

COMP
110

Dictionaries

Motivation

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

Why does using a `list[str]` feel wrong/inefficient?

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"

Other Approach:

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?

Other Approach: Dictionaries!

Dictionaries, like lists, are a *data structure*.

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

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*.

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

Dictionaries

Keys



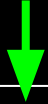
Values



Flavor	Num Orders
"chocolate"	12
"vanilla"	8
"strawberry"	5

Dictionaries

Keys



Values



Flavor	Num Orders
"chocolate"	12
"vanilla"	8
"strawberry"	5

Lists

Indexes



Values



0	12
1	8
2	5

Also called: Map, Hashmap, Key-Value Store

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

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.

Removing elements

Similar to lists, we use pop()

`<dict name>.pop(<key>)`

`temps.pop("Florida")`

Let's try it!

Remove the orders of "mint" from
ice_cream.

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.

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?

Check if key in dictionary

`<key> in <dict name>`

`"DC" in temps`

`"Florida" in temps`

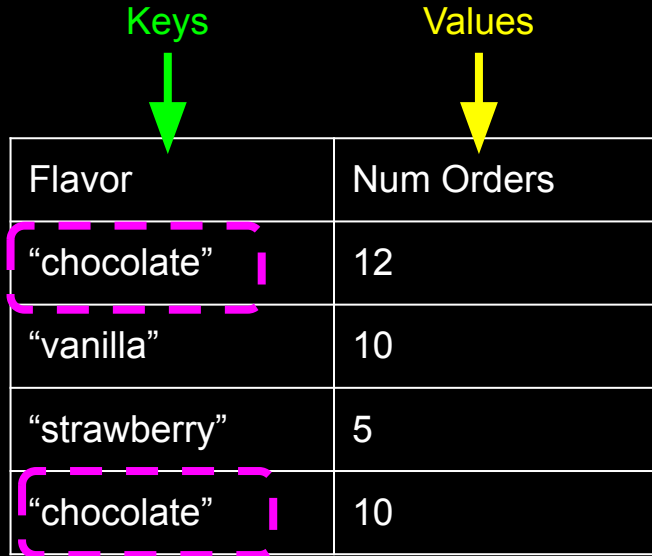
Let's try it!

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

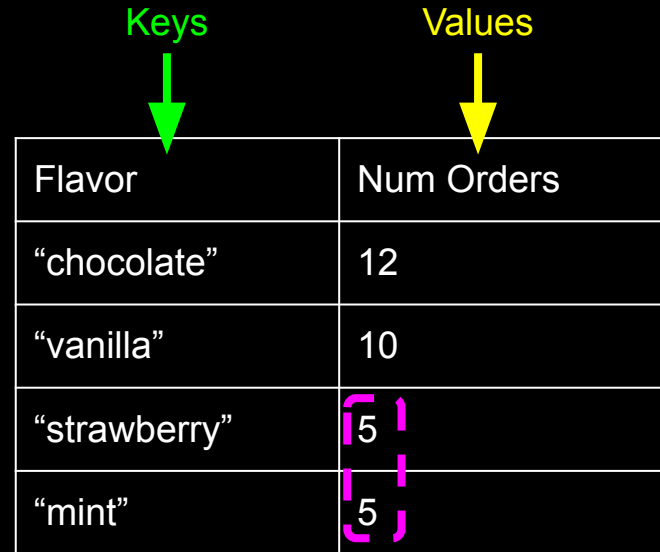
Write a conditional that behaves the following way:
If "mint" is in ice_cream, print out how many orders of "mint" there are.
If it's not, print "no orders of mint".

Important Note: Can't Have Multiple of Same Key

(Duplicate values are okay.)



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



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

In Memory

```
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)
```

Practice!

- Let's implement a function named `grade_bump` where we can call with 2 arguments:
 - A `gradebook: dict[str, str]` that stores letter grades (values) for every student name (key)
 - A `student: str` name of a student
- The `return value` of the function is `None`
- The function should *mutate* `gradebook` the following way:
 - If the grade of `student` is "A", it should change their grade to "A+"
 - Otherwise, it should change their grade to "A"
- Example:

```
grades: dict[str, str] = {"alyssa": "A", "luke": "B"}
```

```
grade_bump(grades, "luke")
```

```
print(grades)
```

```
>>> {"alyssa": "A", "luke": "A"}
```