



CL12: Sets and Dictionaries

Announcements

- LS10 – Dictionaries due today
- EX03 released today, due *Wednesday* (June 4)!
 - Autograder will be posted right after this lecture (by 2pm)
- Quiz 03 on Friday
 - Practice quiz + key posted tomorrow
 - Virtual Review Session on Thursday at 6:00pm

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

```
1  # USD exchange rate to other currencies
2  exchange: dict[str, float] = {
3      "CNY": 7.10, # Chinese Yuan
4      "GBP": 0.77, # British Pound
5      "DKK": 6.86, # Danish Kroner
6  }
7
8  dollars: float = 100.0
9
10 # Access dictionary value by its key
11 pounds: float = dollars * exchange["GBP"]
12
13 # Append a key-value entry to dictionary
14 exchange["EUR"] = 0.92
15
16 # Update a key-value entry in dictionary
17 exchange["CNY"] -= 1.00
18
19 # len is the number of key-value entries
20 count: int = len(exchange)
```


Let's explore Dictionary syntax in VSCode together...

In your cl directory, add a file named cl22_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`

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:

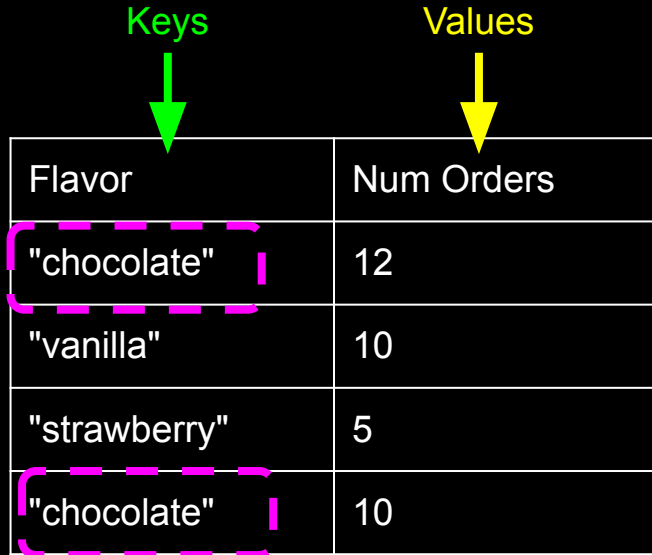
```
<dict name>[<key>] = new_value  
temps["DC"] = 53.1 or temps["DC"] += 1
```

Let's try it!

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

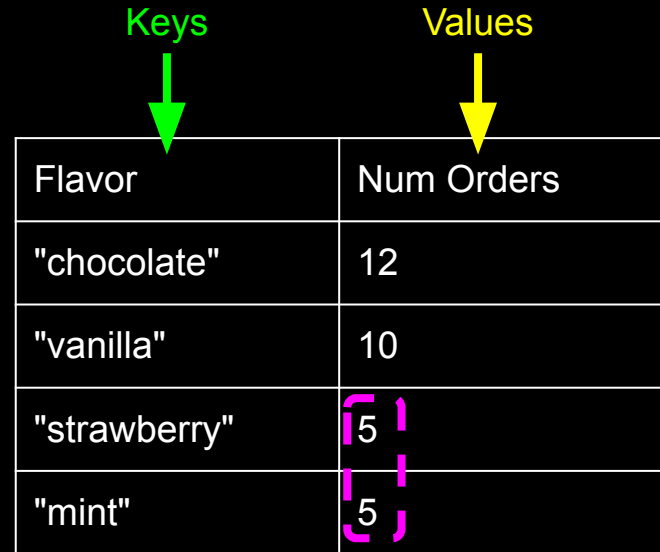
Important Note: Can't Have Multiple of Same Key

(Duplicate values are okay.)



A diagram illustrating a valid dictionary structure. A green arrow labeled 'Keys' points to the 'Flavor' column, and a yellow arrow labeled 'Values' points to the 'Num Orders' column. The table has four rows. The first row has 'chocolate' in the Flavor column and 12 in the Num Orders column. The second row has 'vanilla' in the Flavor column and 10 in the Num Orders column. The third row has 'strawberry' in the Flavor column and 5 in the Num Orders column. The fourth row has 'chocolate' in the Flavor column and 10 in the Num Orders column. The 'chocolate' keys in the first and fourth rows are highlighted with dashed red boxes, showing that duplicate keys are not allowed in a dictionary.

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



A diagram illustrating an invalid dictionary structure. A green arrow labeled 'Keys' points to the 'Flavor' column, and a yellow arrow labeled 'Values' points to the 'Num Orders' column. The table has five rows. The first row has 'chocolate' in the Flavor column and 12 in the Num Orders column. The second row has 'vanilla' in the Flavor column and 10 in the Num Orders column. The third row has 'strawberry' in the Flavor column and 5 in the Num Orders column. The fourth row has 'mint' in the Flavor column and 5 in the Num Orders column. The '5' values in the third and fourth rows are highlighted with dashed red boxes, showing that duplicate values are allowed in a dictionary.

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

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

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.

```
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.

```
1  """Examples of dictionary syntax with Ice Cream Shop order tallies."""
2
3  # Dictionary type is dict[key_type, value_type].
4  # Dictionary literals are curly brackets
5  # that surround with key:value pairs.
6  ice_cream: dict[str, int] = {
7      "chocolate": 12,
8      "vanilla": 8,
9      "strawberry": 4,
10 }
11
12 # len evaluates to number of key-value entries
13 print(f"{len(ice_cream)} flavors")
14
15 # Add key-value entries using subscription notation
16 ice_cream["mint"] = 3
17
18 # Access values by their key using subscription
19 print(ice_cream["chocolate"])
20
21 # Re-assign values by their key using assignment
22 ice_cream["vanilla"] += 10
23
24 # Remove items by key using the pop method
25 ice_cream.pop("strawberry")
26
27 # Loop through items using for-in loops
28 total_orders: int = 0
29 # The variable (e.g. flavor) iterates over
30 # each key one-by-one in the dictionary.
31 for flavor in ice_cream:
32     print(f"{flavor}: {ice_cream[flavor]}")
33     total_orders += ice_cream[flavor]
34
35 print(f"Total orders: {total_orders}")
```

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?

```
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))
```

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

```
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))
```

- 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 $N \cdot M$ operations to occur.

Comparing lists and sets

```
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         if a[idx_a] in b:
7             result.append(a[idx_a])
8         idx_a += 1
9
10    return result
```

```
1 def intersection(a: list[str], b: set[str]) -> set[str]:
2     result: set[str] = set()
3
4     idx_a: int = 0
5     while idx_a < len(a):
6         if a[idx_a] in b:
7             result.add(a[idx_a])
8         idx_a += 1
9
10    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.