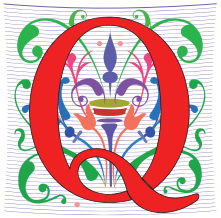
CompSci 101
Dictionaries Practice

```
28  def fastcount(words):
29      d = {}
30      for w in words:
31          if w in d:
32              d[w] += 1
33          else:
34              d[w] = 1
35      return sorted(d.items())
```

Yesenia Velasco
Susan Rodger
March 23, 2023

Q is for …



- **QR code**
  - Black and white and read all over
- **Quicksort**
  - Sort of choice before Timsort?
- **QWERTY**
  - When bad ideas persist

Christine Alvarado

- **Teaching Professor, UCSD**
- **PhD Computer Science, MIT**
- **Her work is in designing CS curriculum that is more accessible and more appealing to all**
- **LogiSketch – draw and simulate digital circuits**

"It's important to choose your own path, and try not to compare yourself to others. You have your own unique circumstance, so what others do or don't do shouldn't really affect your life."

Announcements

- **Assignment 4 GuessWord due today!**
- **APT-5 due Thur, March 30**
  - Recommend to do before Assignment 5/APT Quiz 2
- **Assign 5 Clever Guess Word out – due April 6**
  - Talk about next time
- **Lab 8 Friday, do prelab**
- **Next Week**
  - APT Quiz 2 Thurs, March 30-April 3
- **Exam 2 regrades request**

PFTD

- **Venmo Apt**
- **Dictionaries**
  - More Practice
  - Fast!
- **Family APT**
- **Clever GuessWord next time**

Assignment 5 - How to play Guess Word Cleverly

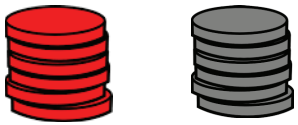
- **Make it hard for the player to win!**
- **One way: Try hard words to guess?**
  - "jazziest", "joking", "bowwowing"
- **Another Way: Keep changing the word, sortof**

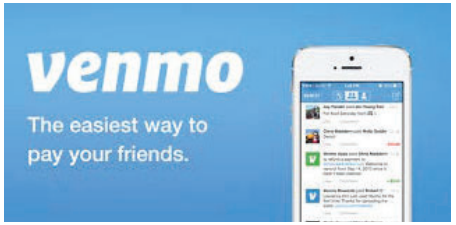
Clever GuessWord

- **Current GuessWord: Pick random secret word**
  - User starts guessing
- **Can you change secret word?**
  - Yes, but must have letters in same place you have told user
    - Change consistent with all guesses
  - Make the user work harder to guess!
- **Discuss how next time**

VenmoTracker APT

- **If Harry pays Sally $10.23,**
  - "Harry:Sally:10.23" then Harry is out $10.23





APT: VenmoTracker

Problem Statement

You've been asked to help manage reports on how often people spend money using Venmo and whether they receive more money than they pay out. The input to your program is a list of transactions from Venmo. Each transaction has the same form: `"from:to:amount"` where *from* is the name of the person paying *amount* dollars to the person whose name is *to*. The value of *amount* will be a valid float **with at most two decimal places.**

Specification

```
filename: VenmoTracker.py

def networth(transactions):
    """
    return list of strings based on transactions,
    which is also a list of strings
    """

    # you write code here
    return []
```

Return a list of strings that has each person who appears in any transaction with the net cash flow through Venmo that person has received. Every cent paid by the person to someone else is a pay-out and every cent received by a person is a pay-in. The difference between pay-out and pay-in is the cash flow received. This will be negative for each person who pays out more than they get via pay-in. See the examples for details.

The list returned should be sorted by name. Strings in the list returned are in the format "name:netflow" where the netflow is obtained by using `str(val)` where val is a float representing the net cash flow for that person.

Store money as `int` values, multiplying by 100 and dividing by 100 as needed for processing input and output, respectively.

APT Venmo Tracker Example

Examples

1. `transactions: ["owen:susan:10", "owen:robert:10", "owen:drew:10"]`

   `returns ['drew:10.0', 'owen:-30.0', 'robert:10.0', 'susan:10.0']`

   Owen pays everyone.

WOTO-1 VenmoTracker
http://bit.ly/101s23-0323-1

Tools We've Used Before

- **Keep track of every person we see**
  - Use a list
- **Keep track of net worth: money in, money out**
  - Use a parallel list
- **Maintain invariant: `names[k] <-> money[k]`**
  - k^{th} name has k^{th} money

Example:

["Harry:Sally:10.23“, “Zeyu:Sally:20.00”,
“Sally:Barak:10.00”]

- **How would we solve this?**
- **Could we use a parallel list?**
- **What would be the output?**

Process Transaction "Harry:Sally:10.23“

names = [ ]

money = [ ]

Put Harry in: "Harry:Sally:10.23“

names = [“Harry”]
0

money = [-10.23]
0

Put Sally in: "Harry:Sally:10.23“

names = [“Harry”, “Sally”]
0 1

money = [-10.23, 10.23]
0 1

Process next transaction "Zeyu:Sally:20.00"

```
names  = [ “Harry”,      “Sally”  ]
             0             1

money = [  -10.23,      10.23   ]
             0             1
```


Put Zeyu in: “Zeyu:Sally:20.00”

```
names  = [ “Harry”,      “Sally”,     “Zeyu”  ]
             0             1            2

money = [  -10.23,      10.23,      -20.00   ]
             0             1            2
```


Update Sally in: “Zeyu:Sally:20.00”

```
names  = [ “Harry”,      “Sally”,     “Zeyu”  ]
             0             1            2

money = [  -10.23,      30.23,      -20.00   ]
             0             1            2
```


Process next Transaction “Sally:Barak:10.00”

```
names  = [ “Harry”,      “Sally”,     “Zeyu”  ]
             0             1            2

money = [  -10.23,      30.23,      -20.00   ]
             0             1            2
```

Update Sally in: "Sally:Barak:10.00"

```
names  = [ "Harry",    "Sally",    "Zeyu"  ]
             0           1           2

money = [  -10.23,     20.23,     -20.00   ]
             0           1           2
```


Add Barak in: "Sally:Barak:10.00"

```
names  = [ "Harry",    "Sally",    "Zeyu",   "Barak"  ]
             0           1           2         3

money = [  -10.23,     20.23,     -20.00,    10.00   ]
             0           1           2         3
```


Coding up Venmo

```
def networth(transactions):
    names = [ ]
    money = [ ]
    for trans in transactions:
        # split up trans
```


Coding up Venmo

```
def networth(transactions):
    names = [ ]
    money = [ ]
    for trans in transactions:
        # split up trans
        data = trans.split(":")
        sender = data[0]
        receiver = data[1]
        amount = float(data[2])
```

Coding up Venmo

```
if sender not in names:
        names.append(sender)
       money.append(0)
# similar if receiver not in names
# update money
 indexSender = names.index(sender)
indexReceiver = names.index(receiver)
money[indexSender]  -=  amount
money[indexReceiver] += amount
# create output in correct format
```


Seen parallel lists before

- **Solution outlined is reasonable, efficient?**
  - How long does it take to find index of name?
  - It depends. Why?
- **`list.index(elt)` or `elt in list` – fast?**
  - What does "fast" mean? Relative to what?

Let's try Dictionaries….

Example:

["Harry:Sally:10.23", "Zeyu:Sally:20.00",
"Sally:Barak:10.00]

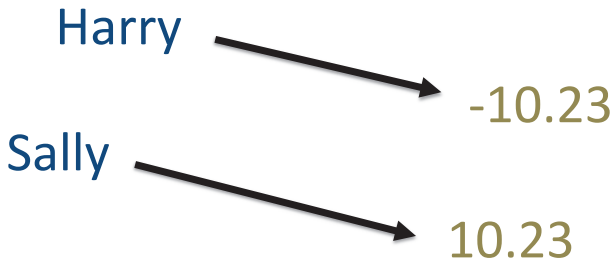
- **How would we solve this?**
- **Could we use a dictionary?**
- **What would be the output?**

Example with Dictionary
1) "Harry:Sally:10.23"

- **Start with empty dictionary, insert Harry**

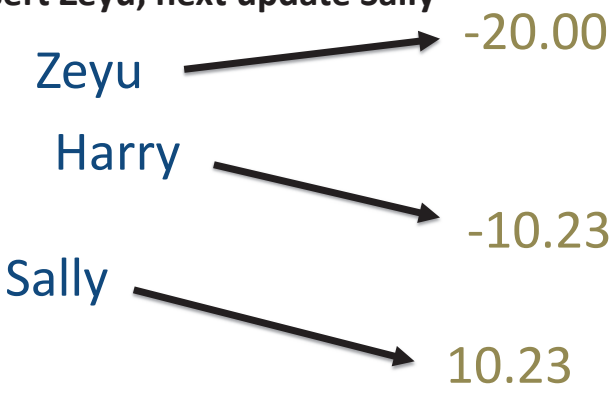
Example with Dictionary
1) "Harry:Sally:10.23"

- **Insert Sally**



Example with Dictionary
2) “Zeyu:Sally:20.00"

- **Insert Zeyu, next update Sally**

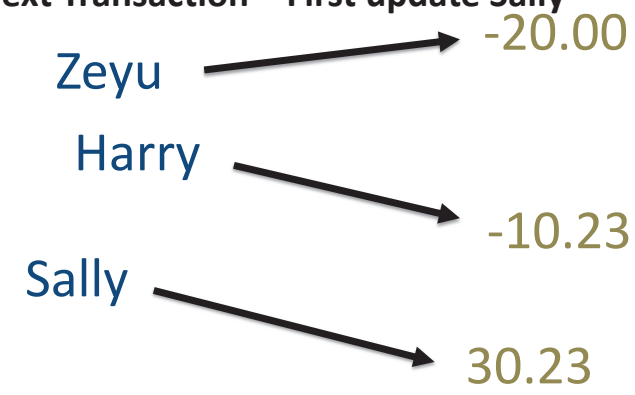


Example with Dictionary
2) “Zeyu:Sally:20.00"

- **Update Sally**

Example with Dictionary
3) “Sally:Barak:10.00"

- **Next Transaction – First update Sally**



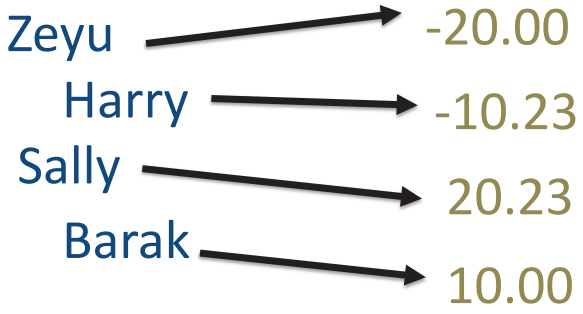
Example with Dictionary
3) “Sally:Barak:10.00"

- **Update Sally**

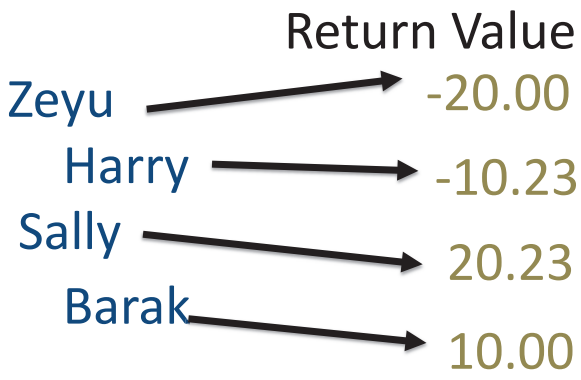
Example with Dictionary
3) “Sally:Barak:10.00"

- **Insert Barak**

Return Value



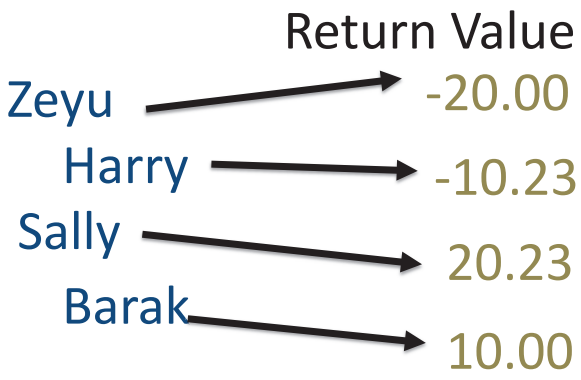
- **List of (key, value) pairs**
- **[("Zeyu", -20.00), ("Harry", -10.23), ("Sally", 20.23), ("Barak", 10.00)]**



- **[("Zeyu", -20.00), ("Harry", -10.23), ("Sally", 20.23), ("Barak", 10.00)]**

Sort by name:

- **[("Barak", 10.00), ("Harry", -10.23), ("Sally", 20.23), ("Zeyu", -20.00)]**



- **[("Barak", 10.00), ("Harry", -10.23), ("Sally", 20.23), ("Zeyu", -20.00)]**

Put in final format:

- **["Barak:10.00", "Harry:-10:23", "Sally:20.23", "Zeyu:-20.00"]**

How would the code be different if we used a dictionary?

Coding up Venmo with Dictionary

```
def networth(transactions):
    venmo = { }
    for trans in transactions:
        # split up trans
```

Coding up Venmo with Dictionary

```
def networth(transactions):
    venmo = { }
    for trans in transactions:
        # split up trans
        data = trans.split(":")
        sender = data[0]
        receiver = data[1]
        amount = float(data[2])
```

Initialize dictionary

This part the same

Coding up Venmo with Dictionary

```
if sender not in venmo:
    venmo[sender] = 0
# similar if receiver not in names
# update money
venmo[sender]  -=  amount
venmo[receiver] += amount
# create output in correct format
```

Insert in dictionary

update values

Code is shorter for dictionaries!
Code is faster for dictionaries!

You will need to finish it

- **Now onto more on Dictionaries…**

Dictionary Iteration (unordered!)

- **Iterate through keys:**
  - `for k in d:`
  - `for k in d.keys():`
- **Iterate through pairs:**
  - `for (k,v) in d.items():`
  - `for k,v in d.items():`

Sorting a list from dictionary - sorted()

d = {'k': 3, 'h': 8, 'a': 12, 'd': 5}

x = sorted(d.keys())
y = sorted(d.values())
z = sorted(d.items())

Sorting a list from dictionary - sorted()

d = {'k': 3, 'h': 8, 'a': 12, 'd': 5}

x = sorted(d.keys())
y = sorted(d.values())
z = sorted(d.items())

x is ['a', 'd', 'h', 'k']
y is [3, 5, 8, 12]
z is [('a', 12), ('d', 5), ('h',8), ('k', 3)]

WordFrequencies Dictionary Example

- **Let's see an example that compares using a dictionary vs not using a dictionary**

slowcount function Short Code and Long Time

- **See module WordFrequencies.py**
  - Find # times each word in a list of words occurs
  - We have tuple/pair: word and word-frequency

```
37  def slowcount(words):
38      pairs = [(w,words.count(w)) for w in set(words)]
39      return sorted(pairs)
```

- **Think: How many times is `words.count(w)` called?**
  - Why is **`set(words)`** used in list comprehension?

WordFrequencies with Dictionary

- **If start with a million words, then…**
- **We look at a million words to count # "cats"**
  - Then a million words to count # "dogs"
  - Could update with parallel lists, but still slow!
  - Look at each word once: dictionary!

- **Key idea: use word as the "key" to find occurrences, update as needed**
  - Syntax similar to `counter[k] += 1`

Using fastcount

- **Update count if we've seen word before**
  - Otherwise it's the first time, occurs once

```
28  def fastcount(words):
29      d = {}
30      for w in words:
31          if w in d:
32              d[w] += 1
33          else:
34              d[w] = 1
35      return sorted(d.items())
```


Using fastcount

- **Update count if we've seen word before**
  - Otherwise it's the first time, occurs once

```
28  def fastcount(words):
29      d = {}
30      for w in words:
31          if w in d:
32              d[w] += 1
33          else:
34              d[w] = 1
35      return sorted(d.items())
```

Initialize dictionary

Check if key is in d

key already in d, update

If key not in, put in with value

Let's run them and compare them!

- **Run with Melville and observe time**

- **Run with Hawthorne and observe time**

Let's run them and compare them!

- **Run with Melville and observe time**
  - slowcount about 0.76 seconds
  - fastcount about 0.00 seconds

- **Run with Hawthorne and observe time**
  - slowcount about 14.6 seconds
  - fastcount about 0.03 seconds

WOTO-2 Counting Dictionaries
http://bit.ly/101s23-0323-2

Problem Solving

- **Given Brodhead University. They have a basketball team.**
- **Data on players and how they did when playing against another team.**

- **List of lists named datalist**
  - Each list has
    - school opponent name
    - player name
    - Points player scored
    - Whether game was 'won' or 'lost'

Example: lists of 20 lists
datalist =

```
[ ['Duke', 'Bolton', '2', 'lost'],
  ['NCSU', 'Stone', '12', 'won'],
  ['Duke', 'Kreitz', '3', 'lost'],
  ['Duke', 'Pura', '6', 'lost'],
  ['GT', 'Dolgin', '4', 'lost'],
  ['WFU', 'Laveman', '20', 'won'],
  ['ECU', 'Parlin', '15', 'won'],
  ['UNC', 'Stone', '17', 'won'],
  ['UNC', 'Dolgin', '12', 'won'],
  ['UNC', 'Kreitz', '5', 'won'],
  ['Duke', 'Stone', '16', 'lost'],
  ['Duke', 'Laveman', '13', 'lost'],
  ['NCSU', 'Kreitz', '8', 'won'],
  ['NCSU', 'Dolgin', '18', 'won'],
  ['NCSU', 'Parlin', '13', 'won'],
  ['GT', 'Bolton', '7', 'lost'],
  ['GT', 'Stone', '9', 'lost'],
  ['WFU', 'Parlin', '14', 'won'],
  ['ECU', 'Laveman', '16', 'won'],
  ['ECU', 'Pura', '15', 'won'] ]
```

1) Write function dictPlayerToNumGamesPlayedIn

Build a dictionary of players mapped to number of games they have played in.

def dictPlayerToNumGamesPlayedIn(datalist):

With previous example, player 'Laveman' would be mapped to 3 games

Woto-3 Players and Games Played in http://bit.ly/101s23-0323-3

Write function dictPlayerToNumGamesPlayedIn

```
def dictPlayerToNumGamesPlayedIn(datalist):
    d = {}
    for line in datalist:
        player = line[1]
        if player in d:
            d[player] += 1
        else:
            d[player] = 1
    return d
```

When each item needs its own count, build a dictionary

This is a counting dictionary

ANOTHER WAY: Write function dictPlayerToNumGamesPlayedIn

```
def dictPlayerToNumGamesPlayedIn(datalist):
    d = {}
    for line in datalist:
        player = line[1]
        if player not in d:
            d[player] = 0
        d[player] += 1
    return d
```

2) Write function

playersPlayedInNumGames(number, datalist)

Calculate list of players who played in 3 or more games, give (player name, number of games played in), sort by player name

[('Dolgin', 3), ('Kreitz', 3), ('Laveman', 3), ('Parlin', 3), ('Stone', 4)]

```
[ ['Duke', 'Bolton', '2', 'lost'],
  ['NCSU', 'Stone', '12', 'won'],
  ['Duke', 'Kreitz', '3', 'lost'],
  ['Duke', 'Pura', '6', 'lost'],
  ['GT', 'Dolgin', '4', 'lost'],
  ['WFU', 'Laveman', '20', 'won'],
  ['ECU', 'Parlin', '15', 'won'],
  ['UNC', 'Stone', '17', 'won'],
  ['UNC', 'Dolgin', '12', 'won'],
  ['UNC', 'Kreitz', '5', 'won'],
  ['Duke', 'Stone', '16', 'lost'],
  ['Duke', 'Laveman', '13', 'lost'],
  ['NCSU', 'Kreitz', '8', 'won'],
  ['NCSU', 'Dolgin', '18', 'won'],
  ['NCSU', 'Parlin', '13', 'won'],
  ['GT', 'Bolton', '7', 'lost'],
  ['GT', 'Stone', '9', 'lost'],
  ['WFU', 'Parlin', '14', 'won'],
  ['ECU', 'Laveman', '16', 'won'],
  ['ECU', 'Pura', '15', 'won'] ]
```


2) Write function

playersPlayedInNumGames(number, datalist)

```
def playersPlayedInNumGames(number, datalist):
    d = dictPlayerToNumGamesPlayedIn(datalist)
    # build a list of tuples
    answer = [ ]
    for player in d.keys():
        if d[player] >= number:
            answer.append( (player, d[player] ))
    return sorted(answer)
```


ANOTHER WAY 2) Write function

playersPlayedInNumGames(number, datalist)

```
def playersPlayedInNumGames(number, datalist):
    d = dictPlayerToNumGamesPlayedIn(datalist)
    # build a list of tuples
    answer = [ ]
    for (player, count) in d.items():
        if count >= number:
            answer.append( (player, count) )
    return sorted(answer)
```

ANOTHER WAY 2) Write function playersPlayedInNumGames(number, datalist)

Another way using a list comprehension!

However, this is putting a lot in one long line.

It may be better to break it up into steps as the previous two slides do. Less chance to make a mistake.

```
def playersPlayedInNumGames(number, datalist):
    d = dictPlayerToNumGamesPlayedIn(datalist)
    # build a list of tuples
    return sorted([ (player, count) for (player,count) in
        d.items() if count >= number] )
```


Stopped here
Will do next time

APT Family

APT: Family

Problem Statement

You have two lists: `parents` and `children`. The ith element in `parents` is the parent of the ith element in `children`. Count the number of grandchildren (the children of a person's children) for the person in the `person` variable.

Hint: Consider making a helper function that returns a list of a person's children.