CS 111: Program Design I

Lecture 23: CS: Network Analysis, Dictionaries, Degree distribution

Robert H. Sloan & Richard Warner
University of Illinois at Chicago
November 19, 2019

UIC

NETWORK ANALYSIS (CONTINUED)

Networkx

- To work with graphs in Python, especially for network analysis:
- `import networkx (as nx)`
- Learn more at:
  - https://networkx.github.io/documentation/networkx-2.3/tutorial.html. Spyder almost certainly has version 2.3, almost identical to version 2.4, released in October. (Some important differences from old versions 1.x from 2017 and before)
  - networkx provides Graph as basic data type and ways to add nodes and edges and do all sorts of things, including visualize

Simple graph example

```
import networkx as nx

g = nx.Graph()        #Create an empty graph object

#Add several nodes
g.add_node('Alice')
g.add_node('Bob')
g.add_node('Charlie')

g.number_of_nodes()       → 3
g.number_of_edges()       → 0
```

Simple graph example continued

```
# Add a single edge
In [9]: g.add_edge('Alice', 'Bob')          # undirected
In [10]: g.number_of_edges()
Out[10]: 1
In [11]: g.nodes()
Out[11]: ['Alice', 'Charlie', 'Bob']
In [12]: g.edges()
Out[12]: EdgeView([('Alice', 'Bob')])
```

Drawing

- networkx can do simple drawing (working with matplotlib.pyplot under hood):

```
nx.draw(g, with_labels='True')
```

Drawing without node labels

```
nx.draw(g)
```

(or, for control freaks or the pedantic)

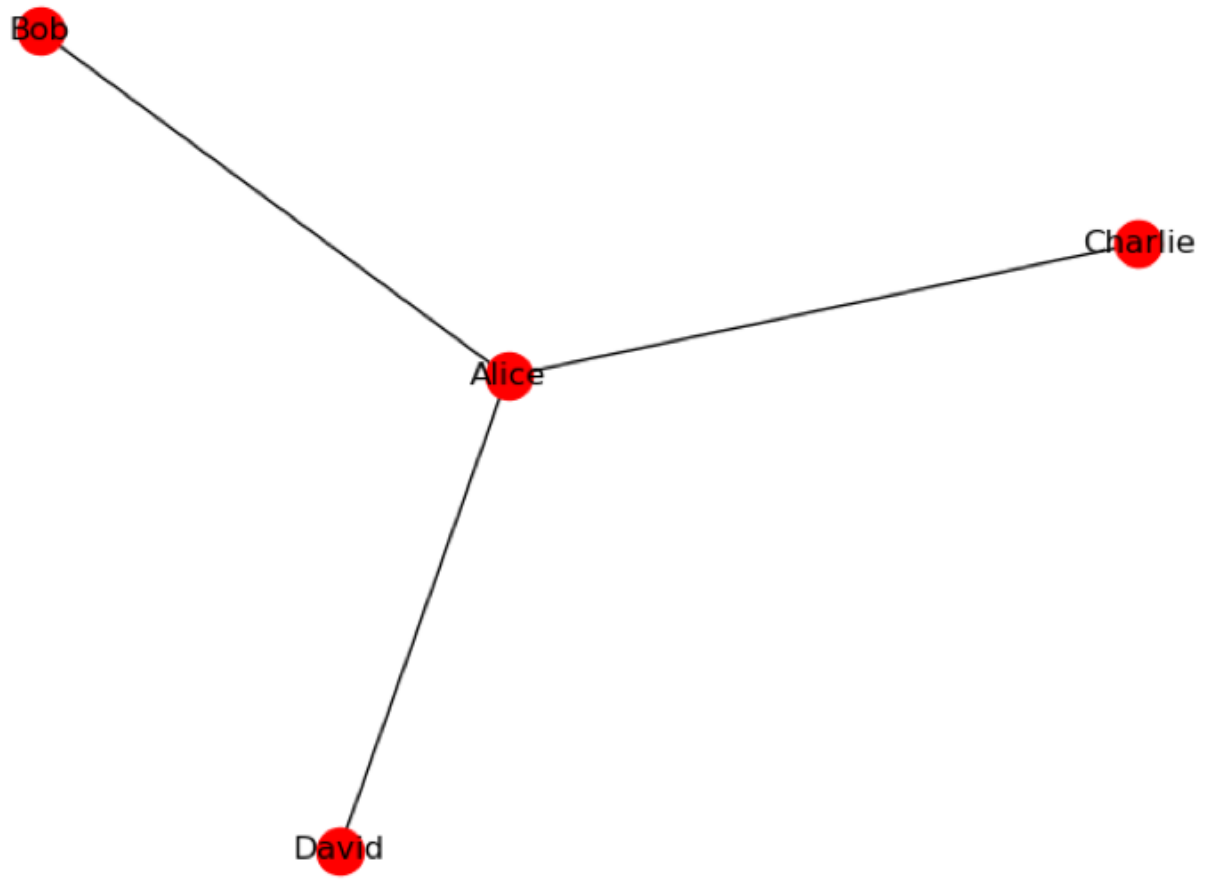
```
nx.draw(g, with_labels=False)
```

Adding a bit to the graph

```
# Add some more edges and nodes g.add_node("David")
g.add_edges_from([("Alice", "Charlie"),
                  ("Alice", "David")])
```

- add_edges_from is a method whose argument should be *list* of pairs of node names, each pair in ()s.

networkx.draw(g) (as updated)



Key graph statistic 1: Degree

- Degree of node = number of neighbors node has
- Range of different degrees discovered in last 20–30 years to vary with nature of graph.
- networkx has graph method degree that gives us special data structure easily converted to a *dictionary* with all degree information for graph

degrees from networkx

```
In [5]: d = networkx.degree(g)
In [6]: d
Out[6]: DegreeView({'Alice': 3, 'Bob': 1, 'Charlie': 1,
'David': 1})
In [7]: ddict = dict(d)
In [8]: ddict
Out[8]: {'Alice': 3, 'Bob': 1, 'Charlie': 1, 'David': 1}
```

Will explain dictionaries shortly; for now: pandas knows them!

networkx degree data → pandas

```
In [7]: degree_data = pandas.Series(dict(nx.degree(g)))
In [8]: degree_data
Out[8]:
Alice      3
Bob        1
Charlie    1
David      1
```

- And we know from earlier in semester how to plot graphs from pandas series: pandas series has method .plot()

Plotting degree data

- We want a *histogram:* Bar plot where the things on the x-axis have a specific meaningful order (e.g., numbers), as opposed to being categorical (e.g., names of justices)

```
degree_data.plot(kind='hist')
```

- (Side note: Abbreviation for that. Can write .hist() instead of .plot(kind='hist')): `degree_data.hist()`

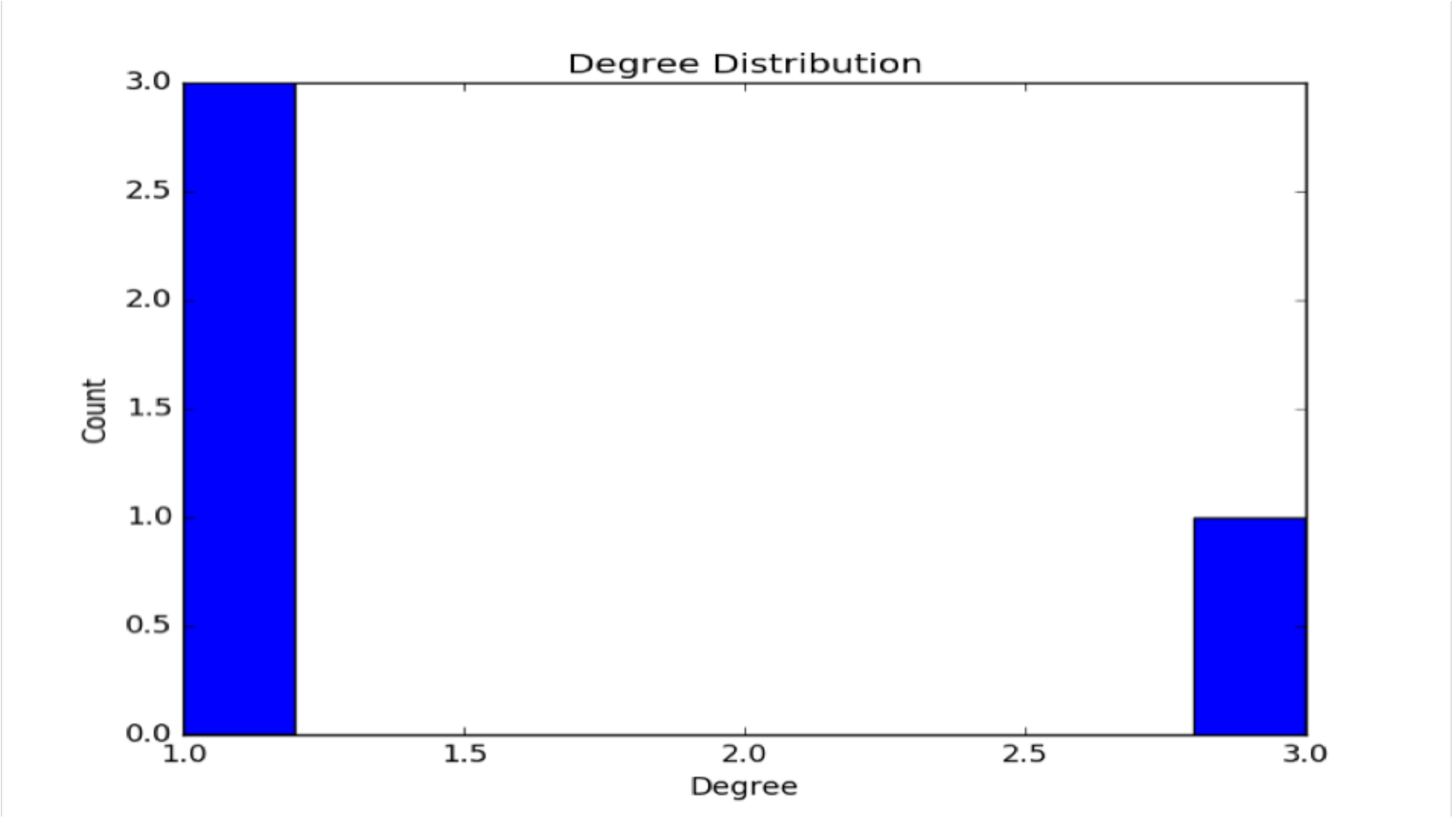
To make plot of series look nice

- pandas put in stuff automatically for some plots we did earlier from dataframes, but doesn't always (especially not for series). If need be:

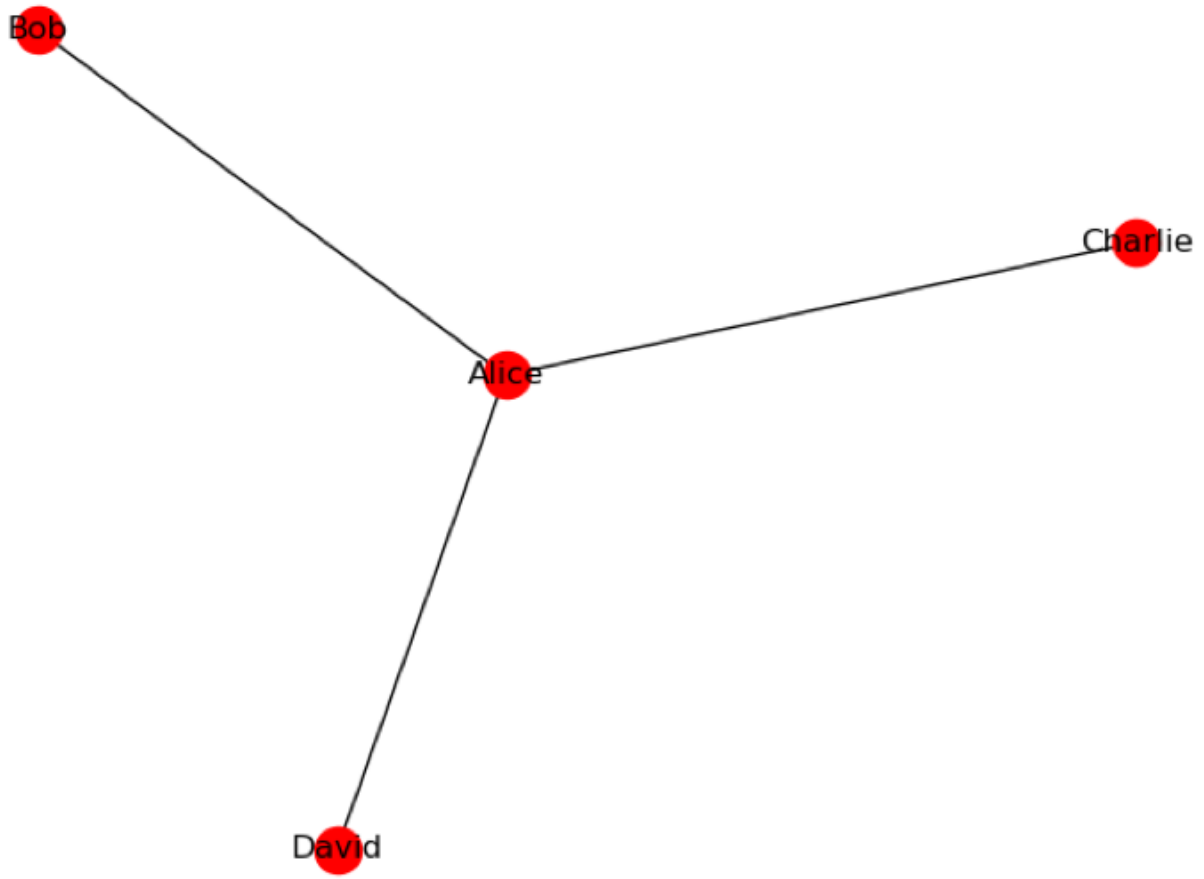
```
import matplotlib.pyplot as plt

plt.xlabel('string I want to see below x axis')
plt.ylabel('Similarly for y')
plt.title('String I want up top in title position')
```

Plot with some appropriate labels



Remember what our graph looks like



Centrality

- Alice is connected to everybody else; Bob, Charlie, and David are connected only to one node each (Alice)
- Alice is obviously the most central
- Various *centrality measures* to tell which nodes are most central
- (Prof. Philip Yu of UIC CS found to be "most central" computer science author by one such measure)

What is maximum degree of node in graph with n nodes?

A. n
B. $n - 1$
C. $n - 2$
D. $n(n - 1) / 2$
E. 42

One simple measure of centrality of node

- degree of node / maximum possible degree of any node in that node's graph
- For *n*-node graph:
  - degree of node / (*n* - 1)
- networkx will give us (a dictionary of) the centrality of every node in graph g:

```
cent = nx.degree_centrality(g)
```

For our little graph

```
In [11]: cent = nx.degree_centrality(g)


In [12]: cent
Out[12]:
{'Alice': 1.0, 'Bob': 0.3333333333333333,
 'Charlie': 0.3333333333333333,
 'David': 0.3333333333333333}
```

Dictionary → pandas Series

- For some purposes want dictionary format (e.g., look up degree of a node)
  - And dictionaries key storage structure for lots of non-data science uses; more shortly
- For some analytics, can convert dictionary of degree centrality form (keys=strings, values=numbers) to pandas.Series and use pandas to do stuff
  - E.g., if I asked you to sort the degrees.

Path lengths

- How many edges do we need to walk over to get from one node to another?
- 0 to get from node to itself
- 1 to get to immediate neighbor
- > 1 to get to all other nodes

Getting all path lengths

- neworkx has operator for this; gives somewhat complex data structure back
- pandas to the rescue: It knows how to handle that data structure and turn it into a dataframe, which we already know about:

```
pandas.DataFrame(dict(nx.all_pairs_shortest_path_length
(g)))
```

All path lengths in our graph

```
p = pandas.DataFrame(nx.all_pairs_shortest_path_length(g))
```

```
>>> p
         Alice  Bob  Charlie  David
Alice        0    1        1      1
Bob          1    0        2      2
Charlie      1    2        0      2
David        1    2        2      0
```

Another stat: Average shortest path length

- networkx will calculate the average over all shortest path lengths for you:

```
# Get the average path length
print(networkx.average_shortest_path_length(g))
1.5
```

Our 4 node graph is kinda dull

- Point is to apply these sorts of techniques to e.g., graphs of various types of social networks with thousands to 1 billion+ nodes
- Our example data (real data):
  - nodes = twitter users
  - edge = follows relationship (could be directed; could ignore direction)
  - ~40,000 pairs of follower, followee
  - (This particular bit of twitter formed by technique called snowball sampling starting at Computational Legal Analytics)

Large networks

- Stored as text files
- One line for each link with line containing names (string or number) of nodes
  - Notice that if we know all the links then we know what the nodes are
  - Both comma and space are common delimiters for between the two nodes of an edge in large network work
  - Both are, broadly speaking, CSV
- We'll use pandas to read these in

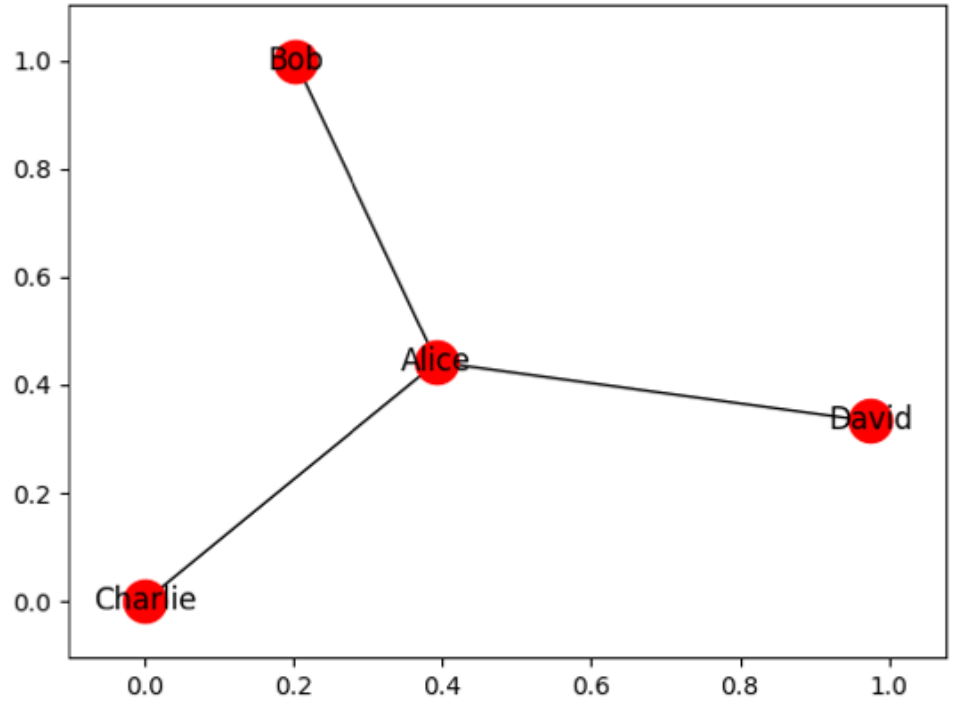
Reading graphs from files pandas

- (The joys of working with real data! ☺)
- Can still have encoding issues for, e.g., Chinese node name
- *CSV* files and csv_read *default*: *rows separated by newlines* and *items in rows separated by commas*, but can specify item separator either
  - Comma: pandas.read_csv(<filename>)
  - Space: pandas.read_csv(<filename>, sep = ' ') # Project?

DRAWING GRAPHS TO DICTIONARIES

Drawing a graph: Node Labels

- Earlier created and drew graph g on right
- Whose nodes were Alice, Bob, Charlie, and David
- `nx.draw(g, with_labels='True'))` put node labels on the graph automatically



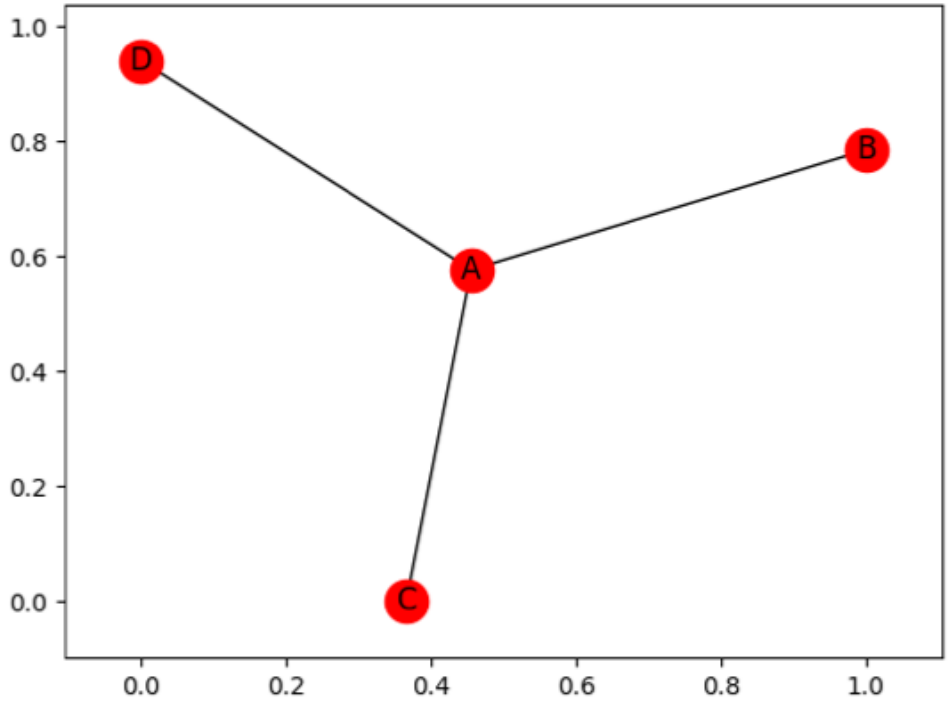
Introducing *Dictionaries*: Setting node labels

```
import networkx as nx

labels_dict = {'Alice':'A', Bob':'B',
      'David':'D', 'Charlie':'C'}
nx.draw(g, labels=labels_dict)
```

would label nodes A, B, C, D.

(Note argument is *labels* plural)



Introducing Dictionaries: Setting node labels (cont)

```
labels_dict = {'Alice':'A', 'Bob':'B', 'David':'D',
                        'Charlie':'C'}
nx.draw_networkx(g, labels=labels_dict)
```

- In general, dictionary is unordered collection of key-value pairs, and here key (left) is node's name, value (right of colon) is what we want printed
  - Much more on dictionaries in 2 slides

Other optional arguments to nx.draw

- If labels, can adjust their size with font_size = <number>
  - Got larger labels for PowerPoint on earlier slide with font_size=20
- Can also change appearance of nodes, e.g.,
  - node_size=8
  - node_color="red" # or "r"

- What super power did I use to learn all those parameters?
  - Documentation at: https://networkx.github.io/documentation/stable/index.html
  - Specifically https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html

DICTIONARIES

Bird Watching

- We want to keep track of how many of each bird we have seen
  - Robin: 3, Pigeon: 45, etc.

- Could use parallel lists

```
birds = ['robin', 'pigeon', 'falcon']
counts = [3, 45, 2]
```

Where did I put that darn pigeon?

```
birds = ['robin', 'pigeon', 'falcon']
counts = [3, 45, 2]
```

- Recall list method .index(val) returns first index at which value val occurs in list *but is error if val not in list*

- birds.index('pigeon') → 1
- birds.index('chicken) → barf

Bird Watching

- We want to keep track of how many of each bird we have seen
  - Robin: 3, Pigeon: 45, etc.

- Could use parallel lists

```
birds = ['robin', 'pigeon', 'falcon']
counts = [3, 45, 2]
```

Adding a new bird sighting?

```
def new_sighting(birds, counts, new_bird):
    """Manages bird counts using 2 parallel lists"""
    if new_bird not in birds:
        birds.append(new_bird)
        (possible) missing line
    ind = birds.index(new_bird)
    counts[ind] = counts[ind] + 1
```

A. `counts.append(0)`
B. `counts.append(1)`
C. `counts.append(new_bird)`
D. No code necessary
E. I don't know

Using Dictionaries

```
bird_dict = {"robin":3, "pigeon":45, "falcon":3}

def new_sighting(bird_dict, new_bird):
    if new_bird not in bird_dict:
        bird_dict[new_bird] = 0
    bird_dict[new_bird] = bird_dict[new_bird] + 1
```

- Only one dictionary
- Instead of looking for index, look up by *key*

Keys and Values

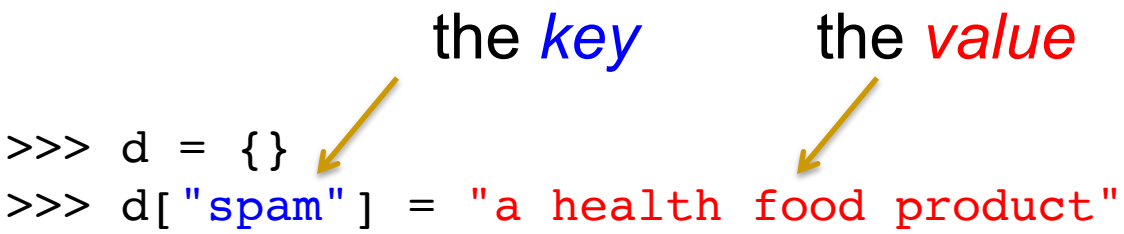
- Keys are immutable
- Values are mutable

- Use `d[k] = v` to add key k with value v to dictionary d
- If k is already present, its value is overwritten

Dictionaries!

```
>>> d = {}
```

Dictionaries!



Dictionaries!


```
>>> d = {}
>>> d["spam"] = "a health food product"
>>> d[42] = "an important number"
>>> d["bart"] = ["bart@geemail.com", "springfield"]
```

Dictionaries!


```
>>> d = {}
>>> d["spam"] = "a health food product"
>>> d[42] = "an important number"
>>> d["bart"] = ["bart@geemail.com", "springfield"]
>>> d
{42: 'an important number', 'bart': ['bart@geemail.com', 'springfield'], 'spam': 'a health food product'}
```

Dictionaries!


```
>>> d = {}
>>> d["spam"] = "a health food product"
>>> d[42] = "an important number"
>>> d["bart"] = ["bart@geemail.com", "springfield"]
>>> d
{42: 'an important number', 'bart': ['bart@geemail.com', 'springfield'], 'spam': 'a health food product'}
>>> d[42]
'an important number'
```

Dictionaries!


```
>>> d = {}
>>> d["spam"] = "a health food product"
>>> d[42] = "an important number"
>>> d["bart"] = ["bart@geemail.com", "springfield"]
>>> d
{42: 'an important number', 'bart': ['bart@geemail.com', 'springfield'], 'spam': 'a health food product'}
>>> d[42]
'an important number'
>>> 42 in d
True
>>> 43 in d
False
>>> "ran" in d
False
```

Dictionaries!

the *key* the *value*

```
>>> d = {}
>>> d["spam"] = "a health food product"
>>> d[42] = "an important number"
>>> d["bart"] = ["bart@geemail.com", "springfield"]
>>> d
{42: 'an important number', 'bart': ['bart@geemail.com', 'springfield'], 'spam': 'a health food product'}
>>> d[42]
'an important number'
>>> 42 in d
True
>>> 43 in d
False
>>> "ran" in d
False
>>> d["ran"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
```

```
"an important number" in d

A. True
B. False
C. Error
```

Dictionaries: Keys and Values, types

- Keys can be *any* immutable type
  - int, string, or float
- Values are mutable and can be any type under the sun
  - including dictionary

- Dictionary *itself* is a mutable type:

- Recall `d[k] = v` overwrites if k was already a key

Getting Values *from* Dictionaries

- `d[k]` returns value associated with key k in dictionary d
  - If k does not exist, this causes an error

- `d.get(k)` also returns value associated with key k in dictionary d
  - Returns None if k does not exist
  - If a second parameter is included `d.get(k, v)`, then v returned instead of None if k not found

What is d at the end of this code?

```
d = {3:4}
d[5] = d.get(4, 8)
d[4] = d.get(3, 9)
```

A. {3:4, 5:8, 4:9}
B. {3:4, 5:8, 4:4}
C. {3:4, 5:4, 4:3}
D. Error caused by get
E. None

How confident are you of your answer?

A. Very Highly confident: I've got this
B. Very confident
C. Somewhat confident
D. Not so confident: educated guess
E. Not confident at all: random guess and/or bullied into by the rest of my small group

Keys, Values and Items

- `d.keys()` returns a dictionary's keys
- `d.values()` returns a dictionary's values
- `d.items()` returns a dictionary's key-value pairs

- These are similar to lists, but *NOT* lists. To turn into a list, `list(d.keys())`
- Just as thing returned by range is similar to but not a list, and thing returned by `ur.connect` is similar to but not a string

Deleting from a dictionary

- Occasionally need to delete key–value entry from dictionary d
- Python has way to do this:
  - del d[key]
  - Syntax is a little odd; technically del is *operator*

Accessing entire dictionary

- for loops can be over dictionaries as well as lists
- loop variable is successive *keys*

```
for key in d:
    Probably stuff involving value d[k] as well as just k
```

DEGREE DISTRIBUTIONS

How many close (real world) neighbors

- Estimate number of other people living within 100 feet of where you sleep at night:

A. 0–7
B. 8–15
C. 16–32
D. 32–64
E. 65+

How many close neighbors

- Estimate highest number of other people living within 100 feet of bed of anyone in Chicago area not in dorm, prison, military, or hospital

A. 0–32
B. 32–64
C. 66–125
D. 128–250
E. 250+