CIS 1100

Searching

Python

Fall 2024

University of Pennsylvania

Overview

We often need to search for an item in a collection

- *Is this student in this recitation roster?*
- *Is this username in our user database?*
- *Is there any data point in our dataset that matches this description?*

In this module, we will learn about how to search for an element in a list.

Learning Objectives

- To be able to use **linear search** to find an element inside an sequence
- To be able to use **binary search** to find an element inside an sequence
- To be able to know when to use linear search and when to use binary search

Problem: Search

Formally, given a sequence of values and a target value, we want to determine if the target value is in the sequence, and if so, where it is located.

Solution: `.index()`

Python has a built-in solution: `sequence.index(target)` returns the position of `target` inside of the `sequence`, or raises a `ValueError` if the `target` is not present.

- You'll just use this (or `.find()` for strings) most of the time
- BUT!
  - How does it work?
  - `index()` the best solution in all cases? Are there better strategies?
    - (There are.)

Problem: Search

Formally, given a sequence of values and a target value, we want to determine if the target value is in the sequence, and if so, where it is located.

- in our case, the "sequence of values" could be a list, tuple, string, `Series`, `DataFrame`...
- the "target value" is the value we are searching for
- the location is the index of the value in the sequence, or `-1` if it's not present.

Concept: The "Feasible Region"

In any problem, the **feasible region** is the name for the set of possible values that might be a solution.

- In the context of search, the feasible region refers to the set of indices in the sequence that might contain the target value.
- A set of indices is functionally a region of the sequence where the target value might be found.

In our search algorithms, we repeatedly reduce the feasible region until we find the target value, or until we determine that the target value is not present in the sequence.

CIS 1100

Linear Search

Python
Fall 2024
University of Pennsylvania

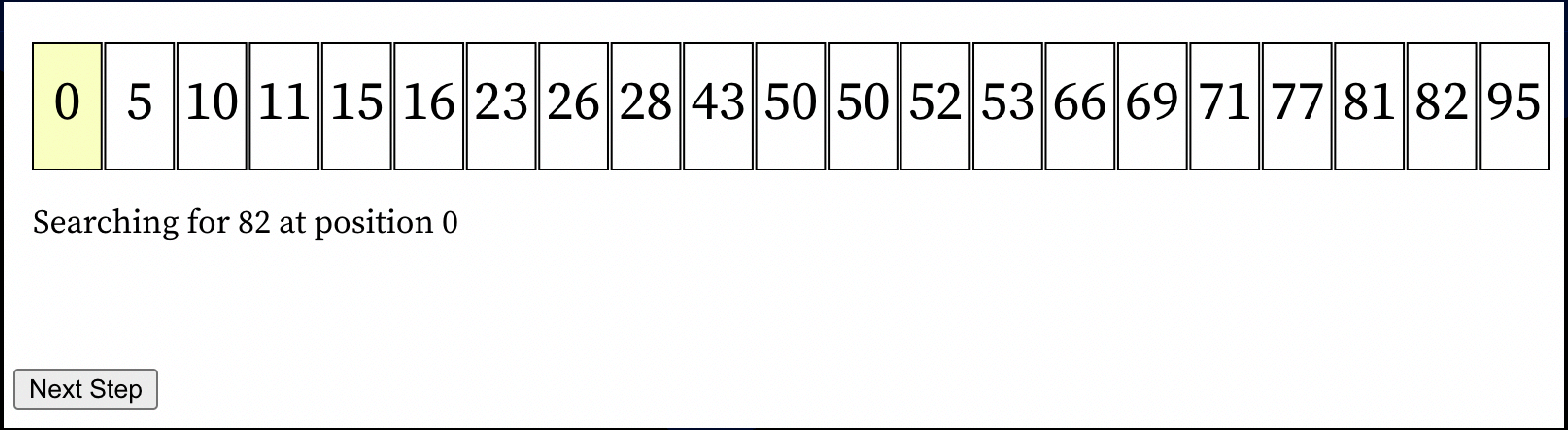
Linear Search

Used to search for a value (the target) in an **unsorted list**

- Use a loop to iterate over the values
- Start at the first element and move to the next element until the target is found
- Returns the position of the target if it was found in the sequence, or -1 if the target was not found in the sequence

With each iteration, we reduce the feasible region by one element.

Linear Search Example



(this image is a link)

Linear Search

```
def linear_search(sequence, target):
    for idx, element in enumerate(sequence):
        if element == target:
            return idx
    return -1
```

```
>>> linear_search(range(30, 300, 4), 30)
0
>>> linear_search(range(30, 300, 4), 262)
58
>>> linear_search(range(30, 300, 4), 31)
-1
```

Linear Search: Thinking Critically

How many iterations of the for loop will we need if…

- the target is the first element in the sequence?
- the target is the 10th element in the sequence?
- the target is not in the sequence?

Linear Search: Thinking Critically

How many iterations of the for loop will we need if...

- the target is the first element in the sequence? **1**
- the target is the 10th element in the sequence? **10**
- the target is not in the sequence? **len(sequence)**

Linear Search: Properties

Linear search is...

- **Complete:** we'll always get an answer
- **Correct:** we'll always get the right answer

These are desirable properties, but linear search is not always the most **efficient.**

- May require more time (~more iterations) than other searching algorithms for "average use"

A Contrasting Point of View

Here's a dumb searching algorithm called **Bogo Search**

```
from random import randrange
def bogo_search(sequence, target):
    while True:
        idx = randrange(len(sequence)) # picks a random index to look at
        if sequence[idx] == target:
            return idx
```

Not even **complete:** if we got unlucky, we could accidentally just look at the same (wrong) index infinitely many times in a row and never return.

CIS 1100

Binary Search

Python

Fall 2024

University of Pennsylvania

Can we do better than linear search? Can we be...

- complete?
- correct?
- faster?

The answer is "yes, yes, and **sometimes."**

Binary Search

Used to search for a target value in a **sorted sequence only**

- Compares the target with the value at the middle index (middle element)
  - If the middle element is the target element, then we're done!
  - If the target is less than the middle element, then we search for the target in the **left half of the sequence** (the positions before the middle element)
  - If the target is greater than the middle element, then we search the target in the **right half of the sequence** (the positions after the middle element)
- Repeat on the remaining search area of the sequence until
  - the element is found
  - there is no feasible search area left

Binary Search

Searching for `"Dustin"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
low			middle			high

- `middle = (low + high) // 2 = 3`
- `names[middle]` is `"Elliot"`, which comes after `"Dustin"` alphabetically.
- So, if `"Dustin"` is present, it must be between positions `0` and `middle - 1`.

Binary Search

Searching for `"Dustin"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
low	middle	high				

- `middle = (low + high) /// 2 = 1`
- `names[middle]` is `"Debbie"`, which comes before `"Dustin"` alphabetically.
- So, if `"Dustin"` is present, it must be between positions `middle + 1` and `2`.

Binary Search

Searching for `"Dustin"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
		low, middle, high				

- `middle = (low + high) // 2 = 2`
- `names[middle]` is `"Dustin"`, which is the target element! So, we return `middle`.

Binary Search: Searching for an Element not Present

Searching for `"Drew"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
low			middle			high

- `middle = (low + high) // 2 = 3`
- `names[middle]` is `"Elliot"`, which comes after `"Drew"` alphabetically.
- So, if `"Drew"` is present, it must be between positions `0` and `middle - 1`.

Binary Search: Searching for an Element not Present

Searching for `"Drew"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
low	middle	high				

- `middle = (low + high) // 2 = 1`
- `names[middle]` is `"Debbie"`, which comes before `"Drew"` alphabetically.
- So, if `"Drew"` is present, it must be between positions `middle + 1` and `2`.

Binary Search: Searching for an Element not Present

Searching for `"Drew"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
		low, middle, high				

- `middle = (low + high) // 2 = 2`
- `names[middle]` is `"Dustin"`, which comes after `"Drew"` alphabetically.
- So, if `"Drew"` is present, it must be between positions `2` and `middle - 1`.

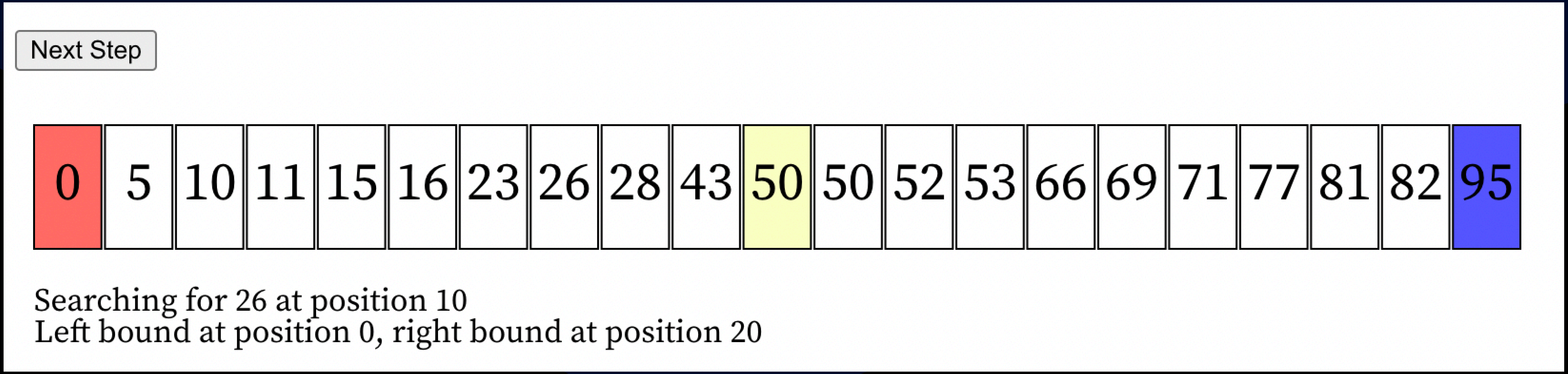
Binary Search: Searching for an Element not Present

Searching for `"Drew"` in the sequence `names`!

Caryn	Debbie	Dustin	Elliot	Jacquie	Jon	Rich
0	1	2	3	4	5	6
	high	low				

- `high` is now less than `low`. The "feasible search area" is now totally empty.
- So, we return `-1` to indicate that the target was not found in the sequence.

Binary Search, Interactive



(this image is a link)

Binary Search

```python
def binary_search(sequence, target):
    low_index, high_index = 0, len(sequence) - 1
    while low_index <= high_index:
        middle_index = (low_index + high_index) // 2
        if target < sequence[middle_index]:
            high_index = middle_index - 1
        elif target > sequence[middle_index]:
            low_index = middle_index + 1
        else:
            return middle_index
    return -1
```

Properties of Binary Search

- Binary Search is **complete** since each iteration of the `while` loop shrinks our feasible search area down to a point where we'll stop, or we return the index where we find the target.
- Binary Search is **correct** since we return the index of the target when we find it and we only return `-1` when the element could not have been present in the sequence.
    - This is only guaranteed if the sequence was sorted, though!
- Is Binary Search any more **efficient** than Linear Search?

CIS 1100

Comparing Linear
& Binary Search

Python
Fall 2024
University of Pennsylvania

Linear Search vs. Binary Search

- Binary search is faster "on average" than linear search 😄🎉🆒
  - Per iteration, binary search shrinks the feasible region by half the remaining elements, linear search only by one element.
  - In both cases, max number of iterations needed is bounded above by the number of iterations needed to shrink the feasible region to empty.
  - On average, binary search requires fewer iterations of the search loop
  - (when is binary search not faster then linear search?)

Linear Search vs. Binary Search

Runtime analysis: how many iterations will it take to determine that the target is not in the sequence?

Length of the sequence	Linear Search	Binary Search
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7

Linear Search vs. Binary Search

Runtime analysis: how many iterations will it take to determine that the target is the first element of the sequence?

Length of the sequence	Linear Search	Binary Search
2	1	2
4	1	3
8	1	4
16	1	5
100	1	7

Linear Search & Binary Search

Linear search is...

- Usable when your sequence is not sorted to start with
- As efficient as any search algorithm can be when you don't know anything about the sequence ahead of time

Binary search is...

- Only usable when your sequence is sorted to start with
- Significantly more efficient than linear search *on average.*

CIS 1100

Data Structures & Efficiency

Python
Fall 2024
University of Pennsylvania

Data Structures

- Python comes with a ton of built-in **data structures,** or organized containers of data.
  - lists, tuples, sets, dicts
- What is the point of having so many different kinds?
  - We know that some support operations that others do not
    - Sometimes they have the same operations, though—why?
  - Most are mutable, but tuples are not—why?

Efficiency & Complexity

The **complexity** of an algorithm or a data structure refers to amount of some resource that is required to perform that algorithm or maintain that data structure.

- Resources are usually:
  - **time**: how many CPU cycles—how many operations—must be performed.
  - **space**: how much of a computer's memory must be used
- Computers are extremely fast. But they are not infinitely fast.
- Efficiency is **not** your primary concern as an intro programming student, but you should try to build good habits early if you can.
  - In "bigger" programs, we need the right tool for the job if the problem is to be feasible at all.

Estimating Time Complexity

In future courses, you'll learn about mathematical analysis for proving time and space complexity. But for now... a timer!

```
python -m timeit "<small snippet of code>"
```

e.g.

```
$ python -m timeit "list('howdy, partner')"
2000000 loops, best of 5: 134 nsec per loop
```

Evaluating `list('howdy, partner')` 2000000 times took an average of 134 nanoseconds per evaluation.

Estimating Time Complexity

Can also add a `setup` statement using `-s <setup-statement>` to provide a line of code that should be run once before the timer.

```
python -m timeit -s "<setup-statement>" "<small snippet of code>"
```

e.g.

```
$ python -m timeit -s "l = list(range(10000))" "l.append(-1)"
10000000 loops, best of 5: 26.3 nsec per loop
```

CIS 1100

Measuring Efficiency of Common Uses

Python
Fall 2024
University of Pennsylvania

Efficiency & Data Structures

Different operations have different costs in different data structures.

- Heck, even *the same operation in the same data structure* can have different costs!

```
$ python -m timeit -s "l = list(range(10000))" "l.append(-1)"
10000000 loops, best of 5: 26.3 nsec per loop
$ python -m timeit -s "l = list(range(10000))" "l.insert(len(l), -1)"
5000000 loops, best of 5: 65.8 nsec per loop
$ python -m timeit -s "l = list(range(10000))" "l.insert(0, -1)"
20000 loops, best of 5: 10.4 usec per loop
```

➡️ Inserting at the end of a list is faster than inserting at the beginning! (True for tuples too.)

Efficiency & Data Structures

Sets and dicts are very fast for adding, too:

```
$ python -m timeit -s "s = set(range(10000))" "s.add(-1)"
10000000 loops, best of 5: 28.1 nsec per loop
$ python -m timeit -s "d = dict(enumerate(range(10000)))" "d[-1] = -2"
10000000 loops, best of 5: 21.2 nsec per loop
```

➡️ Inserting at the end of a list is faster than inserting at the beginning! (True for tuples too.)

Efficiency and Costs

`in` is an operation supported for strings, tuples, lists, sets, and dict. If our goal is tracking *membership*, which should we use?

Problem setting: forbidding common passwords

```
123456 123456789 password adobe123 12345678
qwerty 1234567 111111 photoshop 123123 1234567890
000000 abc123 1234 adobe1 macromedia azerty iloveyou
aaaaaa 654321 12345 666666 sunshine 123321 letmein
monkey asdfgh password1 shadow princess dragon
adobeadobe daniel computer michael 121212 charlie
master superman qwertyuiop 112233 asdfasdf jessica
1q2w3e4r welcome 1qaz2wsx 987654321 fdsa 753951 chocolate
```

Checking If a Password is Known to Be Bad

Setup: Read a file of known "bad passwords" into a data structure.

Experiment: Search for `'chocolate'`, the *last* password in the file.

```
$ python -m timeit -s "ds = list(open('passwords.csv', 'r').readlines())" "'chocolate' in ds"
200000 loops, best of 5: 1.1 usec per loop
$ python -m timeit -s "ds = tuple(open('passwords.csv', 'r').readlines())" "'chocolate' in ds"
200000 loops, best of 5: 1.09 usec per loop
$ python -m timeit -s "ds = open('passwords.csv', 'r').read()" "'\nchocolate\n' in ds"
2000000 loops, best of 5: 176 nsec per loop
$ python -m timeit -s "ds = set(open('passwords.csv', 'r').readlines())" "'chocolate' in ds"
20000000 loops, best of 5: 13.1 nsec per loop
```

➡️ Checking for membership in a set can be nearly

two orders of magnitude faster than checking in a list!

Checking If a Password is Known to Be Bad

Setup: Read a file of known "bad passwords" into a data structure.

Experiment: Search for `'123456'`, the *last* password in the file.

```
$ python -m timeit -s "ds = list(open('passwords.csv', 'r').readlines())" "'123456' in ds"
200000 loops, best of 5: 367 nsec per loop
$ python -m timeit -s "ds = tuple(open('passwords.csv', 'r').readlines())" "'123456' in ds"
200000 loops, best of 5: 368 nsec per loop
$ python -m timeit -s "ds = open('passwords.csv', 'r').read()" "'\n123456\n' in ds"
2000000 loops, best of 5: 246 nsec per loop
$ python -m timeit -s "ds = set(open('passwords.csv', 'r').readlines())" "'123456' in ds"
20000000 loops, best of 5: 13.1 nsec per loop
```

➡️ Checking for membership in a set is usually at least 10x faster than checking in a list!

So... Sets for Everything?

No.

What if we wanted to be able to look up *the most commonly used password?* What if we wanted to look up *the frequency with which a certain password was used?*

Sets Are Not Ordered!

Finding the most commonly used password:

- We could store the passwords in a `list` in descending order of frequency.
  - `passwords[0]` gives us our answer
- We could store the passwords as keys and their frequencies as values in a `dict`.

```
most_common = ""
for password, freq in passwords.items()
    if freq > password[most_common]:
        most_common = password
```

  - Not necessarily the most fast, but at least we can do it.
- We could put the passwords in a set, and then do...?

Choosing the Right Tool for the Job

Data Structure	Use for…	Avoid…
`list`	Ordered sequences, sorted sequences, collections that grow from the end	Repeated membership checks, inserting elements at positions other than the end.
`set`	Unordered collections, repeated membership checks, collections that can grow and shrink over time	When the order of the elements matters
`dicts`	Key-value associations, hierarchical data structures (next module), repeated membership checks, collections that can grow and shrink over time	When the order of the elements matters

CIS 1100

What About Tuples?

Python
Fall 2024
University of Pennsylvania

Okay, What About Tuples?

Look, you can just use a `list` *almost* anywhere you want an ordered sequence of elements.

Tuples as Hashable Types

But, tuples are **necessary** when you want to use sequences as keys for dictionaries!

Suppose you have a collection of circles you want to draw in different colors.

- Could use a sequence of `<x_center, y_center, radius>` as a key
- Could use a sequence of `<r, g, b>` as a value for the color.

Lists Can't Be Keys

```
my_circles = {[0.5, 0.5, 0.1] : [255, 255, 255],
              [0.2, 0.8, 0.2] : [0, 0, 180]}
```

👇

```
TypeError: unhashable type: 'list'
```

Tuples Can Be Keys

```
my_circles = {(0.5, 0.5, 0.1) : [255, 255, 255],
              (0.2, 0.8, 0.2) : [0, 0, 180]}
```

Changing the keys to tuples works!

- Don't even have to change the types of the values
- This is because lists are mutable, and so the way that we "look them up" in the dictionary might change over time.
  - More on this in an algorithms class..

Tuples Have Finality

Suppose we wrote a function that calculated the centroid of a given shape and returned it as an `(x, y)` coordinate.

- Could return the coordinate as a list or a tuple.
- But what would it mean to append a value to the coordinate pair?

```
l = find_centroid(my_shape)
l.append("password") # ???????????
```

- Tuples and their immutability can convey the message: "final answer!"