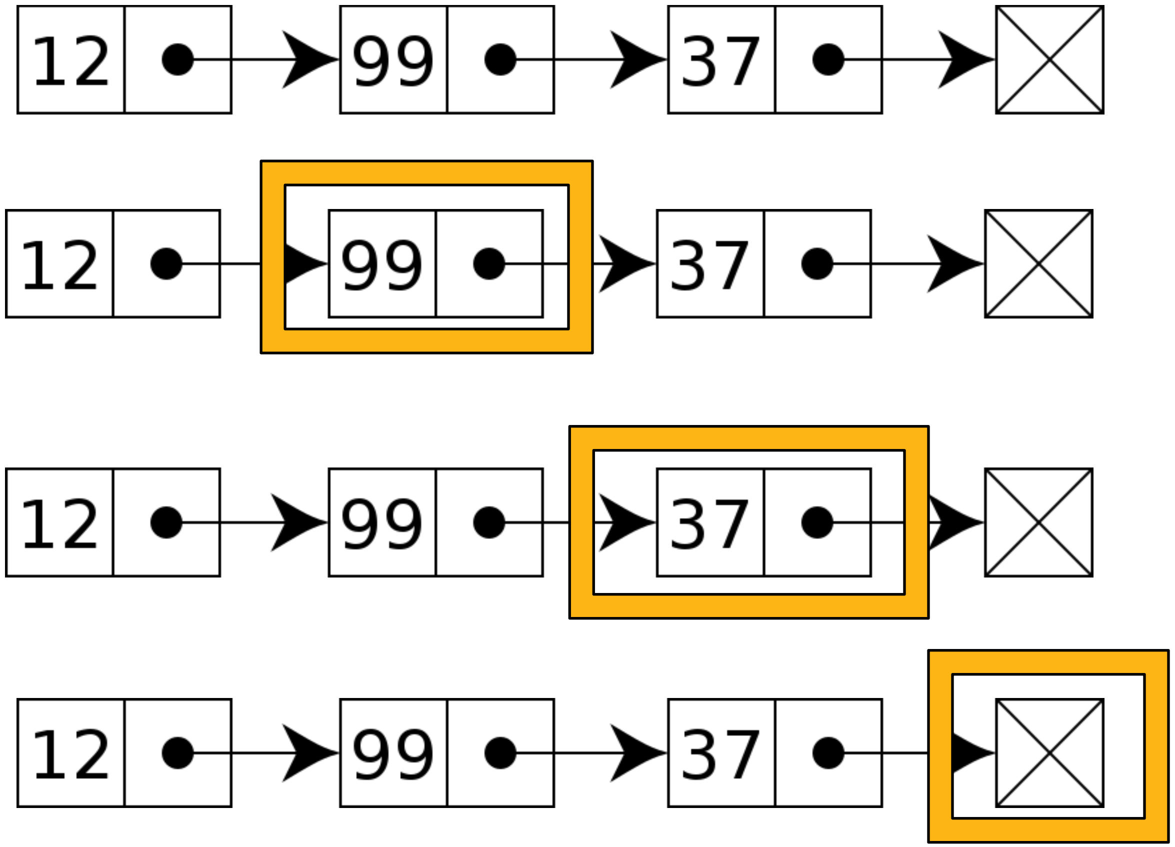
That's all we need!

- We can add a __repr__ method, length, etc.
- Use an empty tuple for clarity / easier than None.
  - () has lots of useful methods defined, like len()

Recursion Is Implicit



Different ways to think of a linked list: “Relative” vs “recursive”



`self.rest`

Iterating or Processing a Linked List

- Our base case or stopping condition?
  - Linked List is empty!
- We can use recursion or iteration.
  - Which is “better”?
  - Depends on the problem we are trying to solve!

Iterating Over All Items in Linked List

```
def print_link(link):
    if not link:
        return
    print(link.first)
    print_link(link.rest)
```

- Base Case: No more items
- Do Action
- Recurse on the rest of the list

Iterating Over All Items in Linked List

```
def print_link(link):
    if not link:
        return
    print(link.first)
    print_link(link.rest)
```

- Base Case: No more items
- Do Action
- Recurse on the rest of the list

```
def print_link(link):
    if not link:
        return
    item = link
    while item:
        print(item.first)
        item = item.rest
```

- Handle the empty list
- Keep track of current item
- Update item to be the next in sequence.

Demo – See the Notebook

Uses for a Linked List

- Modeling a Polynomial Equation
  - each item is (coefficient, exponent, next_term)
- Items in a music `Playlist`
  - each item is a (song, next_song) pair
  - easy to add/remove items
    - Specifically: often want to remove the first item
- Model real-world relationships
  - Anything that is a "chain" is a good option
  - Next up: We'll extend this idea to "trees"

Why are linked lists useful?

- **Honestly, a regular list is easier *most* of the time**
  - Python handles all the hard details!
  - When data gets large, there are lots of edge cases.
- In terms of ***efficiency***: Linked lists make it fast to move items around, insert, and delete **from the front and/or back** (depending on implementation)
  - But they are slower to finding any single item ("random access") – **can't index into a linked list**
- In **Ants** Project: You'll see a list of `Place` objects which are linked together via an entrance and an exit – they're linked lists!

Lists vs Linked Lists

Lists

- Built into Python
- Create with `[]` or `list()`
- Can iterate through with loops
- Can use index to retrieve element (e.g. `lst[0]`)
- Not a recursive data structure

Linked Lists

- `Link` class (created for C88C, isn't built into Python)
- Create with `Link(<first>, <rest>)`
- Can iterate through with loops
  - But not "directly" through a for loop
- Can't use indices to retrieve elements
- Is a recursive data structure

Please send me your feedback (anonymous) :D

https://go.c88c.org/rebecca-lecture

Efficiency of Linked Lists vs Lists

- Linked Lists generally use less memory.
- Linked Lists:
  - Once you've found an item, inserting / removing is easy, O(1)
  - Finding anything other than the first/last item is O(n)
- "Regular" Lists:
  - Inserting / Removing items, other than the last is O(n) – due to internal copying
  - Finding any random item is O(1).

- What if you need to iterate over all items in order?
  - O(n) in both cases