M2: Data Structures

DSE 10200: Introduction to Data Science

Instructor: Michael Grossberg

Where should data live?

The only answer to all Technical Questions

The only answer to all Technical Questions

It depends …

What does it depend on?

What are you going to do with the data?

- Different things have different requirements!
- Think about organizing books in a library.

How should we arrange the books?

- If we know the author ….
- We can arrange by author
- But if we only know the title …
- We can arrange by title

How should we arrange the books?

- If we only know the title ….
- But the books are arranged by author first
- Veerrry slow to search

What do we usually want to do with data?

What do we usually want to do with data?

- Read/access in sequence (e.g. time series)
- Read/access in random order (search)
- Write in sequence (e.g. a log)
- Update random parts of the data (e.g. a database of records)
- Sample the data randomly
- Get the next most important item with changing importance

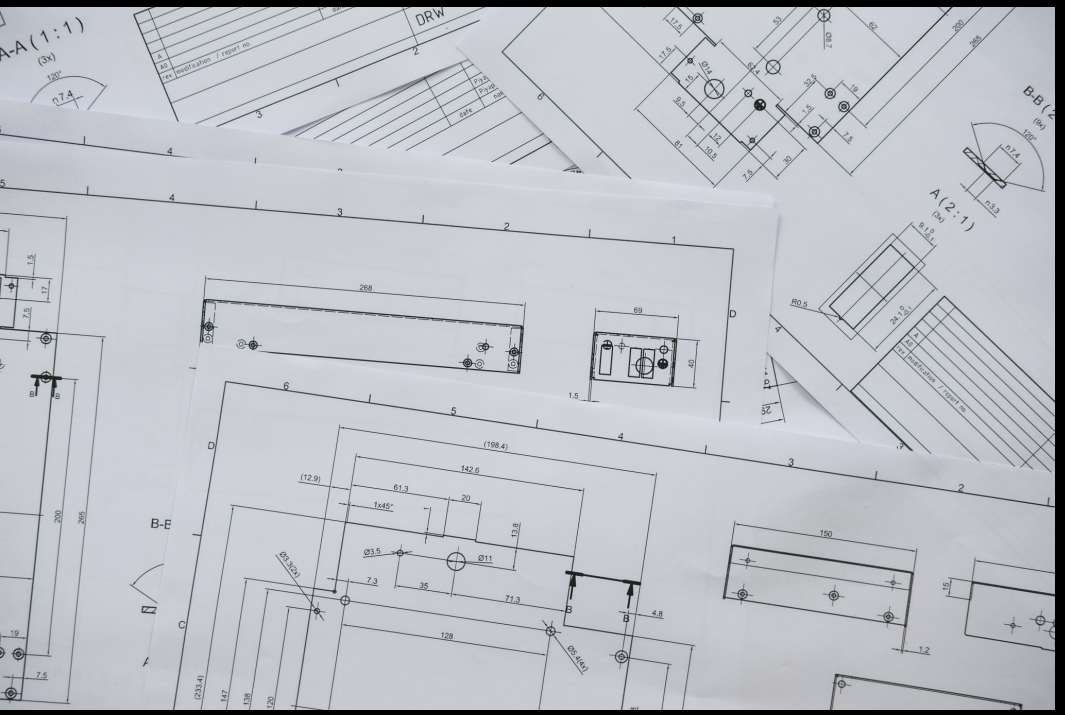
- Etc. Etc. Etc.

Data Structures

- Each data structure is good at some things and less good at others
- Or kind of ok at everything but not awesome at anything
- Some take up lots of space
- Some are compact
- Some can be distributed across different locations for safety or speed
- Some are very reliable
- Some are very fast (but less reliable)

Know your bag of tools and when to use each

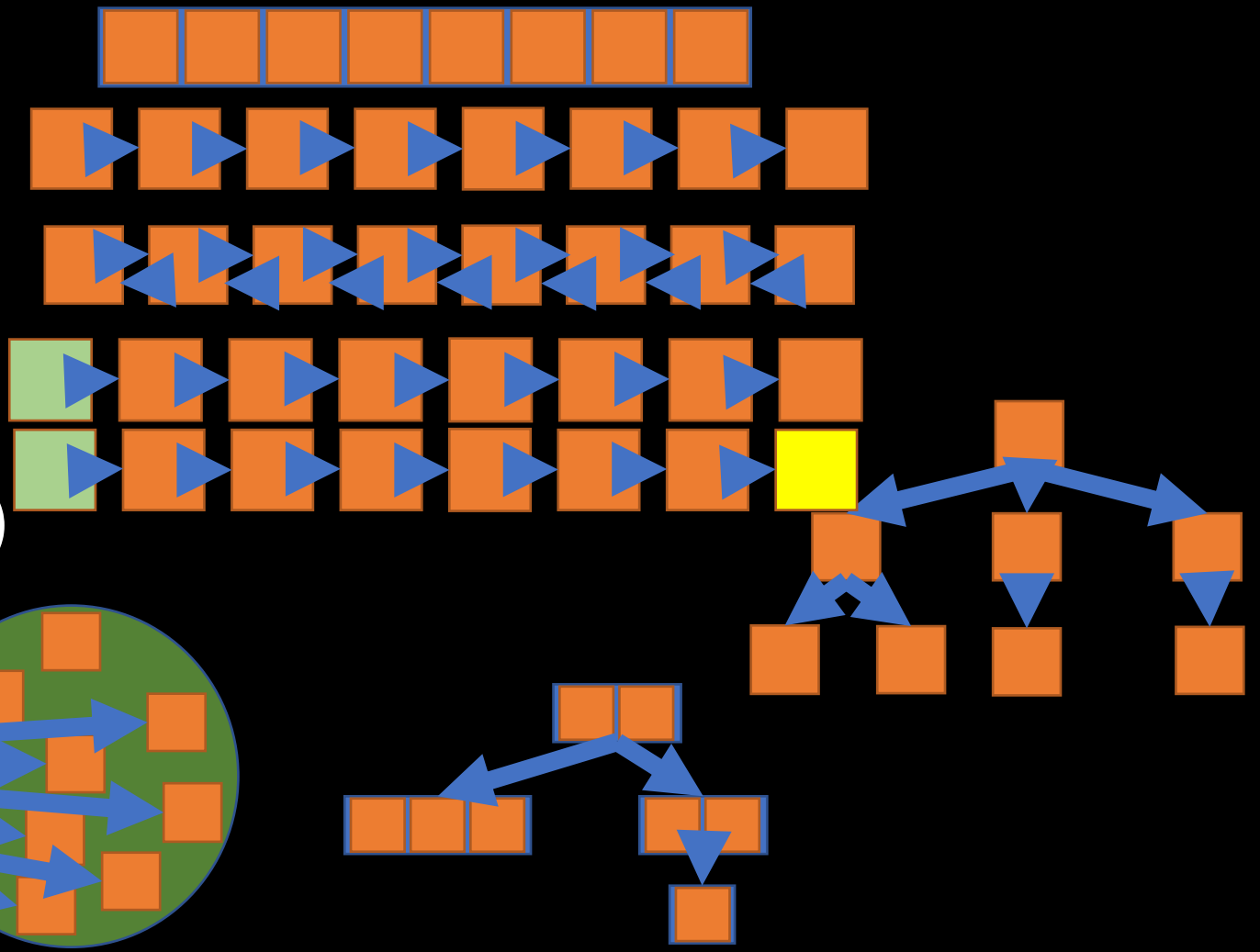
Abstract Data Structure vs Implementation



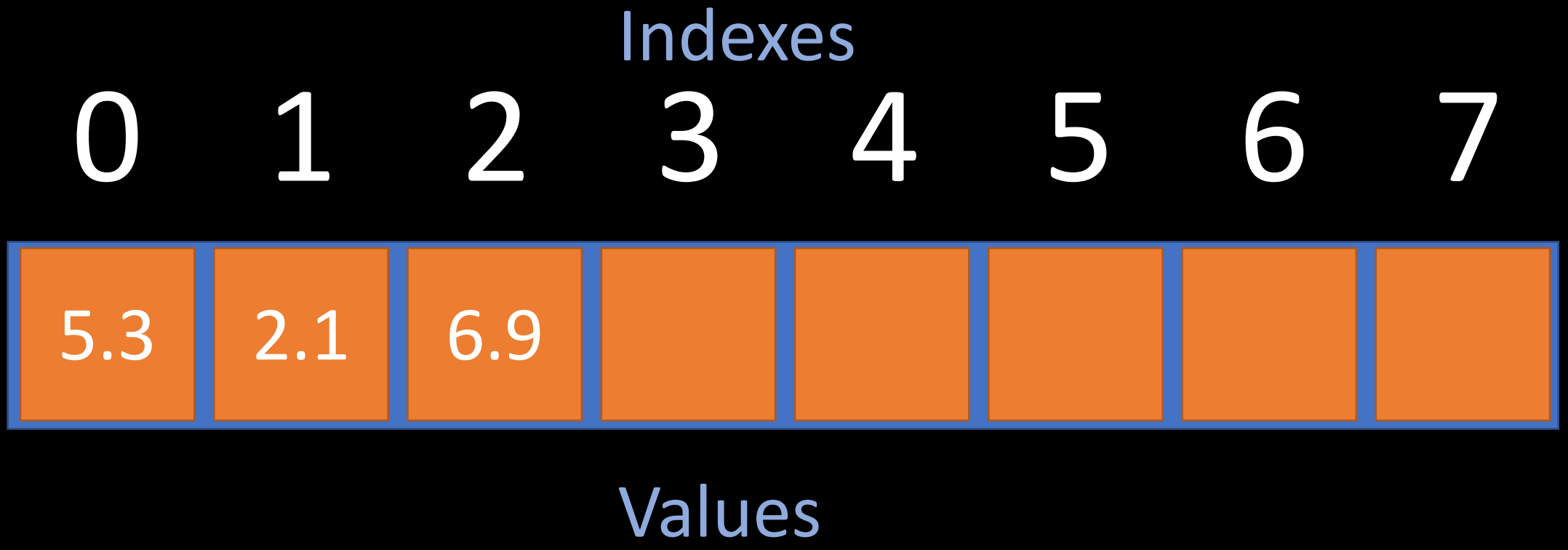
Vs

Some Important Types of Data Structures

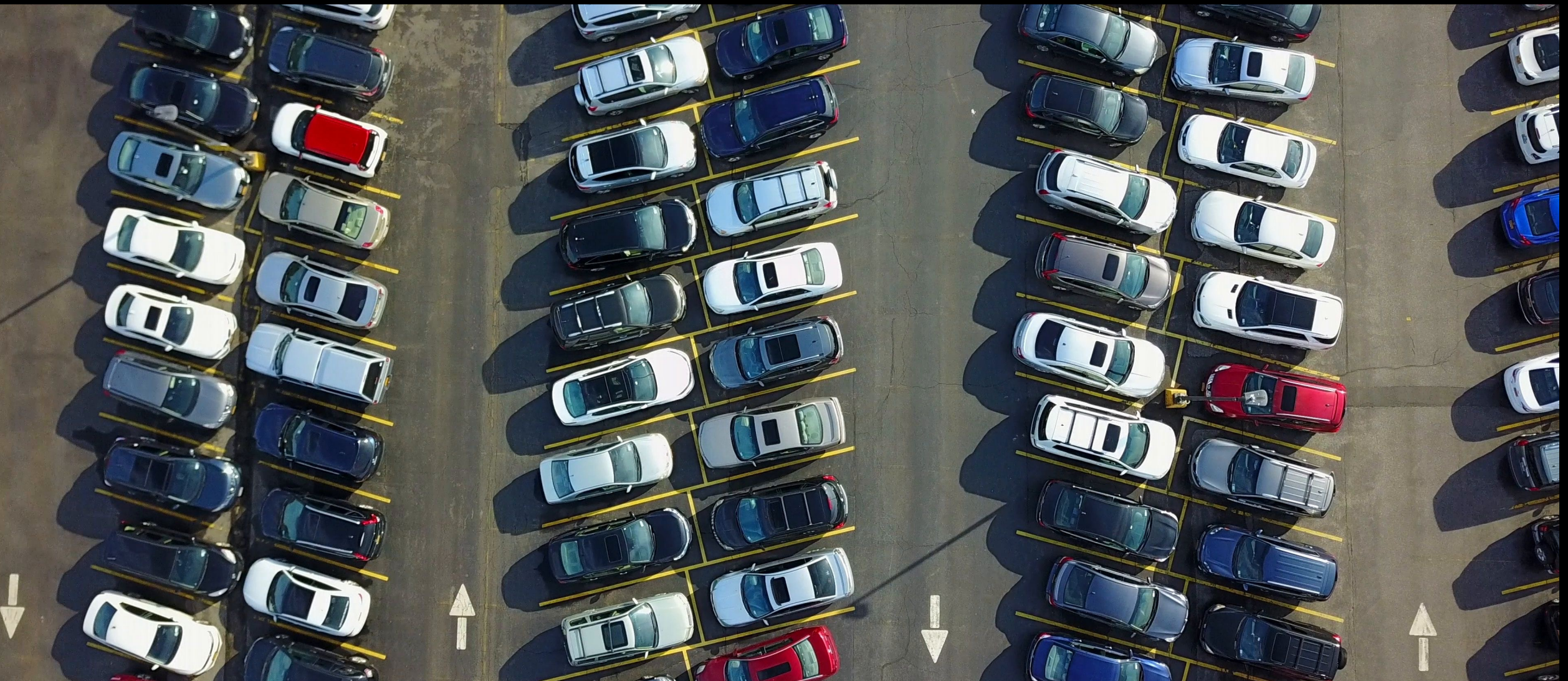
- Array
- Link List
- Double Link List
- Stack and Queue
- Key-Value Store (hash table)
- Tree
- Composite



Array



Array



Array

H	e	l	l	o	_	W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Question 1

- How can we expand the array so we can add more items at the end?

Question 2

- How can we delete the third element from the middle of the array?

Question 3

- How can we insert an element between the 4th and 5th element of the array?

Example Array

numpy.array¶

```
numpy.array(object, dtype=None, *, copy=True, order='K', subok=False,
ndmin=0, like=None)
```

Create an array.

Parameters:

object : *array_like*

An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype : *data-type, optional*

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

copy : *bool, optional*

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (**dtype**, *order*, etc.).

Pros Cons

- Advantages:
- Easy to access anywhere by index
- Easy to update by index
- Easy to loop though sequence

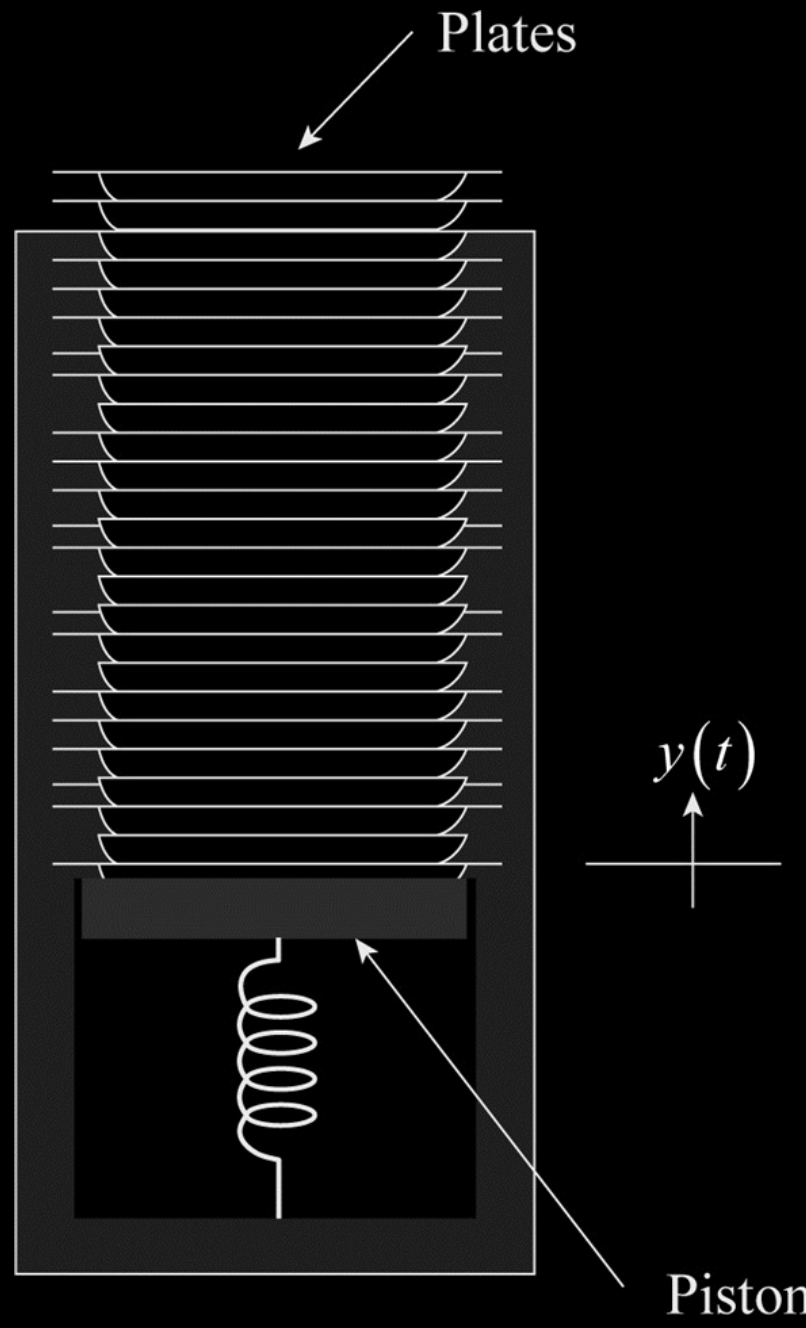
Disadvantages:

- Hard to delete items
- Hard to insert items

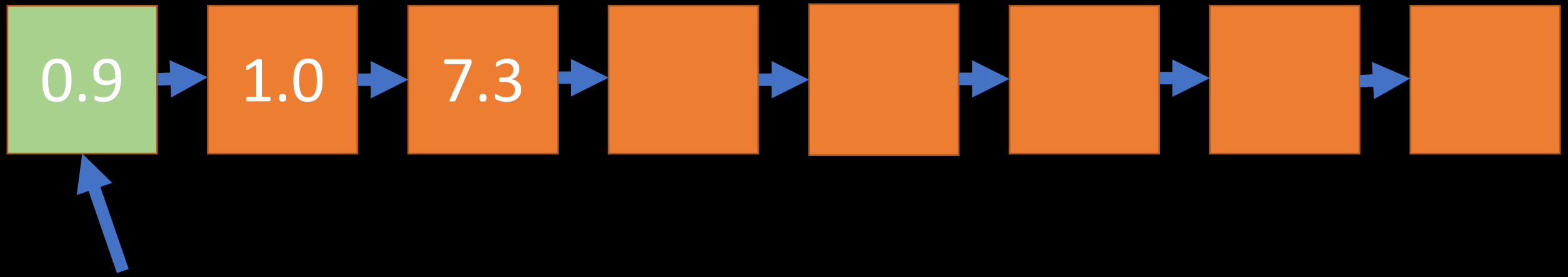
Stack (of plates)

Push on top

Pop off top



Stack (first-in last out = FILO)



Stack

5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Question 1

- How do we iterate though the stack using only push and pop non-destructively?

Question 2

- How do we add a function to a stack to index into it using only push and pop non-destructively?

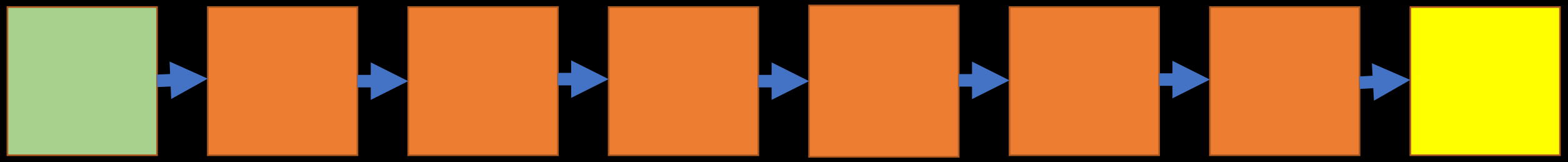
Pros Cons

- Advantages:
- Easy to add to top
- Easy to take from top
- Easy to access top
- Easy to go through sequence (destructively)
- Resizes

Disadvantages:

- Hard to delete items not at top
- Hard to insert items not at top
- Hard to access arbitrary items

Queue (first-in first-out = FIFO)



Queue

Queue

5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now lea
'Eric'
>>> queue.popleft()                 # The second to arrive now le
'John'
>>> queue                           # Remaining queue in order of
deque(['Michael', 'Terry', 'Graham'])
```

Question 1

- How can we implement a queue class similar to stack?

- Hint: If s is a list then 's.insert(i, item)' inserts 'item' at index 'i'

Pros Cons

- Advantages:
- Easy to add to top Easy to take from bottom
- Easy to access bottom
- Easy to go through sequence (destructively)
- Resizes automatically

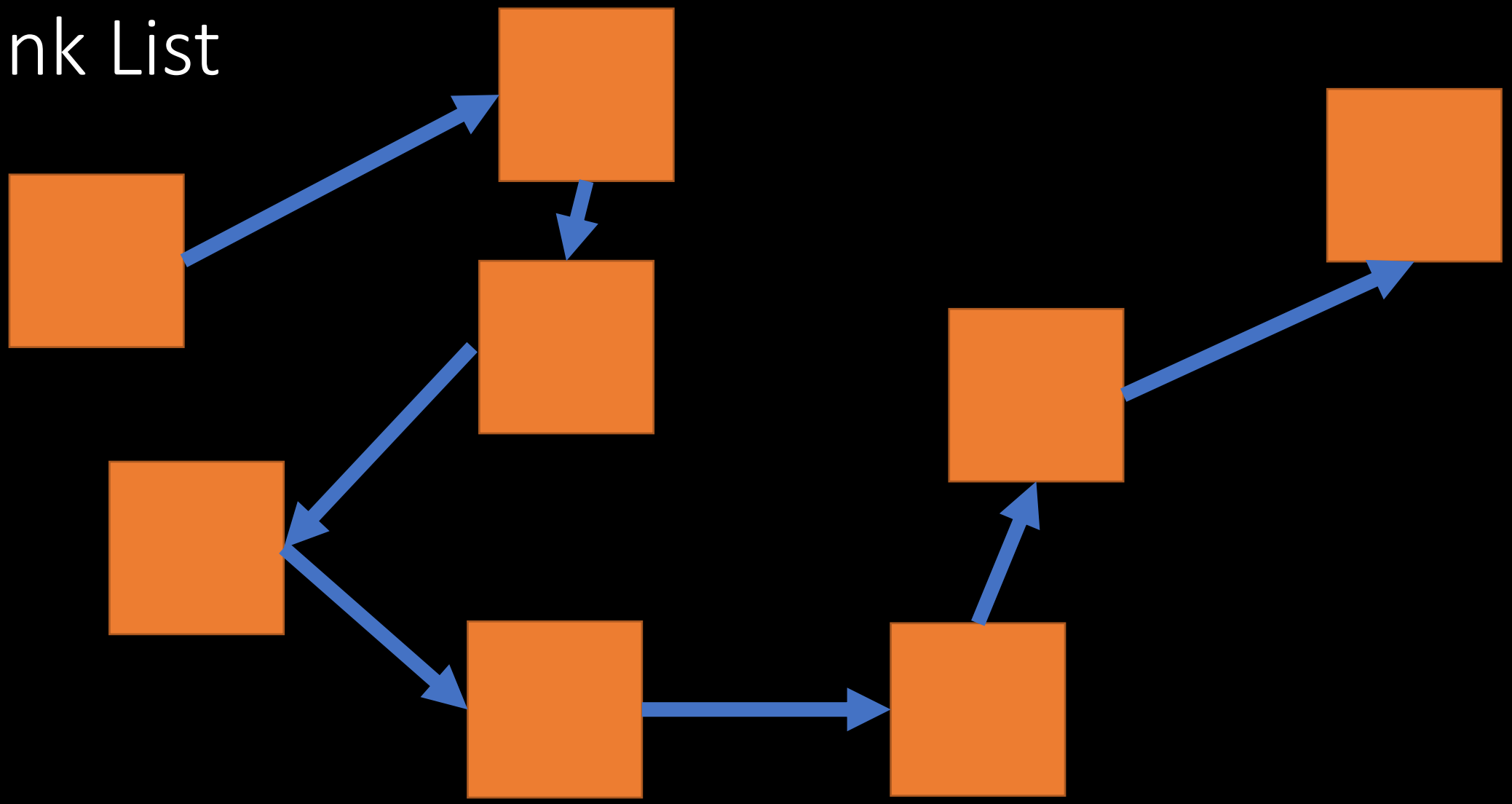
Disadvantages:

- Hard to delete items not at top
- Hard to insert items not at top
- Hard to access arbitrary items

Python List can work like stacks or queues

- s.append(item): adds to the end
- s.count(item): number of items in 's' that match 'item'
- s.index(item): returns the index of first match of item in 's'
- s.insert(ind, item): inserts the item at index 'ind'
- s.remove(ind): removes an item at index 'ind'
- s.reverse(): reverses all the items in place (and returns nothing)
- s.sort(): sorts the elements on a list

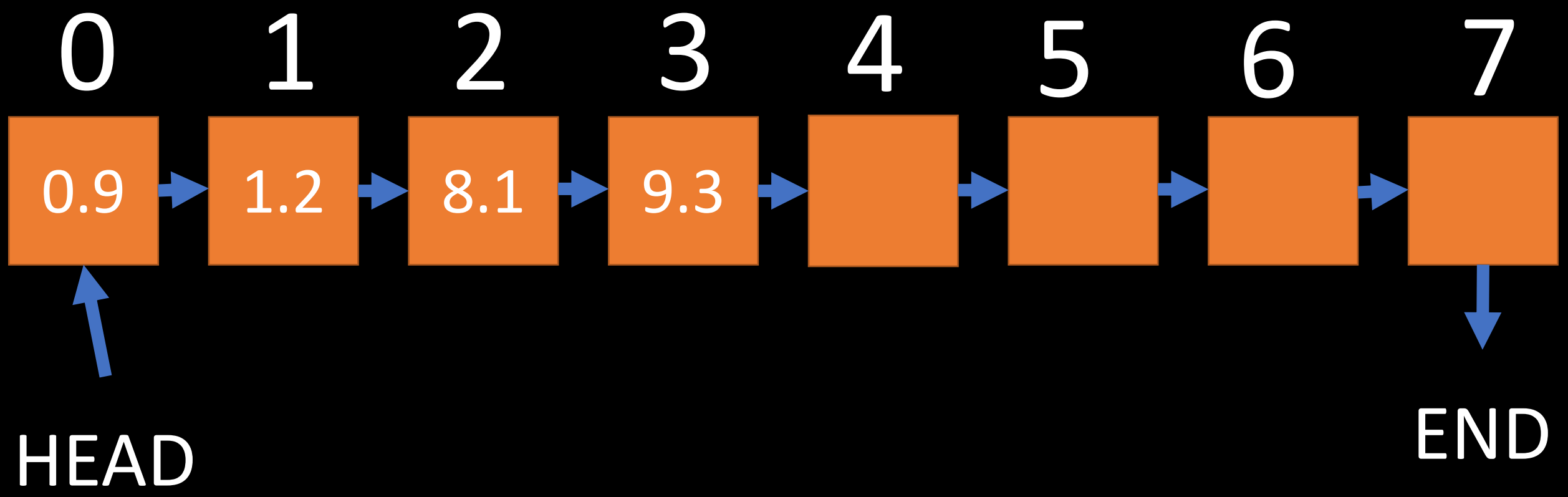
Link List



List of Lists teach us about references

- “id” is a unique indicator
- x = [[1,2], [1,2]]
- x[0][0] = 3
- print(x)
- y = [1, 2]
- z = [y, y]
- z[0][0] = 3

Link List



Linked List


Download from
Dreamstime.com
This watermarked comp image is for previewing purposes only.
ID 9320510
Olikli | Dreamstime.com

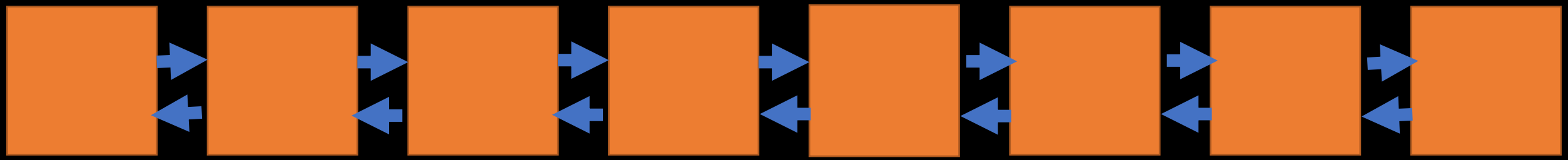
Pros Cons

- Advantages:
- Easy to add to top/bottom (expanding not a problem)
- Easy to take from top/bottom
- Easy to access top/top/bottom
- Easy to go through sequence (destructively)
- Resizes automatically

Disadvantages:

- Hard to delete items not at top
- Hard to insert items not at top
- Hard to access arbitrary items

Doubly Link List



Link List implementation

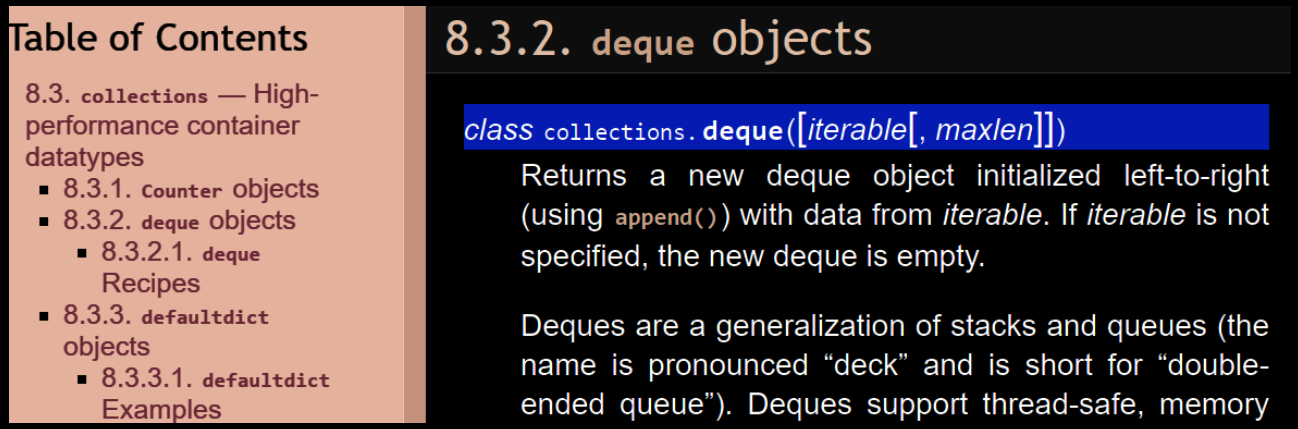


Table of Contents

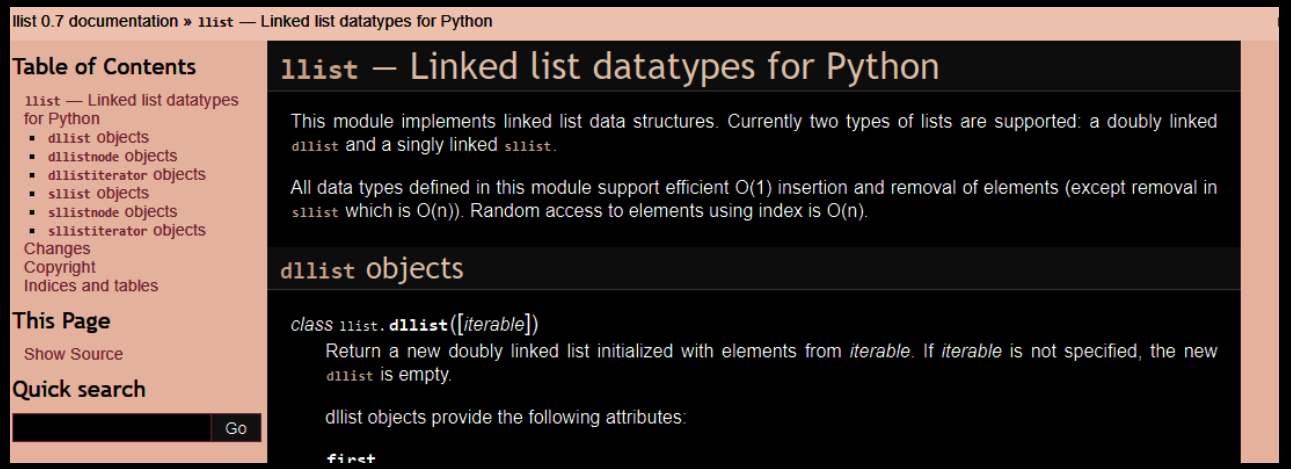
8.3. `collections` — High-performance container datatypes
- 8.3.1. `Counter` objects
- 8.3.2. `deque` objects
  - 8.3.2.1. `deque` Recipes
- 8.3.3. `defaultdict` objects
  - 8.3.3.1. `defaultdict` Examples

8.3.2. `deque` objects

class `collections.`**`deque`**([*iterable*[, *maxlen*]])

Returns a new deque object initialized left-to-right (using `append()`) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory



llist 0.7 documentation » `llist` — Linked list datatypes for Python

Table of Contents

`llist` — Linked list datatypes for Python
- `dllist` objects
- `dllistnode` objects
- `dllistiterator` objects
- `sllist` objects
- `sllistnode` objects
- `sllistiterator` objects

Changes
Copyright
Indices and tables

This Page

Show Source

Quick search

Go

`llist` — Linked list datatypes for Python

This module implements linked list data structures. Currently two types of lists are supported: a doubly linked `dllist` and a singly linked `sllist`.

All data types defined in this module support efficient O(1) insertion and removal of elements (except removal in `sllist` which is O(n)). Random access to elements using index is O(n).

`dllist` objects

class `llist.`**`dllist`**([*iterable*])

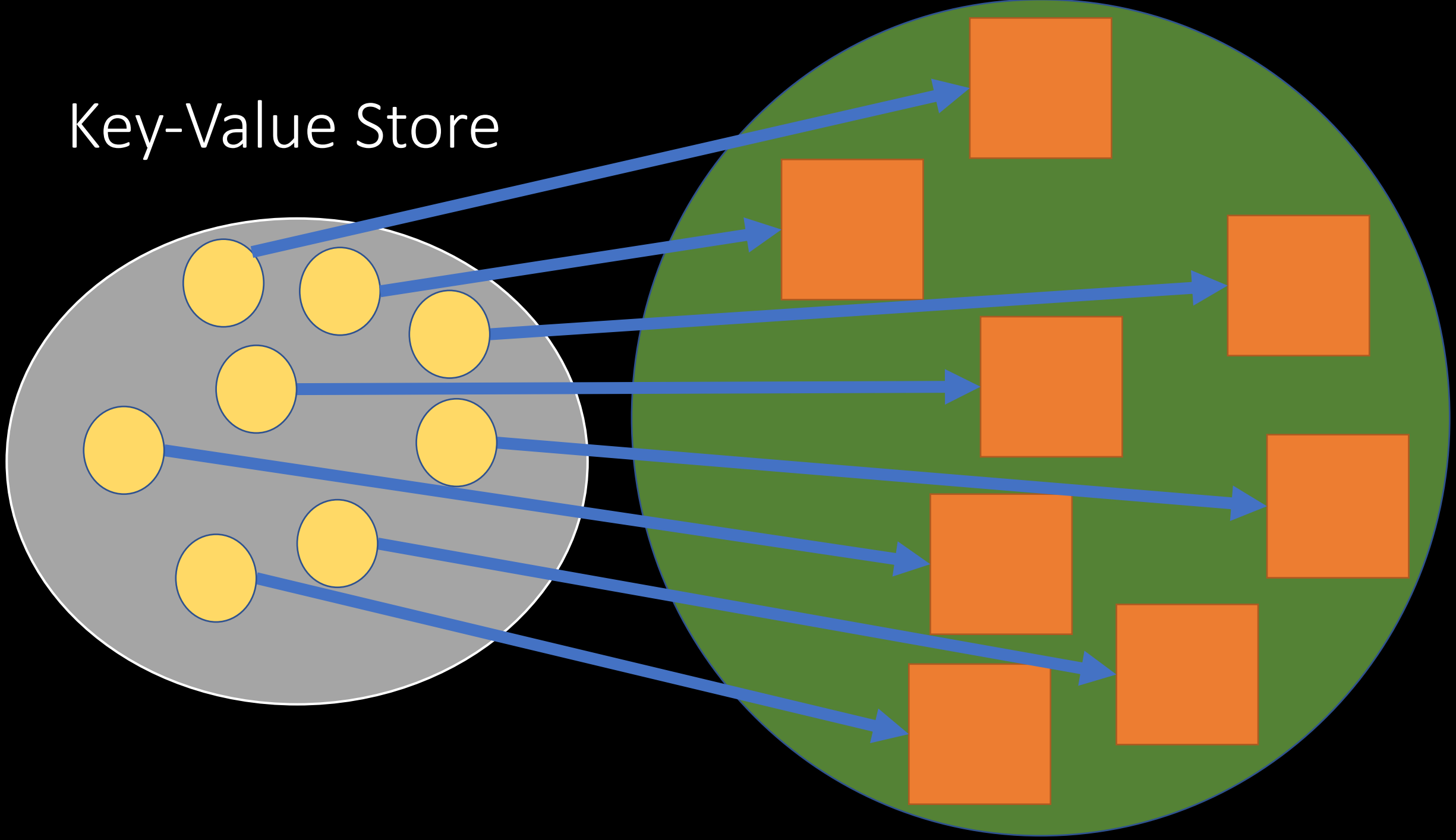
Return a new doubly linked list initialized with elements from *iterable*. If *iterable* is not specified, the new `dllist` is empty.

dllist objects provide the following attributes:

first

https://ajakubek.github.io/python-llist/index.html

Key-Value Store



Key value sort of like this

Key-Value Store

- Often implemented as a hash table
- Hash function takes key and gives almost a random index
- Data stored in a large array (lots of it is empty though)

Dict is a Key Value Store

class **dict**(***kwarg*)
class **dict**(*mapping*, ***kwarg*)
class **dict**(*iterable*, ***kwarg*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

Dictionaries can be created by several means:

- Use a comma-separated list of `key: value` pairs within braces: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- Use a dict comprehension: `{}`, `{x: x ** 2 for x in range(10)}`
- Use the type constructor: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

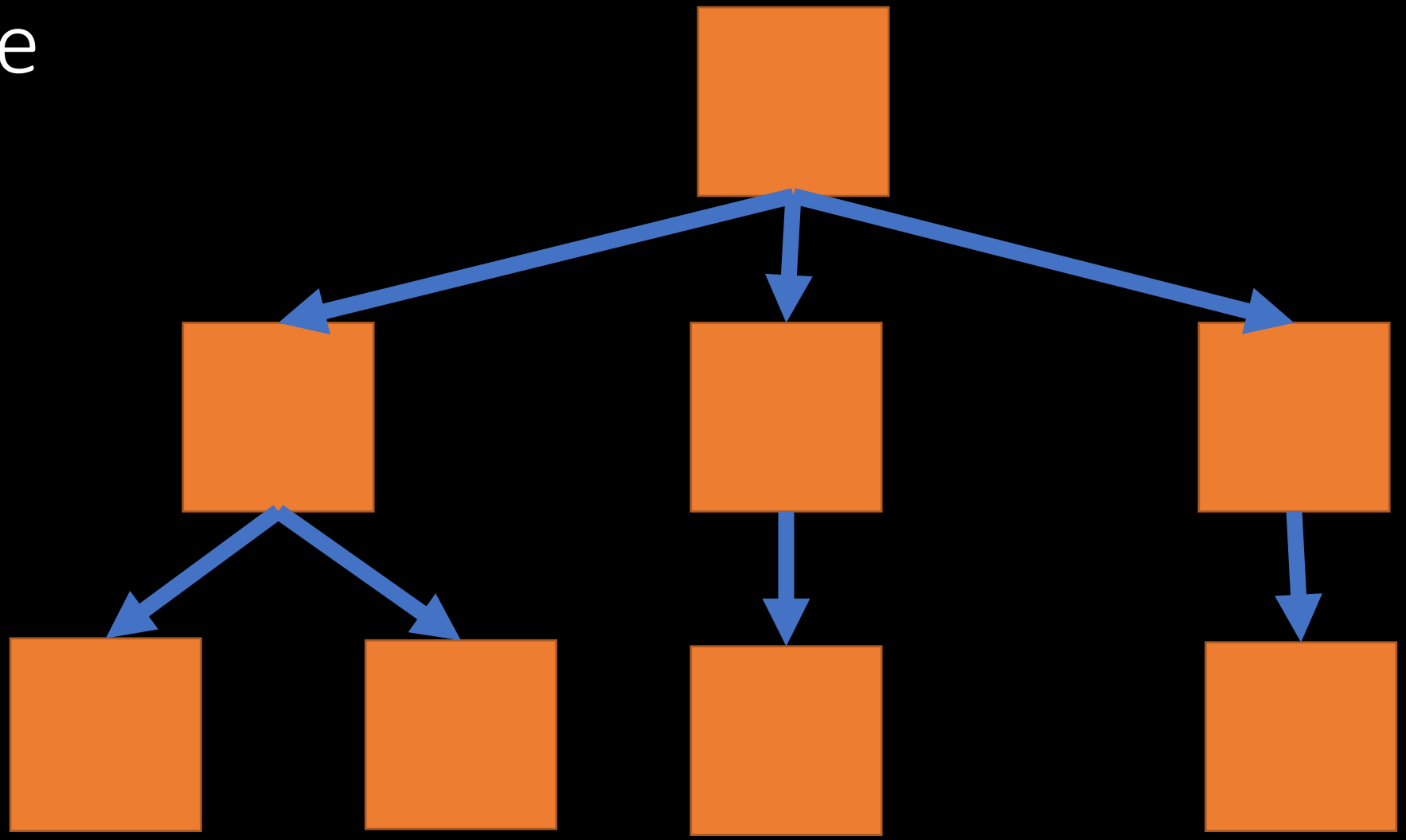
Pros Cons

- Advantages:
- Easy to add by key (resizes)
- Easy to remove by key
- Easy to access by key
- Easy to update by key
- Resizes automatically

Disadvantages:

- Hard to go through as a sequence

Tree



20 Questions … is a tree

treelib

Basic Usage

```
>>> from treelib import Node, Tree
>>> tree = Tree()
>>> tree.create_node("Harry", "harry")  # root node
>>> tree.create_node("Jane", "jane", parent="harry")
>>> tree.create_node("Bill", "bill", parent="harry")
>>> tree.create_node("Diane", "diane", parent="jane")
>>> tree.create_node("Mary", "mary", parent="diane")
>>> tree.create_node("Mark", "mark", parent="jane")
>>> tree.show()
Harry
├── Bill
└── Jane
    ├── Diane
    │   └── Mary
    └── Mark
```

https://treelib.readthedocs.io/en/latest/

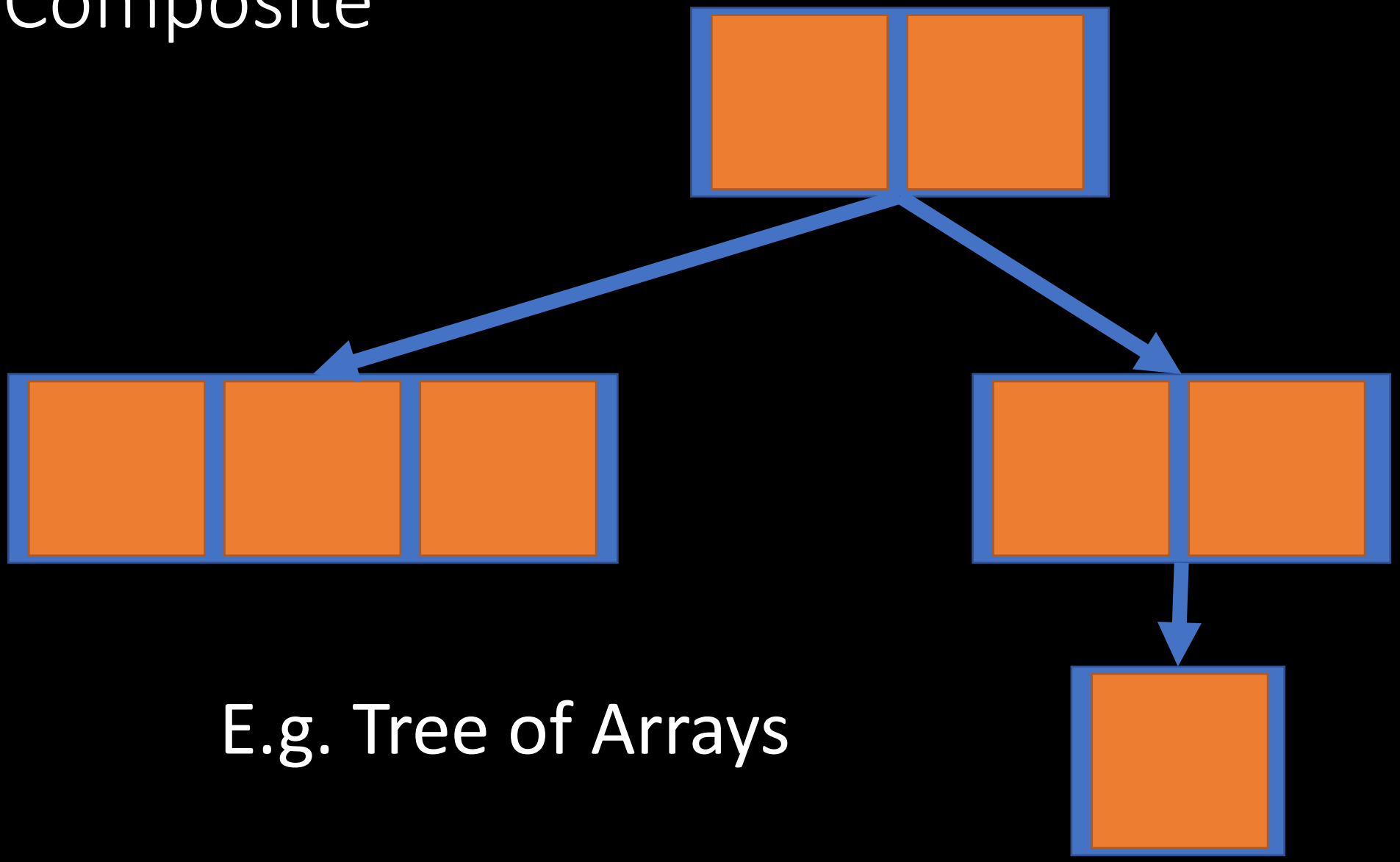
Pros Cons

- Advantages:
- Moderately Easy Resizing
- Moderately easy add/update/delete
- Moderately easy random access
- Moderately easy to go though as a sequence

Disadvantages:

- Complex to implement
- Kind of just ok and most things

Composite



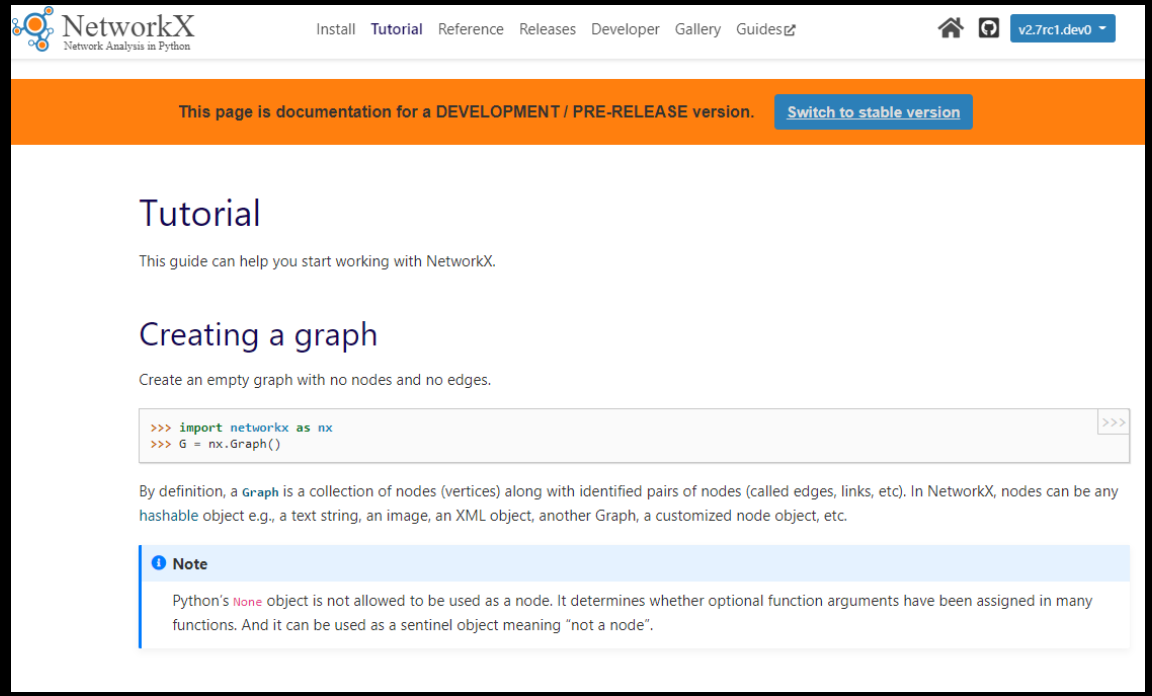
E.g. Tree of Arrays

Composite Can get the best of all worlds

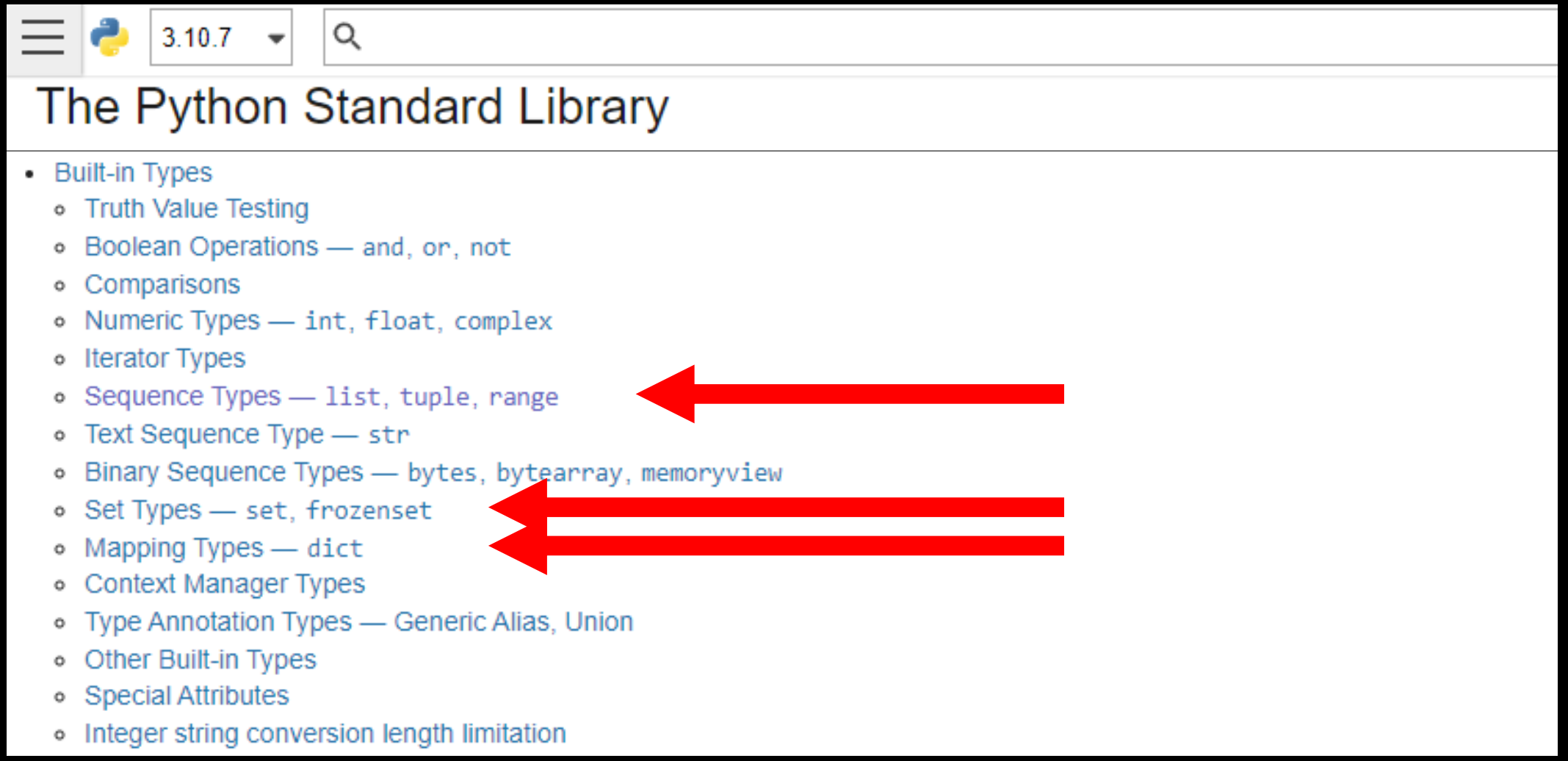
- Need sequencing in key-value?
  - Put the keys in a link list

Networks or Graphs

- More general than Trees are Graphs (sometimes called Networks)
- A good python library for graphs is NetworkX



Built in python built-in implementations



Built in Lists, allow stacks and queues

`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)	
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*	(2)
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*	(3)
`s.reverse()`	reverses the items of *s* in place	(4)

Dict is basically a Hash Table (Key value store)

Mapping Types — `dict`

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

class **dict**(***kwargs*)
class **dict**(*mapping*, ***kwargs*)
class **dict**(*iterable*, ***kwargs*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

Standard Library More Advanced

- Binary Data Services
  - `struct` — Interpret bytes as packed binary data
  - `codecs` — Codec registry and base classes
- Data Types
  - `collections` — Container datatypes
  - `collections.abc` — Abstract Base Classes for Containers
  - `heapq` — Heap queue algorithm
  - `bisect` — Array bisection algorithm
- Concurrent Execution
  - `queue` — A synchronized queue class

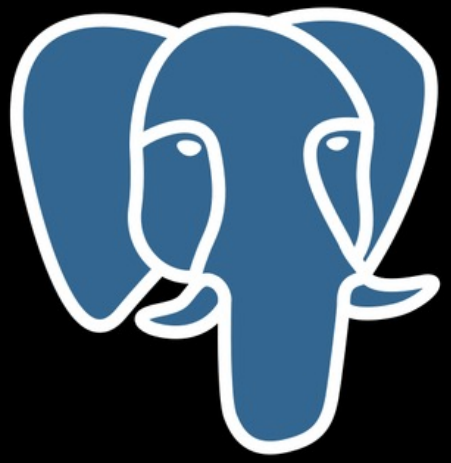
Tabular Data … one way to represent

- Columns: Keys
- Rows: List of values

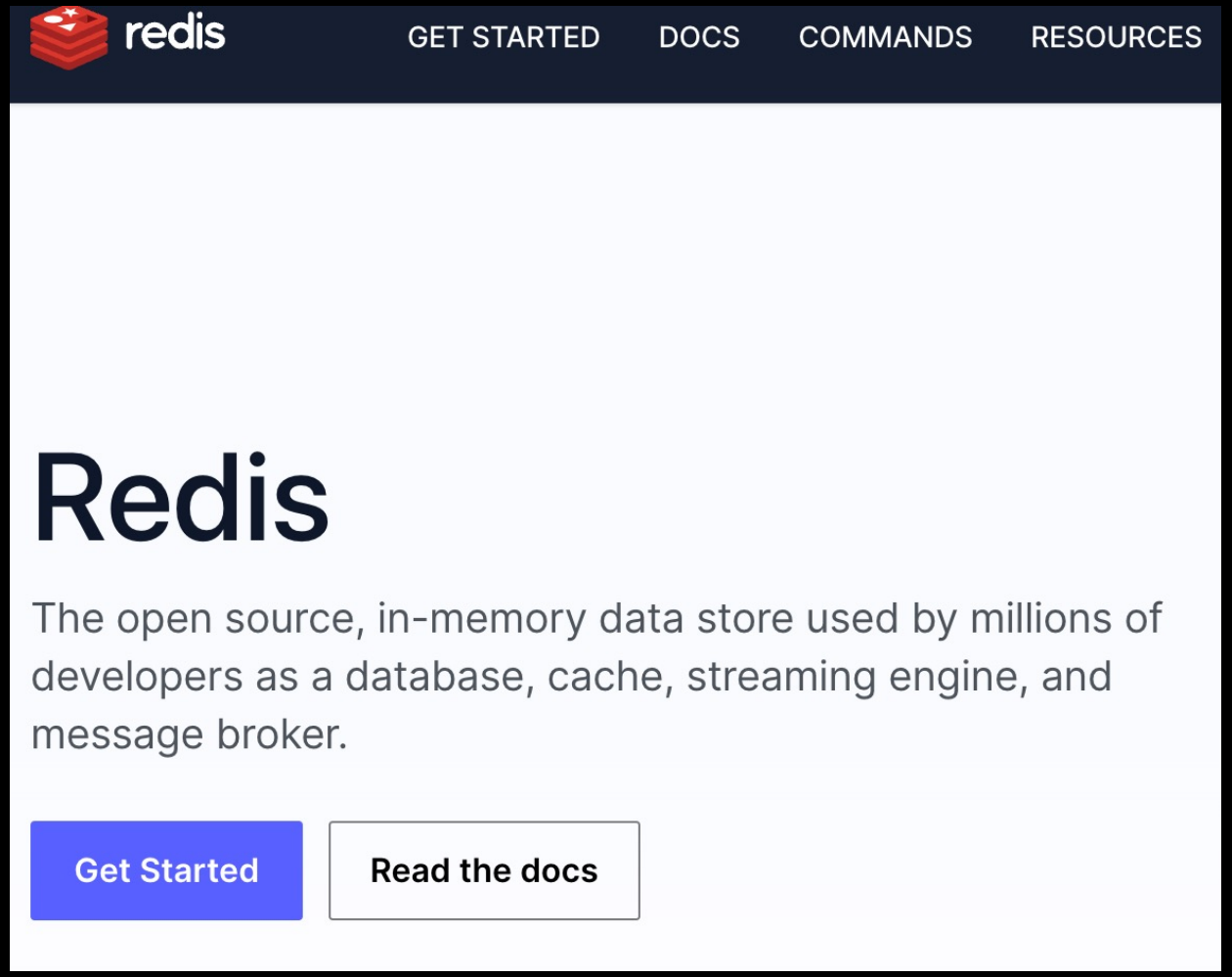
Student	Math_Hours	CS_Hours	ML_Hours
Student_1	10	8	6
Student_2	8	7	5
Student_3	9	6	7
Student_4	7	9	8
Student_5	6	7	6

```
data = {
'Student': ['Student_1', 'Student_2', 'Student_3', 'Student_4', 'Student_5'],
'Math_Hours': [10, 8, 9, 7, 6],
'CS_Hours': [8, 7, 6, 9, 7],
'ML_Hours': [6, 5, 7, 8, 6]
}
```

Tabular Data Usually In a SQL Database RDBSM



In Memory Data Structures for Speed



Graph Database