CSCI 145 · Introduction to Computing · Fall 2026

Tuples and Dictionaries

Lecture 12 — grouping values and looking them up by name

Outline

- Tuples
- Dictionaries
- Dictionary patterns

Part 1 of 3

Tuples

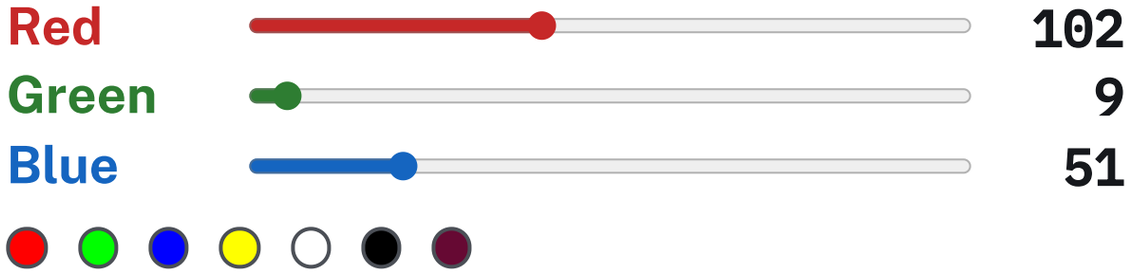
> A **tuple** groups a fixed number of related values
> such as the red, green, and blue amounts of a color
> or the `(x, y)` coordinates of a point

How a screen makes a color



(102, 9, 51)

#660933



- Each pixel mixes red, green, and blue light
- Each amount is a whole number from 0 (none) to 255 (full)
- Three numbers that go together, always in the same order

Same code in hexadecimal (coming in Lecture 15 and Notes 13)

Some tuples

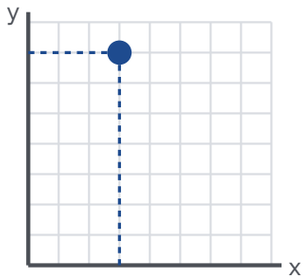
Written in parentheses, separated by commas

```
burgundy = (102, 9, 51)
```



A color: red, green, blue

```
point = (3, 7)
```



A location: x, y

```
student = ("Ana", 92)
```



A name and a score

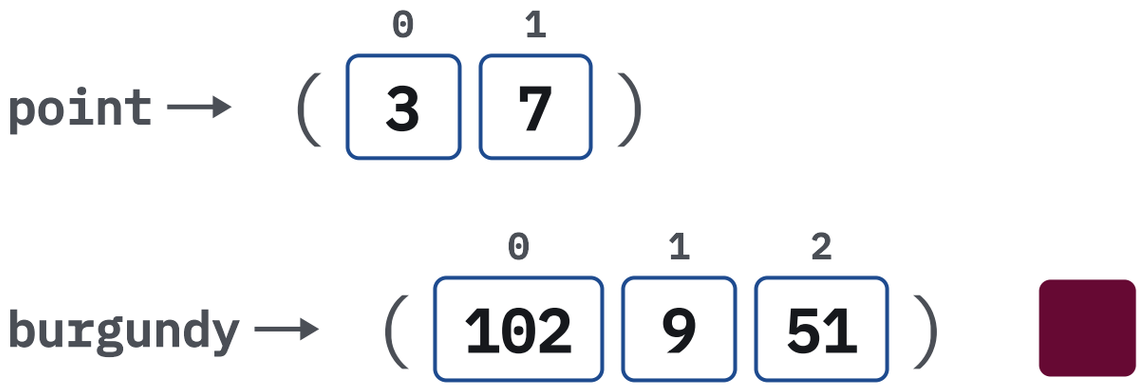
- Each position has its own meaning: `burgundy[0]` is the red amount

Using a tuple

```
point = (3, 7)
print(point[0])
print(len(point))
print(7 in point)
burgundy = (102, 9, 51)
print(burgundy[1:])
```

OUTPUT

```
3
2
True
(9, 51)
```



- Indexing, slicing, `len`, and `in` work as for strings and lists

A tuple can't change

LIST

```
nums = [3, 7]
nums[0] = 5
print(nums)
```

OUTPUT

```
[5, 7]
```

Lists are mutable

TUPLE

```
point = (3, 7)
point[0] = 5
print(point)
```

OUTPUT

```
TypeError: 'tuple' object
does not support
item assignment
```

Tuples are **immutable**, like strings

A NEW TUPLE

```
point = (3, 7)
point = (5, 7)
print(point)
```

OUTPUT

```
(5, 7)
```

Naming a new tuple is fine

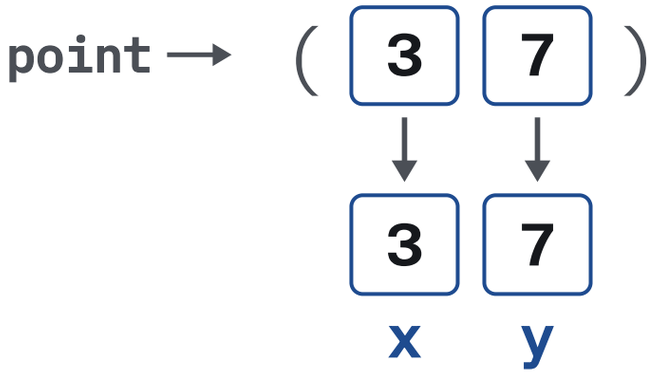
- A list holds many items, often of the same kind
  A tuple holds a few related values: `(x, y)`, `(red, green, blue)`
- A tuple with only one value needs a comma: `(5,)`

Packing and unpacking

```
point = 3, 7
print(point)
x, y = point
print(x)
print(y)
x, y, z = point
```

OUTPUT

```
(3, 7)
3
7
ValueError: not enough values
to unpack (expected 3, got 2)
```



- Commas make a tuple: **packing**
  The parentheses are optional
- One name per value: **unpacking**
- The number of names must match the number of values

Swapping two variables

On paper: what does the first program print?

FIRST TRY

```
a = 1
b = 2
a = b
b = a
print(a, b)
```

OUTPUT

```
2 2
```

`a = b` replaced the 1 before `b` could get it

WITH A TEMPORARY NAME

```
a = 1
b = 2
temp = a
a = b
b = temp
print(a, b)
```

OUTPUT

```
2 1
```

`temp` holds on to the 1

TUPLE SWAP

```
a = 1
b = 2
a, b = b, a
print(a, b)
```

OUTPUT

```
2 1
```

Python packs `b, a` first then unpacks into `a, b`

Returning two values

```
def share(cookies, people):
    each = cookies // people
    extra = cookies % people
    return each, extra

result = share(17, 5)
print(result)
per_person, left_over = share(17, 5)
print(per_person, left_over)
```

OUTPUT

```
(3, 2)
3 2
```

INSIDE THE CALL



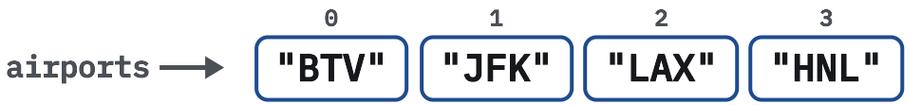
- `return each, extra` packs a tuple
- The caller can unpack it right away
- Today's activity: `min_max(nums)`

Building a list of tuples

```
def legs(airports):
    trip = []
    for i in range(len(airports) - 1):
        trip.append((airports[i], airports[i + 1]))
    return trip

print(legs(["BTV", "JFK", "LAX", "HNL"]))
```

- The inner parentheses make one tuple
  The outer ones belong to `append`
- Homework 4: `hanoi` returns
  a list of moves like this



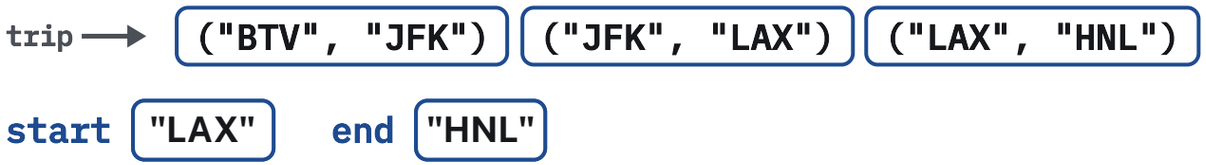


OUTPUT

```
[('BTV', 'JFK'), ('JFK', 'LAX'), ('LAX', 'HNL')]
```

Looping over a list of tuples

```
trip = [("BTV", "JFK"), ("JFK", "LAX"), ("LAX", "HNL")]
for start, end in trip:
    print("Fly from", start, "to", end)
```



OUTPUT

```
Fly from BTV to JFK
Fly from JFK to LAX
Fly from LAX to HNL
```

- Each item is a tuple
- `start, end` unpacks it

THE SAME LOOP IN TWO STEPS

```
for leg in trip:
    start, end = leg
    print("Fly from", start, "to", end)
```

Part 2 of 3

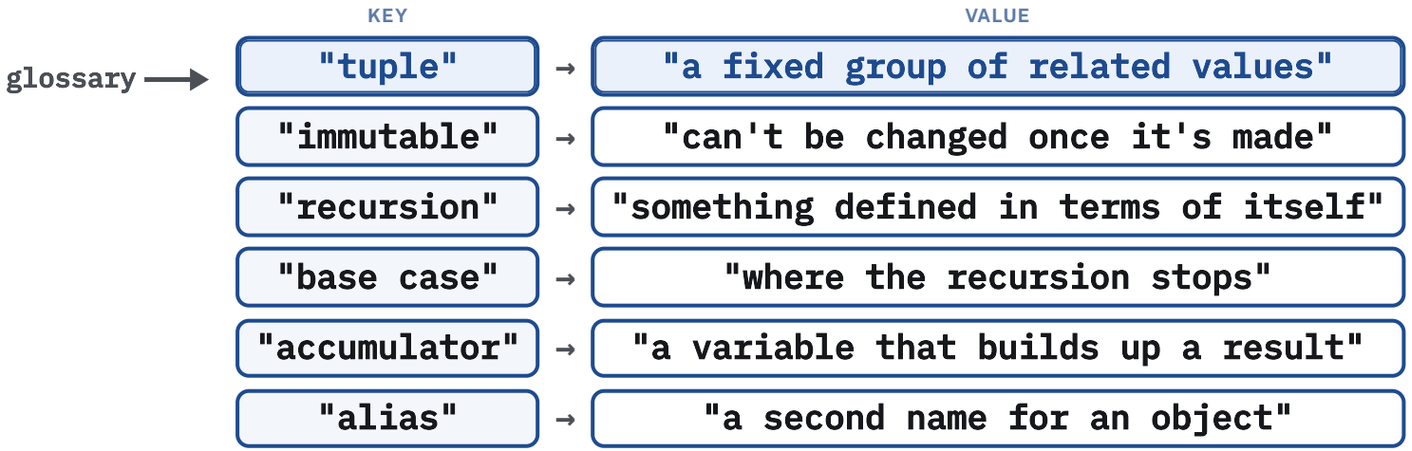
Dictionaries

A **dictionary** is a collection of key-value pairs
like the words and definitions in a language dictionary
You look up a value by its key

Look it up

Call out a word from this course

```
>>> glossary["tuple"]
'a fixed group of related values'
```



- Find a value by its key
  No searching by position
- A word that isn't a key: `KeyError`

Looking things up by name

A LIST OF TUPLES

```
scores = [("Ana", 92), ("Ben", 80), ("Cy", 89)]
for name, score in scores:
    if name == "Ben":
        print(score)
```

To find Ben's score,
check each tuple

A DICTIONARY

```
scores = {"Ana": 92, "Ben": 80, "Cy": 89}
print(scores["Ben"])
```

Ask for Ben's score directly

- A list finds an item by its position
  A **dictionary** finds a value by its key

EVERYDAY DICTIONARIES

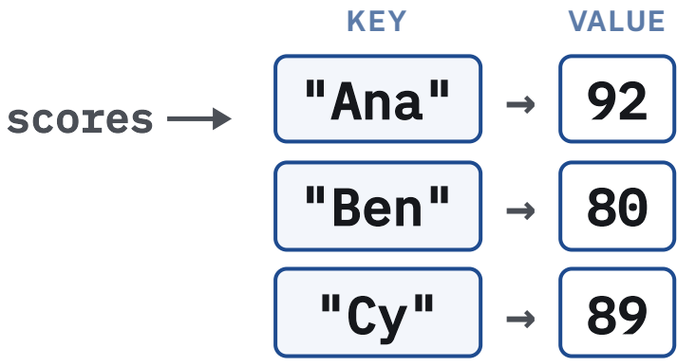
Contacts
name → phone number

Menu
menu item → price

Language dictionary
word → definition

A dictionary

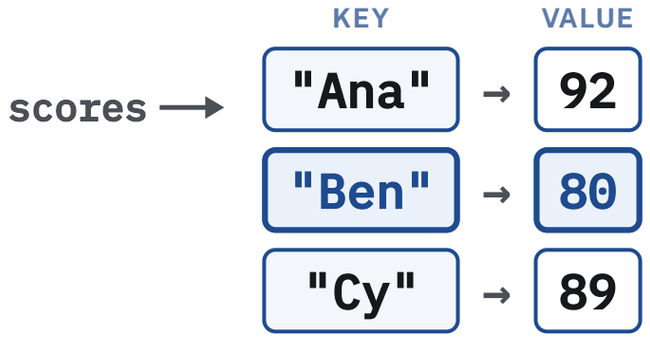
```
scores = {"Ana": 92, "Ben": 80, "Cy": 89}
```



- The **keys**: `"Ana"`, `"Ben"`, `"Cy"`
- Each key maps to a **value**
- A `key: value` pair is an **item**
- Keys are unique
  Values can repeat
- `{}` is an empty dictionary

Looking up a value

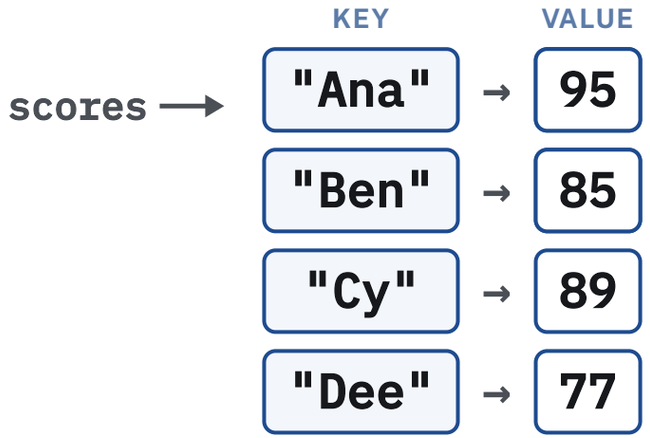
```
>>> scores = {"Ana": 92, "Ben": 80, "Cy": 89}
>>> scores["Ben"]
80
>>> scores["Dee"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Dee'
>>> "Dee" in scores
False
>>> 80 in scores
False
>>> scores.get("Dee", 0)
0
>>> scores.get("Ben", 0)
80
```



- A missing key is an error
- `in` checks the keys, not the values
- `.get(key, default)` gives the default for a missing key

Changing and adding

```
>>> scores = {"Ana": 92, "Ben": 80, "Cy": 89}
>>> scores["Ana"] = 95
>>> scores["Dee"] = 77
>>> scores["Ben"] += 5
>>> scores
{'Ana': 95, 'Ben': 85, 'Cy': 89, 'Dee': 77}
>>> len(scores)
4
```



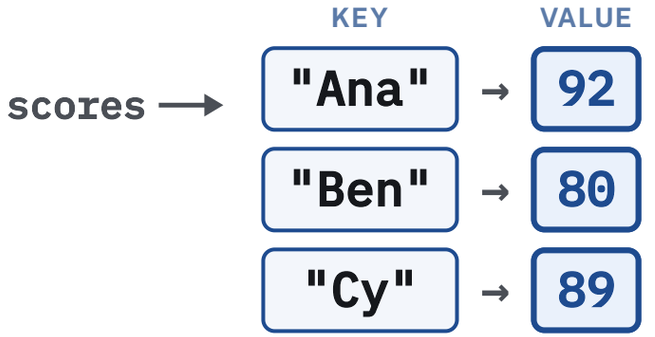
- An existing key: its value changes
- A new key: a new item
- Dictionaries are mutable, like lists
- A list can't grow this way
  `nums[5] = 1` is an IndexError

Looping over a dictionary

```
scores = {"Ana": 92, "Ben": 80, "Cy": 89}
for name in scores:
    print(name)
for name, score in scores.items():
    print(name, score)
print(sum(scores.values()))
```

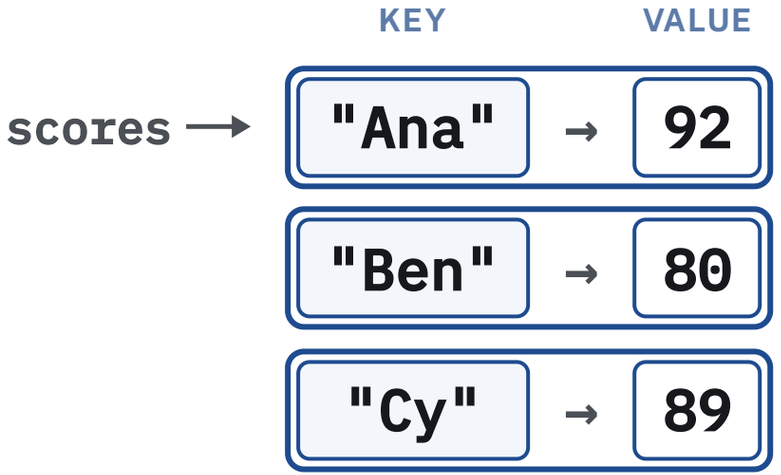
OUTPUT

```
Ana
Ben
Cy
Ana 92
Ben 80
Cy 89
261
```



- A loop over a dictionary gets its keys
- `.items()` gives `(key, value)` tuples Unpack them like the trip
- `.values()` gives just the values

Keys, values, and items




```
>>> scores = {"Ana": 92, "Ben": 80, "Cy": 89}
>>> scores.keys()
dict_keys(['Ana', 'Ben', 'Cy'])
>>> scores.values()
dict_values([92, 80, 89])
>>> scores.items()
dict_items([('Ana', 92), ('Ben', 80), ('Cy', 89)])
```

- `.keys()` : just the keys
- `.values()` : just the values
- `.items()` : each item as a `(key, value)` tuple
- Not lists: you can loop over them but not index them
  `list(scores.values())` makes a list

What can be a key?

```
>>> board = {(0, 0): "start", (2, 1): "gold"}
>>> board[(2, 1)]
'gold'
>>> bad = {[0, 0]: "start"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

KEYS ARE (ROW, COLUMN)

start	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	**gold**	(2, 2)	(2, 3)

- **Keys** must be **immutable**
  Strings, numbers, tuples
- A list can change,
  so it can't be a **key**
- **Values** can be anything,
  even lists

Which would you use?

	List	Tuple	Dictionary
Looks like	`[3, 7]`	`(3, 7)`	`{"Ana": 92}`
Find an item by	Position	Position	Key
Can change?	Yes	No	Yes

One location: (latitude, longitude) **Tuple**

Everyone's quiz scores, to average them **List**

Each student's email, by name **Dictionary**

How many times each word appears **Dictionary**

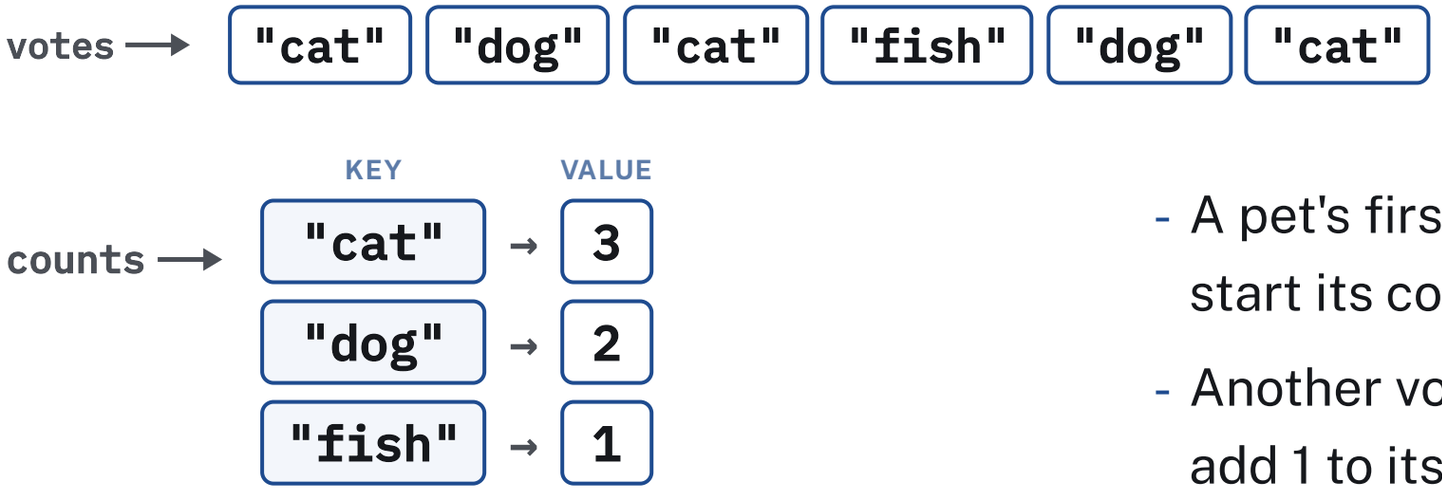
Part 3 of 3

Dictionary patterns

Tally the votes

Favorite pet, one vote per person

By inspection: how many votes did each pet get?



- A pet's first vote:
  start its count at 1
- Another vote:
  add 1 to its count

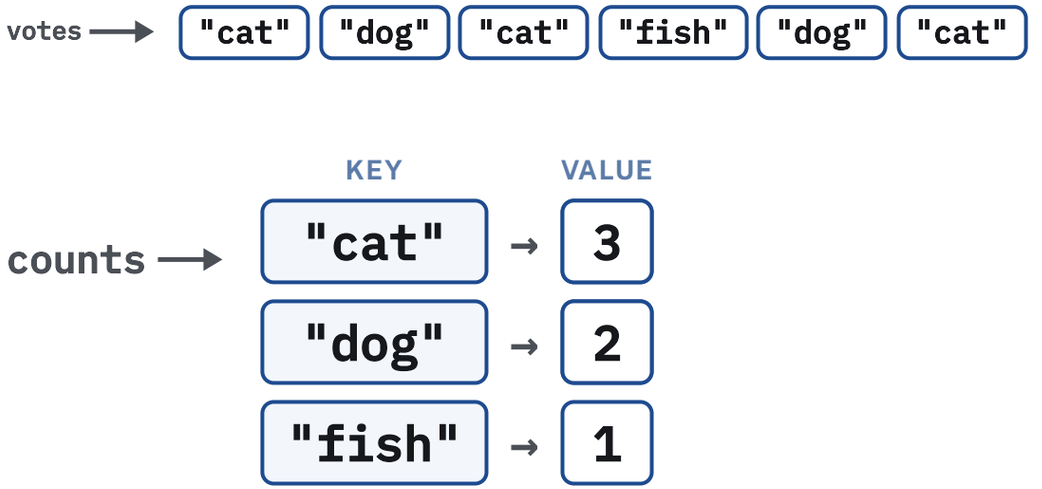
Counting with a dictionary

```
def count_votes(votes):
    counts = {}
    for vote in votes:
        if vote in counts:
            counts[vote] += 1
        else:
            counts[vote] = 1
    return counts

print(count_votes(["cat", "dog", "cat",
                   "fish", "dog", "cat"]))
```

OUTPUT

```
{'cat': 3, 'dog': 2, 'fish': 1}
```



- First time: start at 1
- Again: add 1

How do we fix this code?

```
def count_votes(votes):
    counts = {}
    for vote in votes:
        counts[vote] += 1
    return counts

print(count_votes(["cat", "dog", "cat"]))
```

OUTPUT

```
KeyError: 'cat'
```

REPLACE THE LINE IN THE LOOP WITH

A

```
counts[vote] = 1
```

B

```
counts[vote] = counts[vote] + 1
```

C

```
counts[vote] = counts.get(vote, 0) + 1
```

D

```
counts[vote] = 0
counts[vote] += 1
```

Go to **slido.com** and enter the code: 3674841

Or join directly here

How do we fix this code?

```
def count_votes(votes):
    counts = {}
    for vote in votes:
        counts[vote] += 1
    return counts

print(count_votes(["cat", "dog", "cat"]))
```

OUTPUT

```
KeyError: 'cat'
```

REPLACE THE LINE IN THE LOOP WITH

A

```
counts[vote] = 1
```

B

```
counts[vote] = counts[vote] + 1
```

C

```
counts[vote] = counts.get(vote, 0) + 1
```

D

```
counts[vote] = 0
counts[vote] += 1
```

Answer: C

`counts.get(vote, 0)` is 0 the first time
Then the old count after that

Why not the others?

A and D: every count ends up 1
B: the same `KeyError`

A dictionary as a lookup table

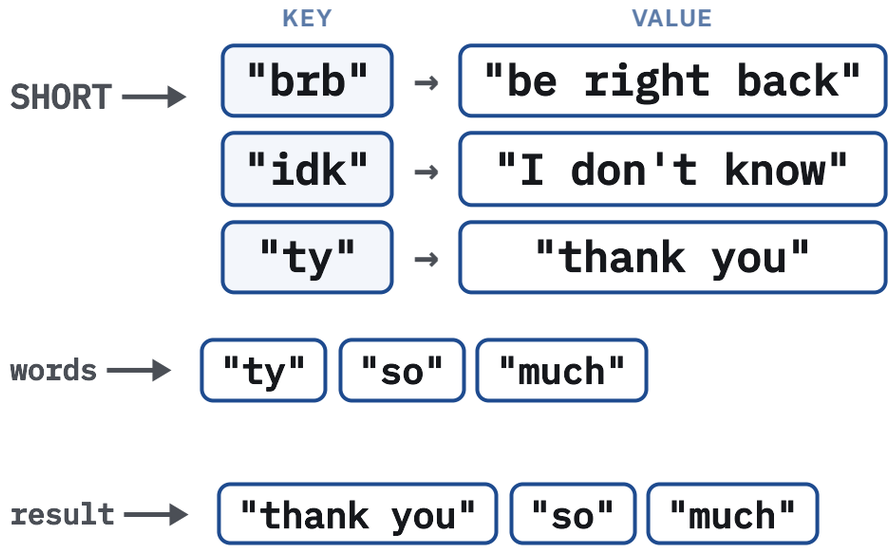
```
SHORT = {"brb": "be right back",
         "idk": "I don't know",
         "ty": "thank you"}

def expand(words):
    result = []
    for word in words:
        result.append(SHORT.get(word, word))
    return result

print(expand(["ty", "so", "much"]))
```

OUTPUT

```
['thank you', 'so', 'much']
```



- A key: use its value
- `so` isn't a key
  `.get(word, word)` keeps the word

Back to `fib`: remember the answers

LECTURE 11

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

WITH A DICTIONARY

```
memo = {0: 0, 1: 1}

def fib(n):
    if n not in memo:
        memo[n] = fib(n - 1) + fib(n - 2)
    return memo[n]
```

- `memo` starts with the base cases
- Each `fib(n)` is computed once, then looked up
- `fib(30)`: 2,692,537 calls before, 59 calls now
- Saving answers to reuse them is called **memoization**

  `fib` and `memo_fib` are in the demo file

Activity

1. Download day12.zip.
2. Drag `day12.zip` to your `class` folder and extract it there.
   Mac — double-click it. **Windows** — right-click it and choose "Extract All...".
   Once you see the `day12` folder, delete `day12.zip`.
3. Launch Thonny. From the File menu, open
   `cs145/class/day12/day12_tuples_dicts.py`.
   The code from today's slides is in `day12_demos.py`.
4. Write `min_max`, `letter_counts`, and `total_cost`.
5. If you have time, write `most_common`.

To do

- Review this slide deck and sections 1–2 of Notes 11
- Exam 2 (Lectures 5–9) is Friday 10/23 in lab
- Quiz 6 (Lectures 11–12) is at the start of next class

See practice materials

- Homework 4 is due Thursday 10/29

Next time

Lecture 13 — sets and references