CIS 1100

Searching (Lecture)

Python
Fall 2024
University of Pennsylvania

Review: Search as a Problem

Given a sequence and a target element, find a position at which that target is found inside of the sequence, or report that the target cannot be found.

Both linear search and binary search will have the following signatures:

```python
def linear_search(seq, target):
    ...
```

Activity: Searching Practice

What do the following function calls return?

- **(S7)** `linear_search([3, 1, 10, 17, 23, 31, -23], -23)`
- **(S8)** `linear_search([2, 2, 4, 8, 12, 14, 16, 32, 64], 2)`
- **(S9)** `binary_search([2, 2, 4, 8, 12, 14, 16, 32, 64], 2)`
- **(S10)** `binary_search([3, 1, 10, 17, 23, 31, -23], -23)`

Review: Particulars of Binary Search

- Doesn't necessarily return the lowest index at which the target appears in the sequence
- Doesn't necessarily give the correct answer if the input sequence is not sorted

New: Sorting

If you want to put a sequence in sorted order, you can use either `sort()` or `sorted()`

- `.sort()` is a method that sorts a sequence *in-place*, meaning that the sequence itself is changed and the function doesn't return anything.
- `sorted()` is a function that returns a new (shallow) copy of the input sequence in sorted order. The original sequence is unchanged.

Activity: `sort()` and `sorted()`

In **(L11)**, describe what happens when the program is run—what are the outputs? Why?

```
list_of_numbers = generate_random_number_list() # pretend this exists
sorted_list = list_of_numbers.sort()
smallest = sorted_list[0]
largest = sorted_list[-1]
print("The range of numbers generated is ({smallest}, {largest}).")
```

In-Place Operations

In-place operations like `.sort()` or `.reverse()` modify the object they're called on without returning a value.

- These permanently modify the thing you're calling them on, even if that thing was an argument passed into another function.
- These methods do not return any values, so you probably don't mean to save the results in a variable

Checking for Membership

Several operations exist for lists/sequences:

- `.index(target)` performs a linear search to find the index of a `target` on a list.
  - Tragic: raises an error if `target` isn't found in the list 😦
- `in` performs a linear search to find whether a target element is contained in a list.

Activity: `linear_search_contains`

(C12)

Assume we have a function `linear_search(seq, target)`. Can you write a short function `linear_search_contains(seq, target)` that...

- returns exactly what `target in seq` would return (i.e. a boolean)
- calls `linear_search()` as a helper function

Generalizing

We *could* do the same for `binary_search_contains()`:

```
def binary_search_contains(seq, target):
    return binary_search(seq, target) != -1
```

But we do need to be careful making sure that `seq` is sorted to start, otherwise we have a problem.

Generalizing

We could even make `binary_search_contains()` "safe":

```
def safe_binary_search_contains(seq, target):
    seq = sorted(seq)
    return binary_search(seq, target) != -1
```

This leads to an interesting question...

Recall: Binary Search is "Faster" On Average

We say that binary search is faster "on average" than linear search.

So why does Python use linear search to implement `in` *and* `.index()` *when we could just sort the sequence and use binary search instead?*

Speedy Snakes

All code takes time to run. A simple heuristic is that a function's runtime is proportional to the number of iterations of the loops it takes to execute.

Let's approximate "speed" with printed snakes: 🐍

```
def linear_search_contains(seq, target):
    for idx, element in enumerate(sequence):
        print("🐍")
        if element == target:
            return True
    return False
```

(L13): How many snakes are printed if we run `linear_search_contains(range(100), 13)`?

Contains with Binary Search

```
def binary_search_contains(sequence, target):
    low_index, high_index = 0, len(sequence) - 1
    while low_index <= high_index:
        print("🐍")
        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 True
    return False
```

Also **(L13)**: How many snakes are printed if we run `binary_search_contains(range(100), 13)`?

Contains with Binary Search

```
def safe_binary_search_contains(sequence, target):
    sequence = sorted(sequence)
    low_index, high_index = 0, len(sequence) - 1
    while low_index <= high_index:
        print("🐍")
        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 True
    return False
```

Also **(L13)**: How many snakes are printed if we run

`safe_binary_search_contains(shuffle(range(100)), 13)`?

A Whole Other Bundle of Snakes

```python
sequence = sorted(sequence)
```

If we're just counting iterations of while loops, it looks like `binary_search_contains` and `safe_binary_search_contains` have the same "snake price."

But this is a **LIE!** Because sorting also costs an appreciable amount of time. In fact, if `sequence` contains `100` elements, then a call to `sorted(sequence)` would print about **700 SNAKES** on average!

🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍
🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍 (this is only 192 snakes!)

Concluding...

(L15) What is the most number of snakes that a *linear search* could print for a sequence of 100 numbers.

Use this result to summarize in **(C16)** why it's not a good idea to **always** use a binary search method to check if a target value is found inside of a sequence.

(If Time) `__eq__()`

```
class Rhyme:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def to_limerick(self):
        print(f"There once was a guy named {self.first} who thought for sure he could {self.second}")

silly = Rhyme("Steve", "leave")
silly.to_limerick()
```

🖨

```
There once was a guy named Steve who thought for sure he could leave
```

"I Need Six Rhymes On My Desk By 5PM"

```
rhymes_for_steve = [
    Rhyme("Steve", "leave"),
    Rhyme("Steve", "achieve"),
    Rhyme("Steve", "grieve"), # idk
    Rhyme("Steve", "leave"),
    Rhyme("Steve", "heave"),
    Rhyme("Steve", "believe")
]
```

Whoops, I did a duplicate. Let's just get rid of that...

```
rhymes_for_steve = list(set(rhymes_for_steve))
print(len(rhymes_for_steve))
```

Wait... still 6?

Object Equality

Objects that are *structurally* the same as each other will not automatically be considered to be `==` to each other 🙁

```
>>> Rhyme("Steve", "leave") == Rhyme("Steve", "leave")
False
```

`__eq__()` to the rescue!

`__eq__` for Equality

In any class, you can write a method with the signature `def __eq__(self, other)` to define how the `==` operation behaves.

- Called a **"magic method"**—a method that defines the behavior of an operation that's called in a different way than the name of the method would apply.
- A perk of Dataclasses—they implement a reasonable version of `__eq__` for you

```
class Rhyme:
    ... # other stuff

    def __eq__(self, other):
        return self.first == other.first and self.second == other.second
```

```
>>> Rhyme("Steve", "leave") == Rhyme("Steve", "leave")
True
```