CIS 1100

Cost of an Function 

Python
Fall 2024
University of Pennsylvania

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
```

(L11): How many snakes are printed if we run `linear_search_contains(range(100), 13)`?

Contains with Binary Search

```python
def 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 **(L11)**: How many snakes are printed if we run `binary_search_contains(shuffle(range(100)), 13)`?

A Whole Other Bundle of Snakes

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

If we're just counting iterations of while loops, it looks like `binary_search_contains` and `linear_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...

Final thing for **(L11)**: 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 **(C12)** 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.

CIS 1100

Choosing the Right Data Structure 

Python

Fall 2024

University of Pennsylvania

Building Collaborative Definitions

1. Take two minutes to talk to a partner. For each of the following types, try to describe some advantages/drawbacks of each.
2. Then, we'll collect everyone's suggestions to build a collaborative inventory.

Type	Advantages & Uses	Disadvantages & Limitations
`list`		
`set`		
`dict`		
`tuple`		

Rapid Fire

Next five slides have five problem statements. We'll look at each for 45 seconds—feel free to discuss with a partner as we do. In each case, decide which data structure will be best/necessary. (We'll go over this after.)

(M1-5)

(M1)

I have a file called `marathon_timings.txt`
that maps runner names to their marathon times.

I want to sort the runners by time and then save the names in order so that I can quickly look up who was in first place, third place, nineteenth place, etc.

I should use a ... to store my runners

A. list

B. set

C. dict

D. tuple

(M2)

I have a file called `marathon_timings.txt`
that maps runner names to their marathon times.

I want to be able to look up the marathon times of a runner by their names.

I should use a ... to store my runners & their times.

A. list

B. set

C. dict

D. tuple

(M3)

I have a file called `marathon_timings.txt`
that maps runner names to their marathon times.

I want to be able to check a name against the names I have stored in this file.

I should use a ... to store my runner names.

A. list

B. set

C. dict

D. tuple

(M4)

I have a file called `marathon_timings.txt`
that maps runner names to their marathon times.

I want to be able to look up the marathon times of a runner by their names.
I also want to be able to add runners & times after I process this file.

I should use a ... to store & update my runner names and times.

A. list

B. set

C. dict

D. tuple

(M5)

I have a file called `marathon_timings.txt`
that maps runner names to their marathon times.

I want to be able to write a function that returns the name & time of the fastest runner in a single value. It makes the most sense for me to return a ...

A. list

B. set

C. dict

D. tuple

`timeit` & `cProfile`

For timing individual lines:

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

For timing whole programs:

```
python -m cProfile your_filename.py
```

Pointing Out Inefficiency

```
def add_to_sorted_list(sorted_list, other_numbers):
  """
  Add all of the values from other_numbers
  to the list sorted_list. sorted_list is
  already sorted, and the returned value should
  be sorted, too.
  """
  output = list(sorted_list)
  for number in other_numbers:
    output.append(number)
    output.sort()
  return output
```

In **(C14)**, can you think of a way to make this function more efficient?

(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
```