More on Sets and Dictionaries

Announcements

- **LS11 – Dictionaries** – due today at 11:59pm
- **EX04 – List Utility Functions** due Tues, March 3 at 11:59pm
- Quiz 01 regrade requests open till 11:59pm today

Limits of Lists for collections of data (1/2)

Using a list, we *could* store everyone in COMP110's PID associated with ONYEN

list[str]	
Index	Value
0	""
1	""
… 710,453,081 items elided …	
710453084	"krisj"
… 9,857,700 items elided …	
720310785	"abyrnes1"
… 9,809,924 items elided …	
730120710	"ihinks"

Warm-up question:
Why does using a **`list[str]`** feel wrong/inefficient?

Limits of Lists for collections of data (2/2)

onyens:

list[str]	
Index	Value
0	"ihinks"
1	"abyrnes1"
2	"sjiang3"
... 296 items elided ...	
299	"krisj"

pids:

list[int]	
Index	Value
0	730120710
1	720310785
2	730820837
... 296 items elided ...	
299	710453084

Suppose we model ONYENs and PIDs with lists. One list has ONYENs, the other has the person's PID at the same index.

Given the onyen "sjiang3", how do you algorithmically find their PID?

We could use the `in` operator (a new concept)…

```
1  # Pretend we initialized pids to hold all of our PIDs
2  pids: list[int] = [700000000, 700000001, 700000002, ..., 710453084, 730120710]
3  pids_of_interest: list[int] = [710453084, 730120710]
4  idx: int = 0
5  while idx < len(pids_of_interest):
6      if pids_of_interest[idx] in pids:
7          print("We found a PID in the list!")
8      idx += 1
```

… but try to avoid using it on lists!

Enter: 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}
```

Great! … But what if we want to associate people’s PIDs with their ONYENs in a data structure?

Enter: Dictionaries!

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

Unlike lists, dictionaries give *you* the ability to decide what to *index* your data by.

- Lists: *always* zero-based, sequential, integer indices!

Dictionaries are indexed by ***keys*** associated with ***values***. *This is a unique, one-way mapping!*

- Analogous: A real-world dictionary's ***keys*** are *words* and associated ***values*** are *definitions.*

`pid_to_onyen:`

`dict[int, str]`	
`key`	`value`
`730120710`	`"ihinks"`
`710453084`	`"krisj"`
`720310785`	`"abyrnes1"`

`onyen_to_seat:`

`dict[str, str]`	
`key`	`value`
`"ihinks"`	`"A1"`
`"abyrnes1"`	`"A2"`
`"sjiang3"`	`"A3"`
`"krisj"`	`"N17"`

Let’s diagram key concepts

```python
# USD exchange rate to other currencies
exchange: dict[str, float] = {
    "CNY": 7.10,  # Chinese Yuan
    "GBP": 0.77,  # British Pound
    "DKK": 6.86,  # Danish Kroner
}

dollars: float = 100.0

# Access dictionary value by its key
pounds: float = dollars * exchange["GBP"]

# Append a key-value entry to dictionary
exchange["EUR"] = 0.92

# Update a key-value entry in dictionary
exchange["CNY"] -= 1.00

# len is the number of key-value entries
count: int = len(exchange)
```

Let’s explore Dictionary syntax in VSCode together…

In your cl directory, add a file named dictionaries.py with the following starter:

```
"""Examples of dictionary syntax with Ice Cream Shop order tallies."""

ice_cream: dict[str, int] = {
  "chocolate": 12,
  "vanilla": 8,
  "strawberry": 4,
}
```

Save, then open up this file in Trailhead's REPL and we will explore key syntax together.

Ready to go? Try evaluating the following expression:

```
ice_cream["vanilla"] += 110
```

Syntax

Data type:

```
name: dict[<key type>, <value type>]
temps: dict[str, float]
```

Construct an empty dict:

```
temps: dict[str, float] = dict() or
temps: dict[str, float] = {}
```

Construct a populated dict:

```
temps: dict[str, float] = {"Florida": 72.5, "Raleigh": 56.0}
```

Let's try it!

Create a dictionary called ice_cream that stores the following orders

Keys	Values
chocolate	12
vanilla	8
strawberry	5

Length of dictionary

len(<dict name>)

len(temps)

Let's try it!

Print out the length of ice_cream.

What exactly is this telling you?

Adding elements

We use subscription notation.**

<dict name>[<key>] = <value>

temps["DC"] = 52.1

Note that we ***do not call a method* to add a key-value pair to a dictionary!

Let's try it!

Add 3 orders of "mint" to your ice_cream dictionary.

Access + Modify

To access a value,
use subscription notation:

```
<dict name>[<key>]
temps["DC"]
```

To modify, also use subscription notation:

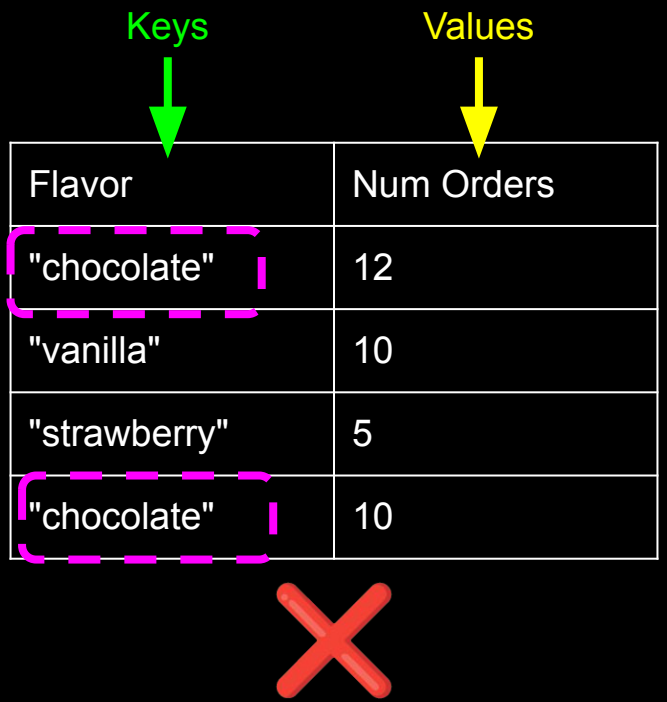
```
<dict name>[<key>] = new_value
temps["DC"] = 53.1  or  temps["DC"] += 1
```

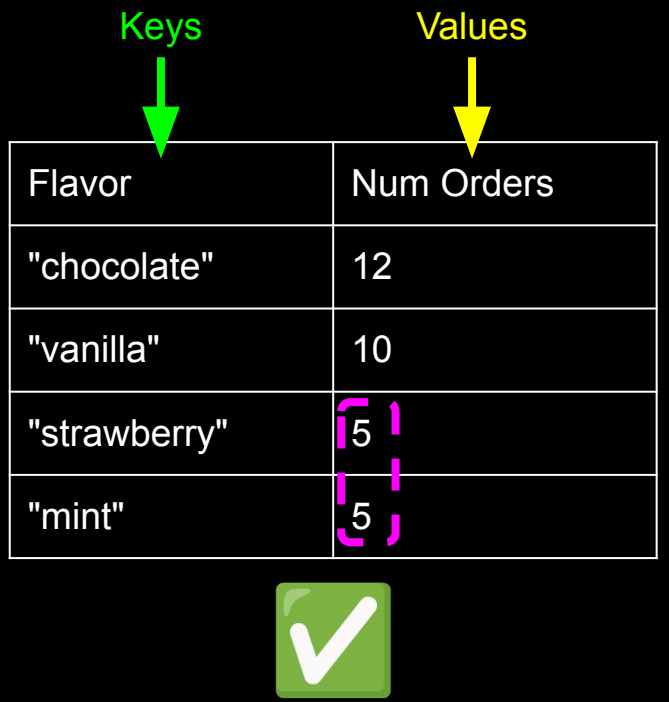
Let's try it!

Print out how many orders there are of "chocolate".
Update the number of orders of Vanilla to 10.

Important Note: Can't Have Multiple of Same ***Key***

(Duplicate *values* are okay.)





Check if key in dictionary

<key> in <dict name>

"DC" in temps

"Florida" in temps

Let's try it!

Check if both the flavors "mint" and "chocolate" are in ice_cream.

Write a conditional that behaves the following way:
If "mint" is in ice_cream, print out how many orders of "mint" there are.
If it's not, print "no orders of mint".

Removing elements from a dict

Similar to lists, we use pop()

<dict name>.pop(<key>)

temps.pop("Florida")

Let's try it!

Remove the orders of "strawberry" from ice_cream.

`for` Loops

`for` loops iterate over the *keys* by default

Let's try it!
Use a for loop to print:
chocolate has 12 orders.
vanilla has 10 orders.
strawberry has 5 orders.
(print statements for the key-value pairs currently in your dictionary)

```
for key in ice_cream:
    print(key)
```

```
for key in ice_cream:
    print(ice_cream[key])
```

Flavor	Num Orders
"chocolate"	12
"vanilla"	10
"strawberry"	5

This is the code we wrote together, for reference.

```python
"""Examples of dictionary syntax with Ice Cream Shop order tallies."""

# Dictionary type is dict[key_type, value_type].
# Dictionary literals are curly brackets
# that surround with key:value pairs.
ice_cream: dict[str, int] = {
    "chocolate": 12,
    "vanilla": 8,
    "strawberry": 4,
}

# len evaluates to number of key-value entries
print(f"{len(ice_cream)} flavors")

# Add key-value entries using subscription notation
ice_cream["mint"] = 3

# Access values by their key using subscription
print(ice_cream["chocolate"])

# Re-assign values by their key using assignment
ice_cream["vanilla"] += 10

# Remove items by key using the pop method
ice_cream.pop("strawberry")

# Loop through items using for-in loops
total_orders: int = 0
# The variable (e.g. flavor) iterates over
# each key one-by-one in the dictionary.
for flavor in ice_cream:
    print(f"{flavor}: {ice_cream[flavor]}")
    total_orders += ice_cream[flavor]

print(f"Total orders: {total_orders}")
```

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

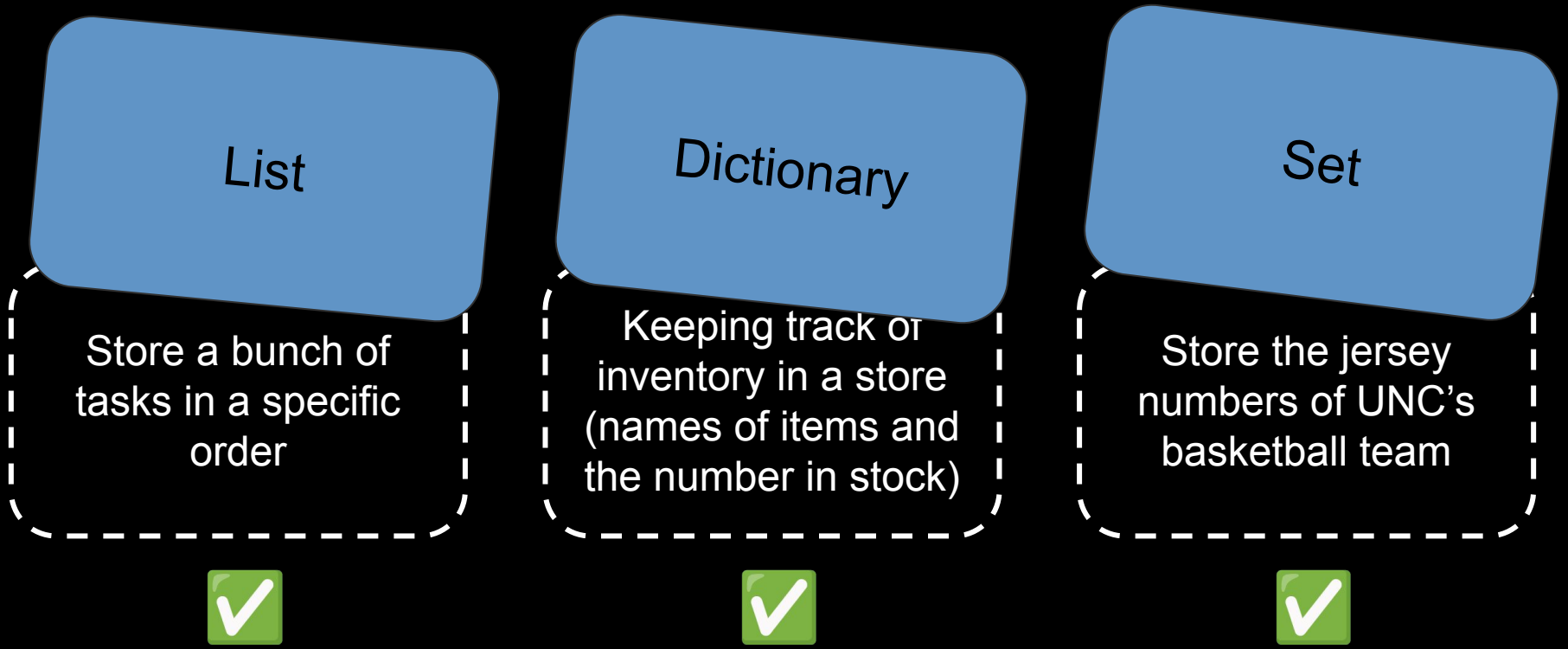
Dictionary

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)
```

Data structure	Empty literal	How to add an element	How to remove an element	How to access the number of elements
list [value1, …]				
set {value1, …}				
dictionary {key1: value1, …}				