Data Structures

When to use what, why, and how

Python Data Structures

- list
- tuple
- set
- dictionary

Common Functions

- indexed
  - access
  - search
- mutable
  - add
  - delete
  - and indexed
    - set
  - and ordered
    - append
    - insert
- contains

list - [0, 1, 2]

- mutable
- ordered
- indexed by position (number)
- homogenous

list - when to use it

- need random access to elements
- will deal with items individually
- when in doubt

list - when to use it and why

- need random access to elements
  - indexed by position
- will deal with items individually
  - homogenous
- when in doubt
  - mutable
  - ordered
  - indexed by position

list - how to use it

- indexed
  - access
  - search
- mutable
  - add
  - delete
  - and indexed
    - set
  - and ordered
    - append
    - insert
- contains

list - how to use it

- indexed
  - access: l[<index>]
  - search: l.index(<item>)
- mutable
  - add: l.append(<item>)
  - delete: l.remove(<item>)
  - and indexed
    - set: l[<index>] = <item>
  - and ordered
    - append: l.append(<item>)
    - insert: l.insert(<index>, <item>)
- contains: <item> in l

list - how to use it

- indexed
  - **access**: O(1)
  - **search**: O(n)
- mutable
  - **add**: O(1)
  - **delete**: O(n)
  - and indexed
    - **set**: O(1)
  - and ordered
    - **append**: O(1)
    - **insert**: O(n)
- **contains**: O(n)

list - how to use it

- indexed
  - **access:** l[<index>]
  - **search:** l.index(<item>)
- mutable
  - **add:** l.append(<item>)
  - **delete:** l.remove(<item>)
  - and indexed
    - **set:** l[<index>] = <item>
  - and ordered
    - **append:** l.append(<item>)
    - **insert:** l.insert(<index>, <item>)
- **contains:** <item> in l

tuple - (0, 1, 2)

- immutable
- heterogenous
- ordered
- indexed by position (number)

tuple - when to use it

- order and positions are meaningful and consistent
- deal with data as a coherent unit

tuple - when to use it and why

- order and positions are meaningful and consistent
  - immutable
  - heterogenous
  - ordered
  - indexed by position
- deal with data as a coherent unit
  - heterogenous

tuple - how to use it

- indexed
  - access
  - search
- mutable
  - add
  - delete
  - and indexed
    - set
  - and ordered
    - append
    - insert
- contains

tuple - how to use it

- indexed
  - access: **t[<index>]**
  - search: **t.index(<item>)**
- mutable: **N/A**
  - add
  - delete
  - and indexed
    - set
  - and ordered
    - append
    - insert
- contains: **<item> in t**

tuple - how to use it

- indexed
  - access: O(1)
  - search: O(n)
- mutable: **N/A**
- contains: O(n)

tuple - how to use it

- indexed
  - access: `t[<index>]`
  - search: `t.index(<item>)`
- mutable: **N/A**
- contains: `<item> in t`

set - {0, 1, 2}

- mutable
- unordered
- no duplicate elements

set - when to use it

- need fast membership checking
- comparing with other sets

set - when to use it and why

- need fast membership checking
    - no duplicate elements
- comparing with other sets

set - how to use it

- indexed
  - access
  - search
- mutable
  - add
  - delete
  - and indexed
    - set
  - and ordered
    - append
    - insert
- contains

set - how to use it

- indexed: **N/A**
  - access
  - search
- mutable
  - add: **s.add(<item>)**
  - delete: **s.remove(<item>)** OR **s.discard(<item>)**
  - and indexed: **N/A**
    - set
  - and ordered: **N/A**
    - append
    - insert
- contains: **<item> in s**

set - how to use it

- indexed: **N/A**
- mutable
  - **add**: O(1) | O(n)
  - **delete**: O(1) | O(n)
  - and indexed: **N/A**
  - and ordered: **N/A**
- **contains**: O(1) | O(n)

set - how to use it

- indexed: **N/A**
- mutable
  - **add**: `s.add(<item>)`
  - **delete**:
    - `s.remove(<item>)` # KeyError if <item> not in s
    - `s.discard(<item>)` # nothing if <item> not in s
  - and indexed: **N/A**
  - and ordered: **N/A**
- **contains**: `<item> in s`

set - comparing with other sets

- do s and s' have no elements in common?
  s.isdisjoint(s')
- is s a subset of s'? == is every elt in s in s'?
  s.issubset(s') == s <= s'
- new set w/ elements from s and s'
  s.union(s') == s | s'
- new set w/ elements that s and s' have in common
  s.intersection(s') == s & s'
- new set w/ elements in s, but not s'
  s.difference(s') == s - s'
- new set w/ elements in s or s', but not both
  s.symmetric_difference(s') == s ^ s'

dictionary - {'zero': 0, 1: 'one', (2,'two'): '2'}

- mutable
- unordered
- indexed by unique keys
  - string, number, or tuples containing keys

dictionary - when to use it

- need fast membership checking
- when a tuple won’t work

dictionary - when to use it and why

- need fast membership checking
  - indexed by unique keys
- when a tuple won’t work
  - need to update: mutable
  - too many fields: indexed by unique keys

dictionary - how to use it

- indexed
  - **access**
  - **search**
- mutable
  - **add**
  - **delete**
  - and indexed
    - **set**
  - and ordered
    - **append**
    - **insert**
- **contains**

dictionary - how to use it

- indexed
  - access: d[<key>]
  - search: for k,v in d.items(): if v==<value>: return k
- mutable
  - add: d[<key>] = <value>
  - delete: del d[<key>]
  - and indexed
    - set: d[<key>] = <value>
  - and ordered: N/A
    - append
    - insert
- contains: <value> in d.values() AND <key> in d.keys()

dictionary - how to use it

- indexed
    - **access:** O(1) | O(n)
    - **search:** O(n)
- mutable
    - **add:** O(1) | O(n)
    - **delete:** O(1) | O(n)
    - and indexed
        - **set:** O(1) | O(n)
    - and ordered: **N/A**
- contains
    - **value:** O(n)
    - **key:** O(1) | O(n)

dictionary - how to use it

- indexed
  - **access**: d[<key>]
  - **search**: for k,v in d.items(): if v==<value>: return k
- mutable
  - **add**: d[<key>] = <value>
  - **delete**: del d[<key>]
  - and indexed
    - **set**: d[<key>] = <value>
  - and ordered: **N/A**
- contains
  - **value**: <value> in d.values()
  - **key**: <key> in d.keys()

What data structure should we use to...

- Track how a stock performs each day
- Store an (x, y) coordinate
- Store a to-do list

What data structure should we use to...

- Track how a stock performs each day
  list
- Store an (x, y) coordinate
  tuple
- Store a to-do list
  list or tuple

What data structure should we use to...

- Store a to-do list (and check off tasks)
- Manage inventory
- Describe a car

What data structure should we use to...

- Store a to-do list (and check off tasks)

  list
- Manage inventory at a grocery store

  dictionary
- Describe a car

  tuple or dictionary

What data structure should we use to...

Map the path we took on a hike

What data structure should we use to...

Map the path we took on a hike

list of tuples

What data structure should we use?

My roommate and I use an app that lets us collaboratively build our grocery list. The app lets us both view the list, so he takes the top half and I take the bottom half.

We bump into each other in the cereal aisle as we both get a box of Cap’n Crunch. Both of our lists our correct, but they have our items in different orders.

What data structure should we use?

What data structure should we use to make sure we still get the correct items in the correct quantities on our list?

What data structure should we use?

What data structure should we use to make sure we still get the correct items in the correct quantities on our list?

set

What data structure should we use?

If you wrote the grocery list app, what data structure would you use to store the list?

What data structure should we use?

If you wrote the grocery list app, what data structure would you use to store the list?

list

What data structures should we use to...

Build a calendar for 2015

What data structures should we use to...

tuple (i = day) of lists of tuples (events)

```
([],
 [],
 [(“my bday”, “”), (“midterm 2”, “latimer”)],
 ...)
```

What data structures should we use to...

tuple (i = day) of lists of dictionaries (events)

```
([],
 [],
 [{title: “my bday”, location: “”},
  {title: “midterm 2”, location: “latimer”}],
 ...)
```

What data structures should we use to...

tuple (i = month) of tuple (i = day) of lists of tuples/dictionaries (events)

```
(([], [],
  [{title: “my bday”, location: “”},
   {title: “midterm 2”, location: “latimer”}],
  ...),
 (...),
 ...)
```

Bonus! Recursive Data Structures

- linked list
- tree

linked list - Link(0, Link(1, Link(2)))

- mutable
- homogenous
- ordered
- memory-efficient

linked list - when to use it

- need O(1) insertions/deletions
- don’t know how many items up front
- don’t need random access to elements
- need to insert items in middle of list

linked list - interface

- l = Link(<first>, <rest>=empty)
- l.first
- l.rest
- l[<index>]
- len(l)

tree - Tree(0, [Tree(1), Tree(2)]

- mutable
- ordered

tree - when to use it

- data is hierarchical
- don’t know how many items up front

tree - interface

- t = Tree(entry, branches)
- t.entry
- t.branches
- t.is_empty
- t.left
- t.right

binary tree - BinaryTree(0, BinaryTree(1), BinaryTree(2))

- mutable
- ordered

binary tree - when to use it

- binary search tree: search at moderate pace
  - (list < bst < linked list)

binary tree - interface

- b = BinaryTree(entry, left=empty, right=empty)
- b.entry
- b.is_empty
- b.left
- b.right