DICTIONARIES

(download slides and .py files to follow along)

6.100L Lecture 14

Ana Bell

HOW TO STORE STUDENT INFO

- Suppose we want to store and use grade information for a set of students
- Could store using separate lists for each kind of information

```
names =   ['Ana', 'John', 'Matt', 'Katy']
grades = ['A+' ,  'B'  ,   'A'  ,   'A'  ]
microquizzes = ...
psets = ...
```

- Info stored across lists at **same index**, each index refers to information for a different person
- Indirectly access information by finding location in lists corresponding to a person, then extract

HOW TO ACCESS STUDENT INFO

```
def get_grade(student, name_list, grade_list):
    i = name_list.index(student)
    grade = grade_list[i]
    return (student, grade)
```

find location in list for person

Use location index to access other info

- **Messy** if have a lot of different info of which to keep track, e.g., a separate list for microquiz scores, for pset scores, etc.
- Must maintain **many lists** and pass them as arguments
- Must **always index** using integers
- Must remember to change multiple lists, when adding or updating information

HOW TO STORE AND ACCESS STUDENT INFO

- Alternative might be to use a list of lists

```
eric = ['eric', ['ps', [8, 4, 5]], ['mq', [6, 7]]]
ana = ['ana', ['ps', [10, 10, 10]], ['mq', [9, 10]]]
john = ['john', ['ps', [7, 6, 5]], ['mq', [8, 5]]]

grades = [eric, ana, john]
```

- Then could access by searching lists, but code is still messy

```
def get_grades(who, what, data):
    for stud in data:
        if stud[0] == who:
            for info in stud[1:]:
                if info[0] == what:
                    return who, info

print(get_grades('eric', 'mq', grades))
print(get_grades('ana', 'ps', grades))
```

But idea of associating data with names is worth exploring

A BETTER AND CLEANER WAY – A DICTIONARY

- Nice to use **one data structure**, no separate lists
- Nice to **index item of interest directly**
- A Python **dictionary has entries** that map a key:value

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index element

BIG IDEA

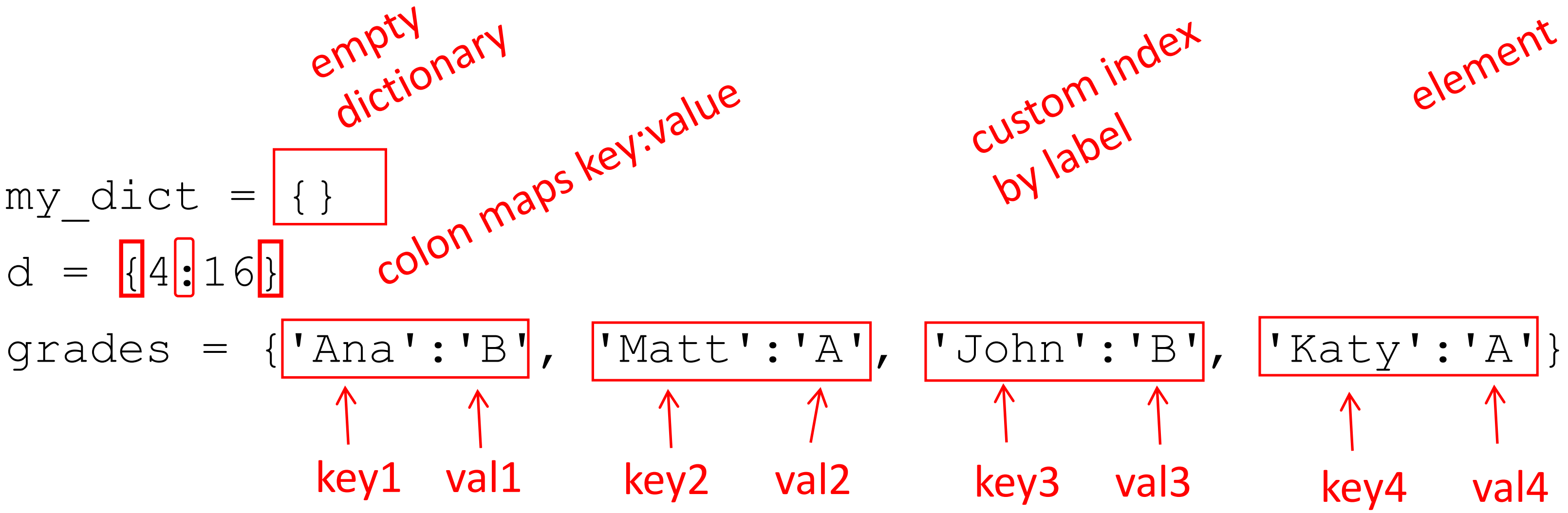
Dict value refers to the value associated with a key.

This terminology is may sometimes be confused with the regular value of some variable.

A PYTHON DICTIONARY

- Store **pairs of data** as an **entry**
  - key (any immutable object)
    - str, int, float, bool, tuple, etc
  - value (any data object)
    - Any above plus lists and other dicts!

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'



```
my_dict = {}
d = {4:16}
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

empty dictionary

colon maps key:value

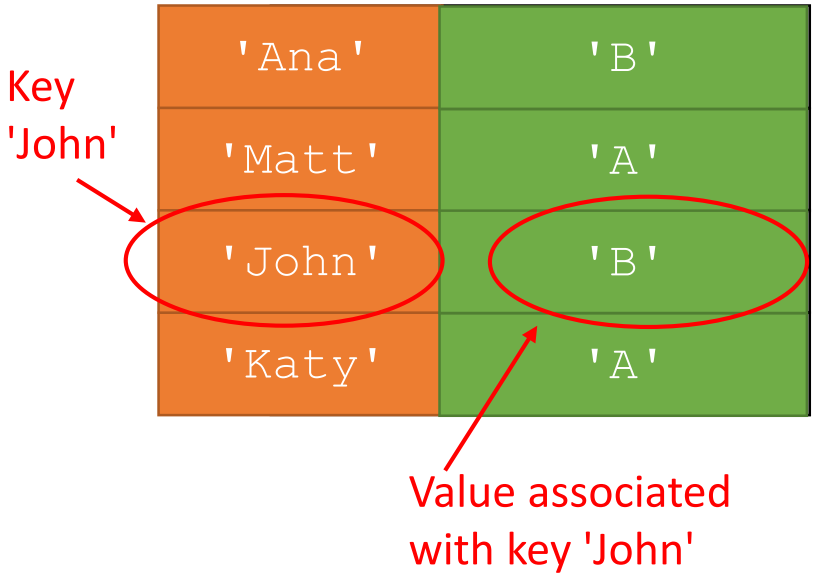
custom index by label

element

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- Similar to indexing into a list
- **Looks up** the **key**
- **Returns** the **value** associated with the key
  - If key isn't found, get an error
- There is **no simple expression to get a key back given some value**!



'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

Key 'John'

Value associated with key 'John'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

`grades['John']` → evaluates to `'B'`

`grades['Grace']` → gives a `KeyError`

YOU TRY IT!

- Write a function according to this spec

```
def find_grades(grades, students):
    """ grades is a dict mapping student names (str) to grades (str)
        students is a list of student names
    Returns a list containing the grades for students (in same order) """

# for example

d = {'Ana':'B', 'Matt':'C', 'John':'B', 'Katy':'A'}
print(find_grades(d, ['Matt', 'Katy'])) # returns ['C', 'A']
```

BIG IDEA

Getting a dict value is just a matter of indexing with a key.

No. Need. To. Loop

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'
'Grace'	'C'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Add** an entry

```
grades['Grace'] = 'A'
```

An assignment statement, but to a location in a dictionary – different from a list

- **Change** entry

```
grades['Grace'] = 'C'
```

- **Delete** entry

```
del(grades['Ana'])
```

Note that the dictionary is being mutated!

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- **Test** if key in dictionary

```
'John' in grades         →  returns True
'Daniel' in grades       →  returns False
'B' in grades            →  returns False
```

The in keyword only checks keys, not values

YOU TRY IT!

- Write a function according to these specs

```
def find_in_L(Ld, k):
    """ Ld is a list of dicts
        k is an int
    Returns True if k is a key in any dicts of Ld and False otherwise """

# for example
d1 = {1:2, 3:4, 5:6}
d2 = {2:4, 4:6}
d3 = {1:1, 3:9, 4:16, 5:25}

print(find_in_L([d1, d2, d3], 2)  # returns True
print(find_in_L([d1, d2, d3], 25)  # returns False
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all keys**

`grades.keys()` → returns `dict_keys(['Ana', 'Matt', 'John', 'Katy'])`

`list(grades.keys())` → returns `['Ana', 'Matt', 'John', 'Katy']`

- Get an **iterable that acts like a tuple of all dict values**

`grades.values()` → returns `dict_values(['B', 'A', 'B', 'A'])`

`list(grades.values())` → returns `['B', 'A', 'B', 'A']`

DICTIONARY OPERATIONS

most useful way to iterate over dict entries (both keys and vals!)

'Ana'	'B'
'Matt'	'A'
'John'	'B'
'Katy'	'A'

- Can iterate over dictionaries but assume there is no guaranteed order

```
grades = {'Ana':'B', 'Matt':'A', 'John':'B', 'Katy':'A'}
```

- Get an **iterable that acts like a tuple of all items**

```
grades.items()
```

→ returns `dict_items([('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')])`

```
list(grades.items())
```

→ returns `[('Ana', 'B'), ('Matt', 'A'), ('John', 'B'), ('Katy', 'A')]`

- Typical use is to **iterate over key,value tuple**

```
for k,v in grades.items():
    print(f"key {k} has value {v}")
```

```
key Ana has value B
key Matt has value A
key John has value B
key Katy has value A
```

YOU TRY IT!

- Write a function that meets this spec

```
def count_matches(d):
    """ d is a dict
    Returns how many entries in d have the key equal to its value """


# for example
d = {1:2, 3:4, 5:6}
print(count_matches(d))   # prints 0
d = {1:2, 'a':'a', 5:5}
print(count_matches(d))   # prints 2
```

DICTIONARY KEYS & VALUES

- Dictionaries are **mutable** objects (aliasing/cloning rules apply)
  - Use = sign to make an alias
  - Use d.copy() to make a copy
- **Assume there is no order** to keys or values!
- Dict values
  - Any type (**immutable and mutable**)
    - Dictionary values can be lists, even other dictionaries!
  - Can be **duplicates**
- Keys
  - Must be **unique**
  - **Immutable** type (`int`, `float`, `string`, `tuple`,`bool`)
    - Actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - Be careful using `float` type as a key

WHY IMMUTABLE/HASHABLE KEYS?

- A dictionary is stored in memory in a special way
- Next slides show an example

- Step 1: A **function is run on the dict key**
  - The function **maps any object to an int**
    E.g. map “a” to 1, “b” to 2, etc, so “ab” could map to 3
  - The int corresponds to a position in a block of memory addresses
- Step 2: At that memory address, **store the dict value**
- To do a **lookup** using a key, **run the same function**
  - If the object is immutable/hashable then you get the same int back
  - If the object is changed then the function gives back a different int!

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

1 + 14 + 1 = 16
16%16 = 0

Ana	C

5 + 18 + 9 + 3 = 35
35%16 = 3

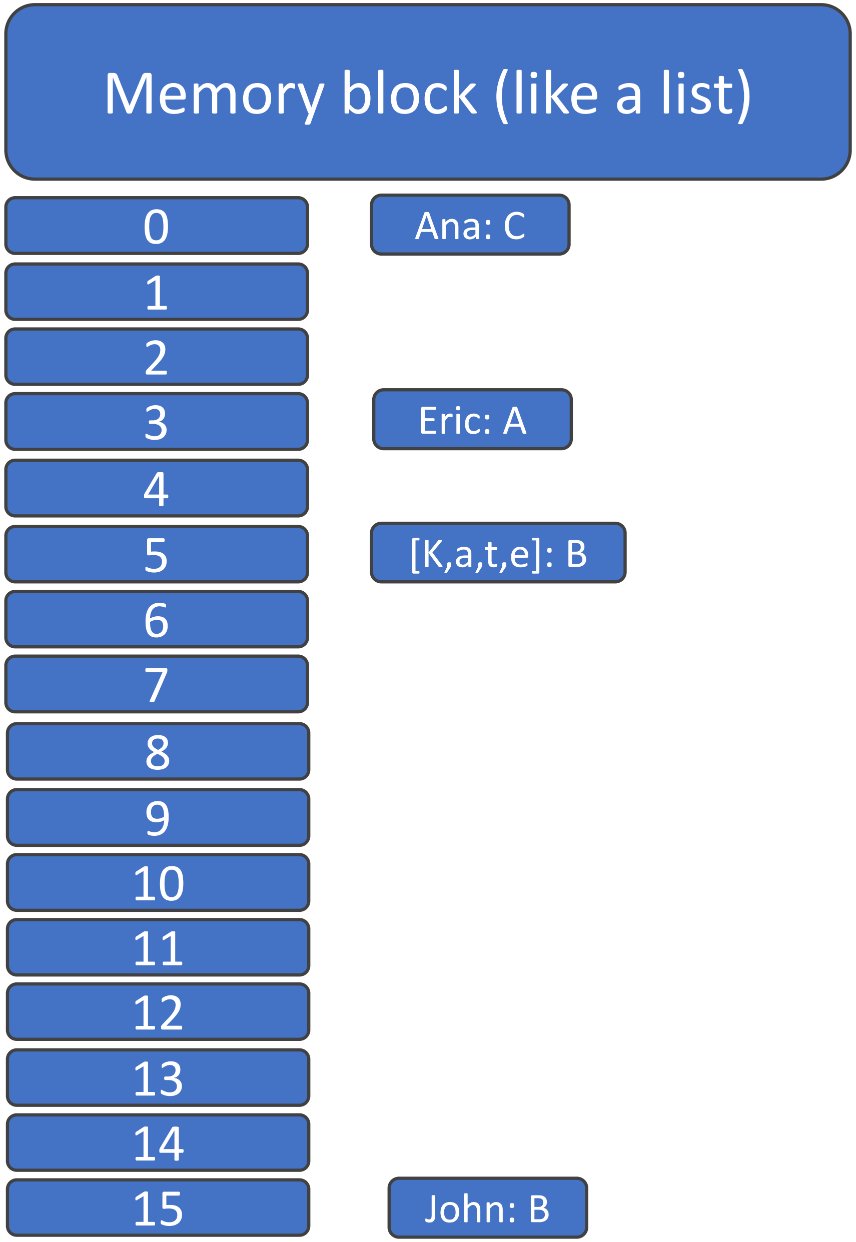
Eric	A

10 + 15 + 8 + 14 = 47
47%16 = 15

John	B

11 + 1 + 20 + 5 = 37
37%16 = 5

[K, a, t, e]	B



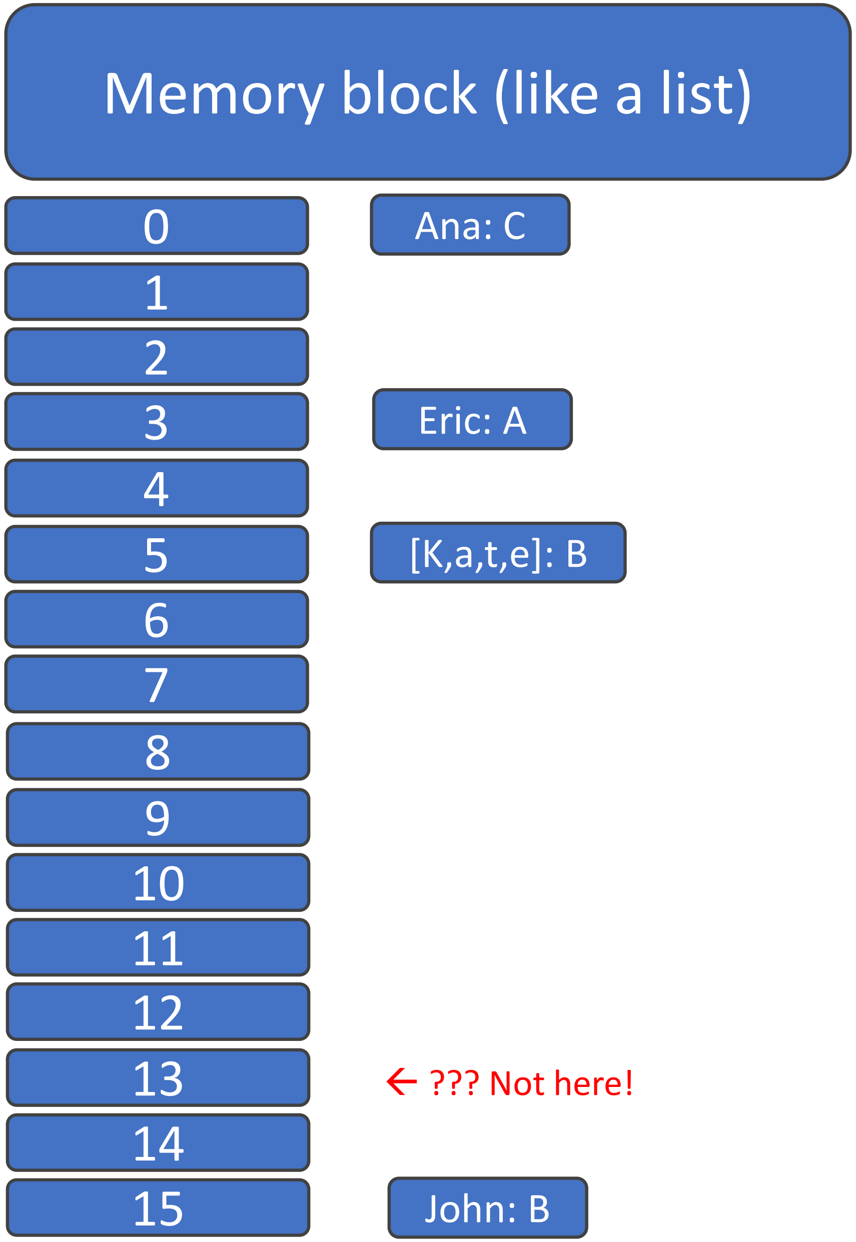
Memory block (like a list)

0	Ana: C
1	
2	
3	Eric: A
4	
5	[K,a,t,e]: B
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	John: B

Hash function:

1) Sum the letters
2) Take mod 16 (to fit in a memory block with 16 entries)

Kate changes her name to Cate. Same person, different name. Look up her grade?



Memory block (like a list)

0 Ana: C

1

2

3 Eric: A

4

5 [K,a,t,e]: B

6

7

8

9

10

11

12

13 ← ??? Not here!

14

15 John: B

3 + 1 + 20 + 5 = 29

29%16 = 13

[C, a, t, e]

A PYTHON DICTIONARY for STUDENT GRADES

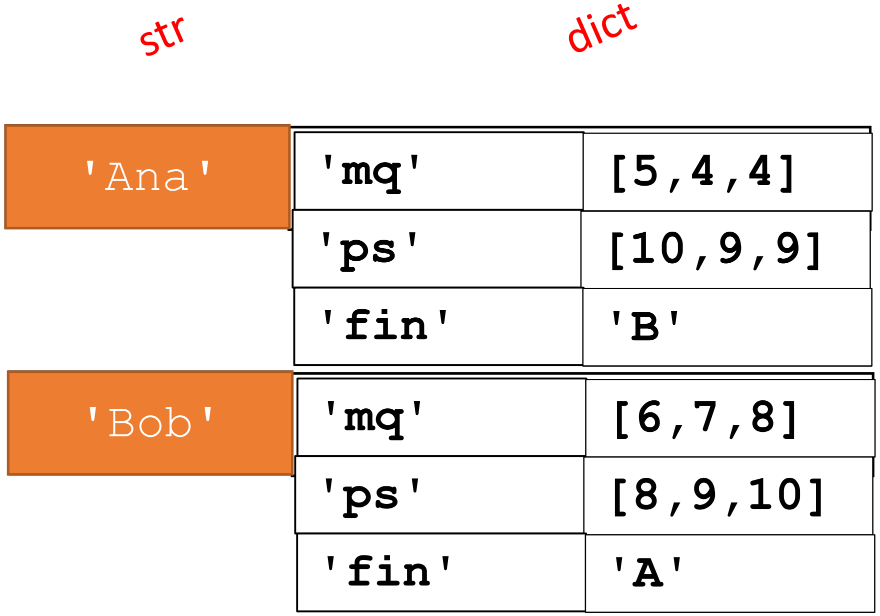
- Separate students are separate dict entries
- Entries are separated using a comma

Key 1
Val 1
Key 2
Val 2

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- Each dict entry maps a key to a value
- The mapping is done with a : character
- grades maps str:dict



str

dict

'Ana'

'mq' [5,4,4]

'ps' [10,9,9]

'fin' 'B'

'Bob'

'mq' [6,7,8]

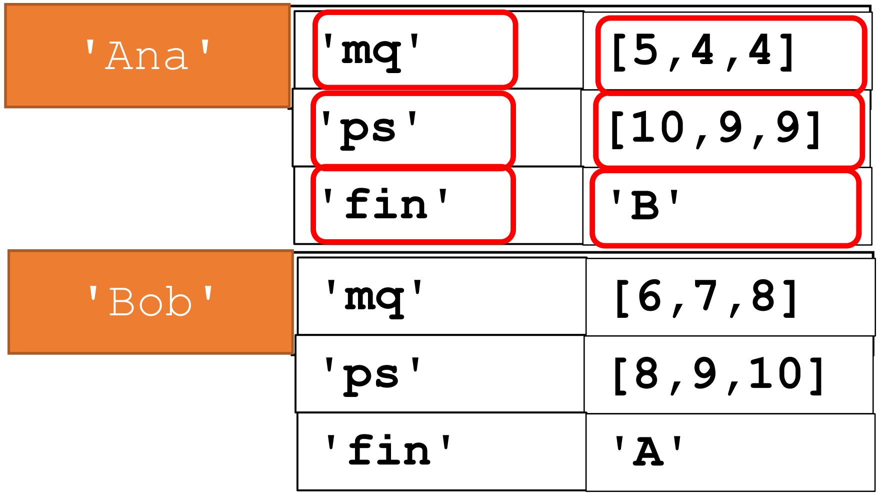
'ps' [8,9,10]

'fin' 'A'

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str

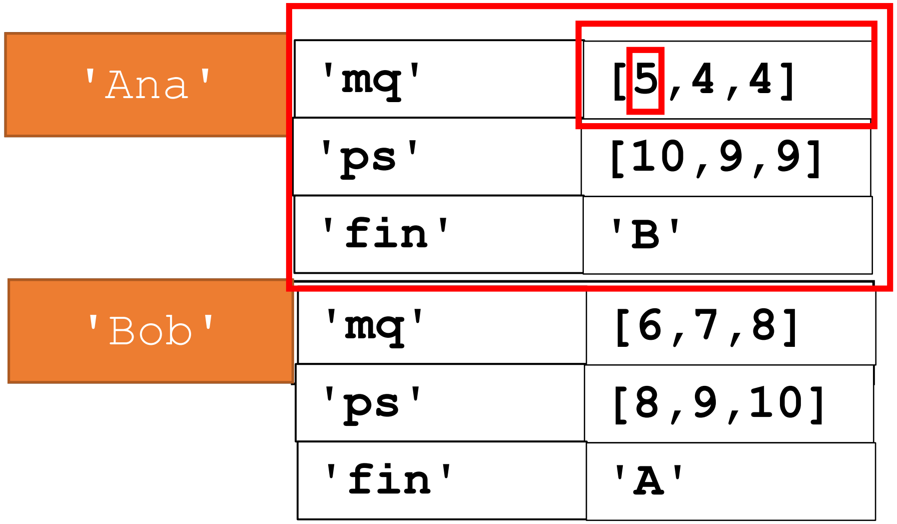


'Ana'	'mq'	[5,4,4]
	'ps'	[10,9,9]
	'fin'	'B'
'Bob'	'mq'	[6,7,8]
	'ps'	[8,9,10]
	'fin'	'A'

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

A PYTHON DICTIONARY for STUDENT GRADES

- The values of grades are dicts
- Each value maps a
  - str:list
  - str:str



'Ana'

'mq'	[5,4,4]
'ps'	[10,9,9]
'fin'	'B'

'Bob'

'mq'	[6,7,8]
'ps'	[8,9,10]
'fin'	'A'

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

`grades['Ana']['mq'][0]` **returns** `5`

YOU TRY IT!

```
my_d ={'Ana':{'mq':[10], 'ps':[10,10]},
       'Bob':{'ps':[7,8], 'mq':[8]},
       'Eric':{'mq':[3], 'ps':[0]}      }

def get_average(data, what):
    all_data = []
    for stud in data.keys():
         INSERT LINE HERE
    return sum(all_data)/len(all_data)
```

Given the dict `my_d`, and the outline of a function to compute an average, which line should be inserted where indicated so that `get_average(my_d, 'mq')` computes average for all 'mq' entries? i.e. find average of all mq scores for all students.

A) `all_data = all_data + data[stud][what]`
B) `all_data.append(data[stud][what])`
C) `all_data = all_data + data[stud[what]]`
D) `all_data.append(data[stud[what]])`

`list` vs `dict`

- **Ordered** sequence of elements
- Look up elements by an integer index
- Indices have an **order**
- Index is an **integer**
- Value can be any type

- **Matches** “keys” to “values”
- Look up one item by another item
- **No order** is guaranteed
- Key can be any **immutable** type
- Value can be any type

EXAMPLE: FIND MOST COMMON WORDS IN A SONG'S LYRICS

1) Create a **frequency dictionary** mapping `str:int`

2) Find **word that occurs most often** and how many times

- Use a list, in case more than one word with same number
- Return a tuple `(list,int)` for (words_list, highest_freq)

3) Find the **words that occur at least X times**

- Let user choose "at least X times", so allow as parameter
- Return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
- IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

Python Tutor LINK

```
song = "RAH RAH AH AH AH ROM MAH RO MAH MAH"
def generate_word_dict(song):
    song_words = song.lower()
    words_list = song_words.split()
    word_dict = {}
    for w in words_list:
        if w in word_dict:
            word_dict[w] += 1
        else:
            word_dict[w] = 1
    return word_dict
```

Convert all chars to lower case

Convert string to list of words; divides based on spaces

Can iterate over list of words in song

If word in dict (as a key), increase # times you've seen it, update entry

If word not in dict, first time seeing word, create entry

Return is a dict mapping str:int

USING THE DICTIONARY
Python Tutor LINK

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}


def find_frequent_word(word_dict):
    words = []
    highest = max(word_dict.values())
    for k,v in word_dict.items():
        if v == highest:
            words.append(k)
    return (words, highest)
```

Highest frequency in dict's values

Loop to see which word has the highest freq

Append to list of all words that have that highest freq

Return is a tuple of (['ah', 'mah'], 3)

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Repeat the next few steps as long as the highest frequency is greater than x
- Find highest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with the biggest frequency

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {'rah':2,            'rom':1,            'ro':1}
```

- Save them in the result

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Find highest frequency in the mutated dict

```
word_dict = {'rah':$\boxed{2}$,            'rom':1,            'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Use function `find_frequent_word` to get words with that frequency

```
word_dict = {'rah':2,          'rom':1,          'ro':1}
```

- The result so far…

```
freq_list = [(['ah','mah'],3)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- Remove the entries corresponding to these words from dictionary by mutation

```
word_dict = {                    'rom':1,           'ro':1}
```

- Add them to the result so far

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

FIND WORDS WITH FREQUENCY GREATER THAN x=1

- The highest frequency is now smaller than x=2, so stop

```
word_dict = {                 'rom':1,          'ro':1}
```

- The final result

```
freq_list = [(['ah','mah'],3), (['rah'],2)]
```

LEVERAGING DICT PROPERTIES
Python Tutor LINK

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def occurs_often(word_dict, x):
    freq_list = []
    word_freq_tuple = find_frequent_word(word_dict)

    while word_freq_tuple[1] > x:

        word_freq_tuple = find_frequent_word(word_dict)
        freq_list.append(word_freq_tuple)
        for word in word_freq_tuple[0]:
            del(word_dict[word])
    return freq_list
```

Gives us a word tuple
Like ([‘ah’, ‘mah’], 3)

Stay in loop while we still have
frequencies higher than x

Add those words to result

Mutate dict to remove ALL
those words; on next loop, will
find next most common words

SOME OBSERVATIONS

- Conversion of **string into list** of words enables use of list methods
  - Used `words_list = song_words.split()`
- **Iteration over list** naturally follows from structure of lists
  - Used `for w in words_list:`
- Dictionary stored the **same data in a more appropriate way**
- Ability to **access all values and all keys** of dictionary allows natural looping methods
  - Used `for k,v in word_dict.items():`
- **Mutability of dictionary** enables iterative processing
  - Used `del(word_dict[word])`
- **Reused functions** we already wrote!

SUMMARY

- Dictionaries have entries that **map a key to a value**
- **Keys are immutable/hashable and unique** objects
- **Values** can be **any object**
- Dictionaries can make code efficient
  - Implementation-wise
  - Runtime-wise