CL13: Sets, Dictionaries, and Intro to Time Complexity

Announcements

- EX03 due *tomorrow* (June 4)!
- Quiz 03 on Friday
  - Practice quiz + key posted today
  - Virtual Review Session on Thursday at 6:00pm
  - Please visit Office Hours for help!

With a neighbor, try diagramming:

```
1   def intersection(a: list[str], b: list[str]) -> list[str]:
2       result: list[str] = []
3
4       idx_a: int = 0
5       while idx_a < len(a):
6           idx_b: int = 0
7           found: bool = False
8           while not found and idx_b < len(b):
9               if a[idx_a] == b[idx_b]:
10                  found = True
11                  result.append(a[idx_a])
12              idx_b += 1
13          idx_a += 1
14
15      return result
16
17
18  foo: list[str] = ["a", "b"]
19  bar: list[str] = ["c", "b"]
20  print(intersection(foo, bar))
```

… and after diagramming:

Assume our unit of "operation" is the number of times the block of lines #9-12 are evaluated.

Q1. Can different values of a and b lead to a difference in the number of operations required for the intersection function evaluation to complete?

Q2. If so, provide example item values for a and b which require the fewest operations to complete? Then try for the maximal operations to complete?

Q3. Assuming the item values of a and b are random and unpredictable, about how many operations does this function take to complete?

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```

As the lengths of `a` and `b` grow, the number of operations grows *quadratically*

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        idx_b: int = 0
        found: bool = False
        while not found and idx_b < len(b):
            if a[idx_a] == b[idx_b]:
                found = True
                result.append(a[idx_a])
            idx_b += 1
        idx_a += 1

    return result


foo: list[str] = ["a", "b"]
bar: list[str] = ["c", "b"]
print(intersection(foo, bar))
```

- Outer while loop iterates through each element of `a`
  - *If there are N elements, we'll iterate N times*
- And within each iteration of the outer while loop…
- The inner while loop iterates through elements of `b` until either:
  - We find a value that == the current element in `a` OR,
  - We have "visited" (accessed) every element in `b`
    - *If there are M elements in `b`, we'll iterate up to M times*

Assuming `a` and `b` both have 3 elements…

1. Example of values of `a` and `b` that will cause the ***fewest*** operations to occur?
   `intersection(a=["a", "a", "a"], b=["a", "b", "c"])`
2. Example of values of `a` and `b` that will cause the ***most*** operations to occur?
   `intersection(a=["a", "b", "c"], b=["d", "e", "f"])`

If list `a` has N elements and list `b` has M elements, the "worst case scenario" is that this code will cause N*M operations to occur.

Comparing lists and sets

```python
def intersection(a: list[str], b: list[str]) -> list[str]:
    result: list[str] = []

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.append(a[idx_a])
        idx_a += 1

    return result
```

```python
def intersection(a: list[str], b: set[str]) -> set[str]:
    result: set[str] = set()

    idx_a: int = 0
    while idx_a < len(a):
        if a[idx_a] in b:
            result.add(a[idx_a])
        idx_a += 1

    return result
```

Suppose **`a`** and **`b`** each had 1,000,000 elements. The worst case difference here is approximately 1,000,000 operations, versus 1,000,000**2 or 1,000,000,000,000 operations.

If your device can perform 100,000,000 operations per second, then...

A call to **`a`** will complete in 2.78 hours and **`b`** will complete in 1/100th of a second.

Sets!

Sets, like lists, are a *data structure* for storing collections of values.

Unlike lists, sets are *unordered* and each value has to be *unique*.
    Lists: *always* zero-based, sequential, integer indices!

Benefit of sets: testing for the existence of an item takes only one “operation,” regardless of the set’s size.

```
pids: set[int] = {730120710, 730234567, 730000000}
```

To add a value to the set:

```
pids.add(730123456)  # Add a value to the set
```

To remove a value from the set:

```
pids.remove(730120710)  # Remove a value from the set
```

Data structure	Allows duplicates?	Ordered?	Fast lookups?	Use Case
list [ ]	✅	✅	❌	Ordered collections
set {}	❌	❌	✅	Unique values, membership testing (fast lookups)
dictionary {key: value}	✅ (duplicate values allowed; keys must be unique!)	It's complicated	✅	Mappings, fast lookups, counting

Match the Data Structure to its Application

Set

List

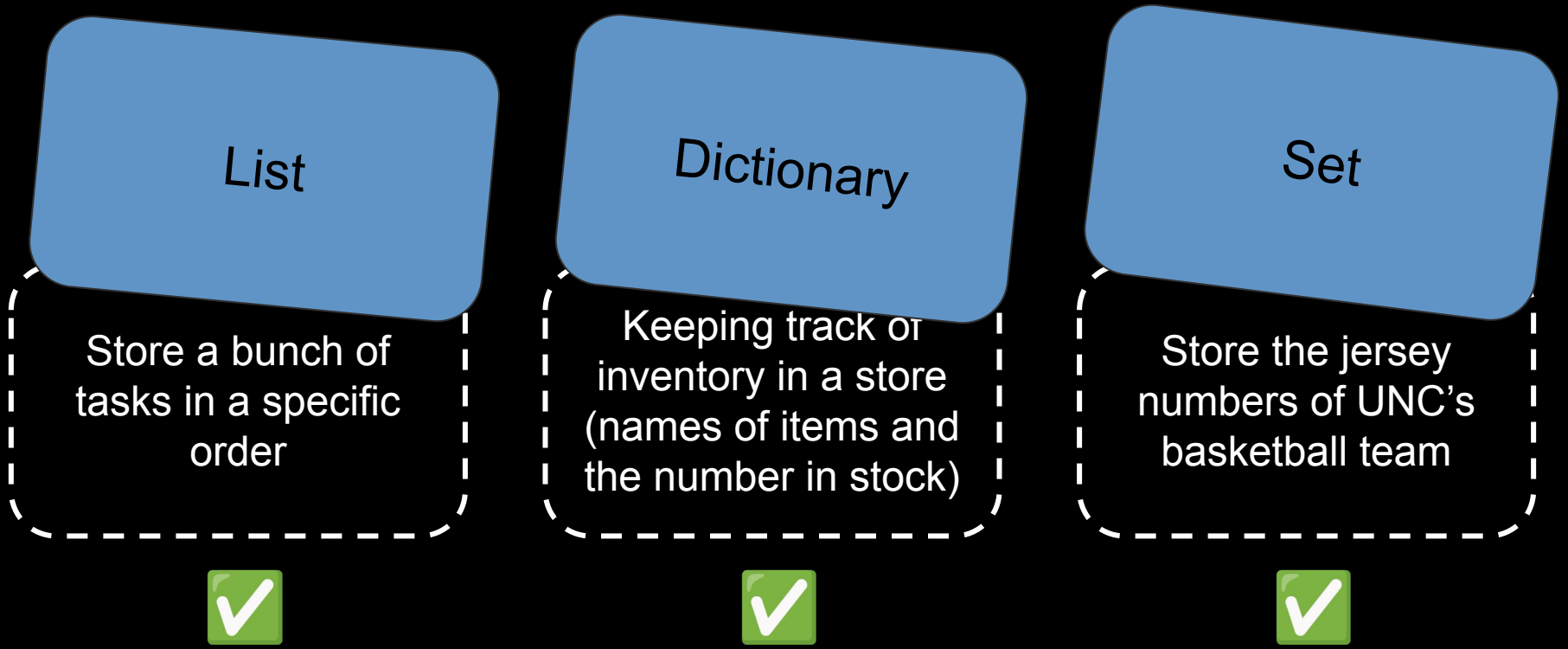
Dictionary

Store a bunch of tasks in a specific order

Keeping track of inventory in a store (names of items and the number in stock)

Store the jersey numbers of UNC's basketball team

Match the Data Structure to its Application



```
1 vend: dict[str,str] = {"A1":"Oreos", "A2":"Lays", "B1":"Coke", "B2":"7up"}
2 flavors: set[str] = {"Orange", "Cherry", "Lime"}
```

2.1. What will be printed?

```
1 for prod in vend:
2   print(prod)
```

2.2. What will be printed?

```
1 for prod in vend:
2   print(vend[prod])
```

2.3. What will be printed?

```
1 for flav in flavors:
2   print(flav)
```

2.4. What will be printed?

```
1 if "Berry" in flavors:
2   print("Available!")
3 else:
4   print("Out...")
```

2.5. What will be printed?

```
1 def buy(vm: dict[str,str])->str:
2   for thing in vm:
3     return thing
4   return "Other"
5
6 print(buy(vm=vend))
```

Memory Diagram

```python
def group_names(names: list[str]) -> dict[str, int]:
    groups: dict[str, int] = {}
    first_letter: str
    for n in names:
        first_letter = n[0]
        if first_letter in groups:
            groups[first_letter] += 1
        else:
            groups[first_letter] = 1
    return groups

ppl: list[str] = ["Karen", "Emily", "Kris"]
output: dict[str, int] = group_names(names=ppl)
print(output)
output["I"] = 1
print(output)
```