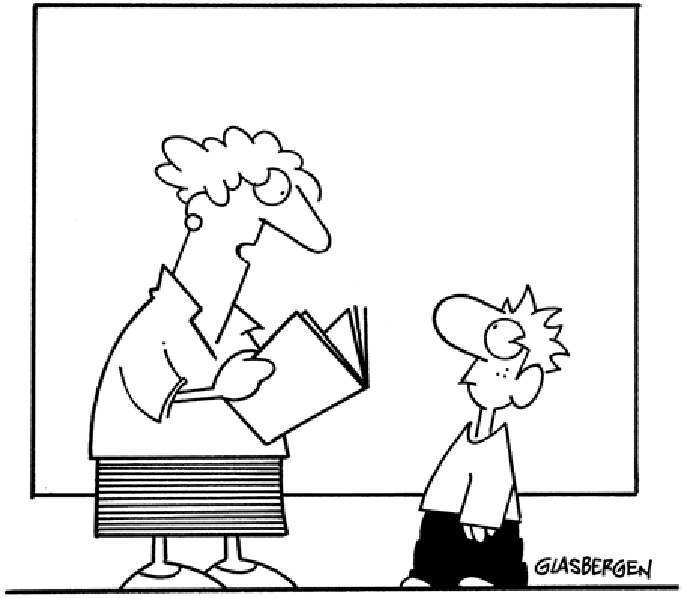
CSc 110, Spring 2017

Lecture 28: Sets and Dictionaries

Adapted from slides by Marty Stepp and Stuart Reges

© Randy Glasbergen / glasbergen.com



"Yes, some books come in high definition — dictionaries!"

Mountain peak

Write a program that reads elevation data from a file, draws it on a `DrawingPanel` and finds the path from the highest elevation to the edge of the region.

Data:

```
34 76 87 9 34 8  22 33 33 33 45 65 43 22
5  7  88 0 56 76 76 77 4  45 55 55 4  5
```

…

Mountain peak

```
data =
 [ [34, 76, 87, 9, 34, 8, 22, 33, 33, 33, 45, 65, 43, 22]
   [5,   7, 88, 0, 56, 76, 76, 77, 4, 45, 55, 55,  4, 5]
   ...
 ]
```

Next steps:

4) Find the peak
5) Find the steepest path down
6) Draw the path in yellow

4) Find the peak

```
data =
  [ [34, 76, 87, 9, 34, 8, 22, 33, 33, 33, 45, 65, 43, 22]
    [5,   7, 88, 0, 56, 76, 76, 77, 4, 45, 55, 55,  4, 5]
    ...
  ]
```

Find the largest elevation in the list of lists. Write `find_peak(data)`

Return a tuple of the location in the 2d list

5) Find the steepest path

```
data =
[['2537', '2483', '2475', '2480', '2518', '2532', '2480', '2478', '2431']
 ['2541', '2549', '2614', '2700', '2647', '2746', '2690', '2621', '2550']
 ['2525', '2525', '2640', '2769', '2802', '2883', '2856', '2694', '2631']
 ['2514', '2505', '2526', '2614', '2717', '2715', '2867', '2836', '2771']
 ['2506', '2482', '2480', '2528', '2518', '2561', '2586', '2662', '2654']
 ['2527', '2477', '2464', '2459', '2452', '2475', '2480', '2500', '2518']
 ['2544', '2505', '2488', '2454', '2442', '2445', '2446', '2467', '2470']
 ['2528', '2486', '2464', '2446', '2434', '2436', '2442', '2444', '2450']
 ['2464', '2505', '2482', '2456', '2433', '2463', '2462', '2489', '2467']
 ['2532', '2541', '2519', '2515', '2496', '2502', '2529', '2519', '2553']]
```

How do we determine the steepest path?

We would need to compare the peak to each neighbor.

5) Find the steepest path down

```
data =
[['2537', '2483', '2475', '2480', '2518', '2532', '2480', '2478', '2431']
 ['2541', '2549', '2614', '2700', '2647', '2746', '2690', '2621', '2550']
 ['2525', '2525', '2640', '2769', '2802', '2883', '2856', '2694', '2631']
 ['2514', '2505', '2526', '2614', '2717', '2715', '2867', '2836', '2771']
 ['2506', '2482', '2480', '2528', '2518', '2561', '2586', '2662', '2654']]
 ...
]
```

We will simplify this problem.

Look at only three neighbors:

up

down

front

If peak is at location `data[r][c]`, define each above.

5) Find the steepest path down

```
data =
[['2537', '2483', '2475', '2480', '2518', '2532', '2480', '2478', '2431']
 ['2541', '2549', '2614', '2700', '2647', '2746', '2690', '2621', '2550']
 ['2525', '2525', '2640', '2769', '2802', '2883', '2856', '2694', '2631']
 ['2514', '2505', '2526', '2614', '2717', '2715', '2867', '2836', '2771']
 ...]
```

Compare and find the smallest of the three to create the next path element.

What happens if there are ties?

5) Find the steepest path down

Rules for ties.

If up == down but < front, choose randomly between them.

up = 2550
down = 2550
front = 2690

If front ties with up or down, choose front.

up = 2690
down = 2550
front = 2550

up = 2550
down = 2690
front = 2550

5) Pseudocode for find_path

```
initialize current location  🡐      (this is both a row and column)
make an empty list for path
while location is still within the list bounds
    assign up, front and down
    if (up < down and  up  < front)
        append up location to path
    else if (down < up and down < front)
        append down location to path
    else if (down == up and up < front)
        chose randomly between down and up
        append one of them to path
    else
        append front location to path
    update current location based on the chosen next location for path
return path
```

6) Pseudocode for draw_path

```
For each tuple in the path
    Using the column and row given in the tuple,  draw
    a rectangle that is one pixel wide and filled in with yellow
```

Write a function `print_rlist(rlist)` that takes a rectangular list as a parameter and prints it out as a grid.

For example, given the list defined below:

```
grid = [[8,2,7,8,2,1], [1,5,1,7,4,7],
        [5,9,6,7,3,2], [7,8,7,7,7,9],
        [4,2,6,9,2,3], [2,2,8,1,1,3]]
```

The call `print_rlist(grid)` prints the following output:

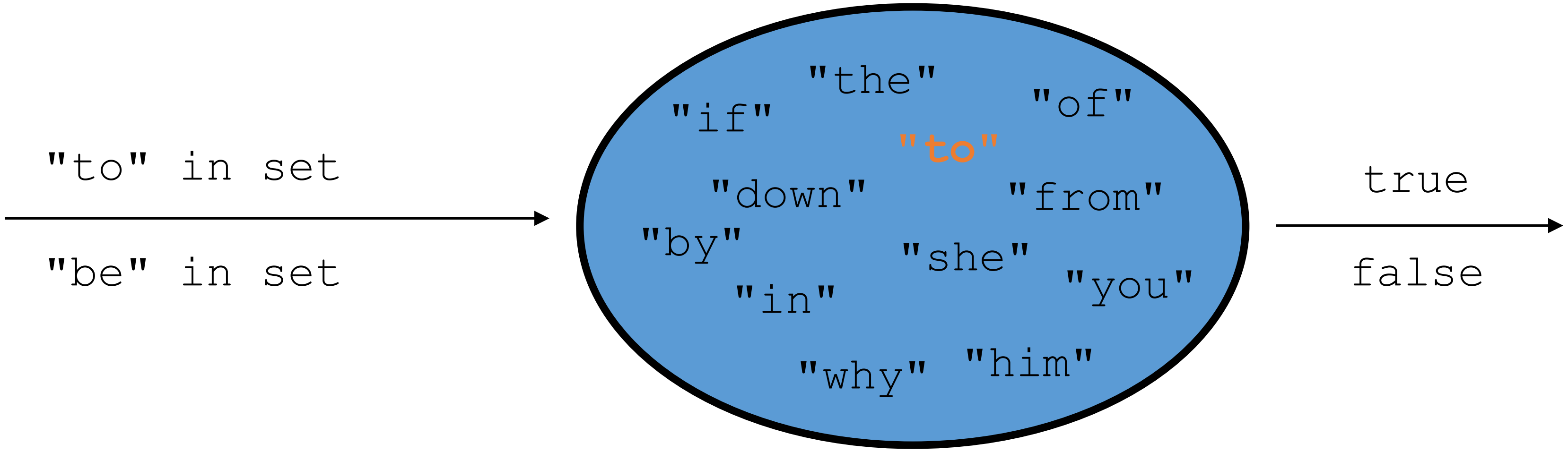
```
8 2 7 8 2 1
1 5 1 7 4 7
5 9 6 7 3 2
7 8 7 7 7 9
4 2 6 9 2 3
2 2 8 1 1 3
```

Exercise

- Write a program that counts the number of unique words in a large text file (say, *Moby Dick* or the King James Bible).
  - Store the words in a structure and report the # of unique words.
  - Once you've created this structure, allow the user to search it to see whether various words appear in the text file.
- What structure is appropriate for this problem? List? Tuple?

Sets

- **set**: A collection of unique values (no duplicates allowed)
- Sets are not indexed. They do not have an order.
- The following operations can be performed efficiently on sets:
  - add, remove, search (contains)



`"to" in set`

`"be" in set`

`"the"` `"if"` `"of"` `"to"` `"down"` `"from"` `"by"` `"she"` `"you"` `"in"` `"why"` `"him"`

`true`

`false`

set

Creating a set

- Use the function `set`:

```
a = set()
```

- Use `{`value, …, valuen`}`:

```
b = {"the", "hello", "happy"}
```

`a.add(`**val**`)`	adds element val to a
`a.discard(`**val**`)`	removes val from a if present
`a.pop()`	removes and returns a random element from a
`a - b`	returns a new set containing values in a but not in b
`a | b`	returns a new set containing values in either a or b
`a & b`	returns a new set containing values in both a and b
`a ^ b`	returns a new set containing values in a or b but not both

You can also use `in`, `len()`, etc.

Looping over a set?

- You must use a `for element in` structure loop
  - needed because sets have no indexes; can't `get` element `i`

Example:

```
    for item in a:
        print(item)
```

Outputs:

```
    the
    happy
    hello
```

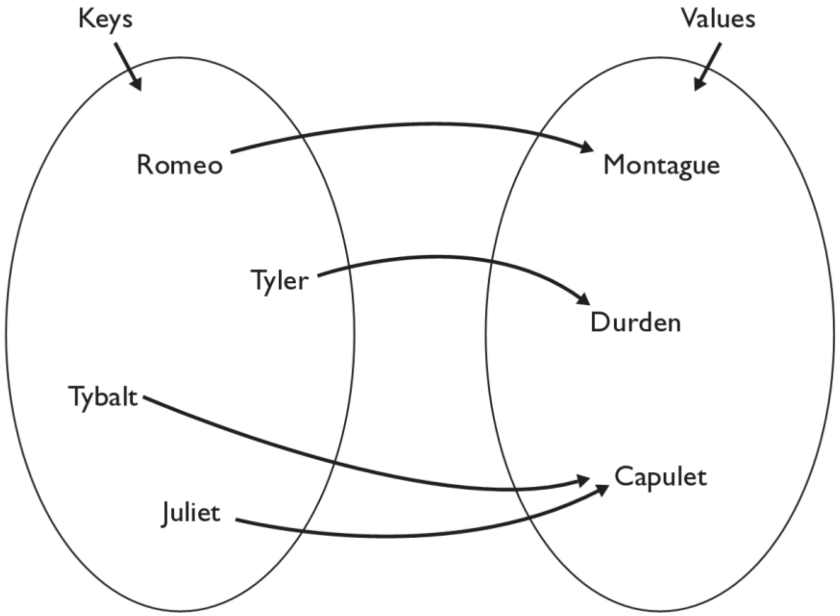
Exercise

- Write a program to <u>count the number of occurrences</u> of each unique word in a large text file (e.g. *Moby Dick*).
  - Allow the user to type a word and report how many times that word appeared in the book.
  - Report all words that appeared in the book at least 500 times.

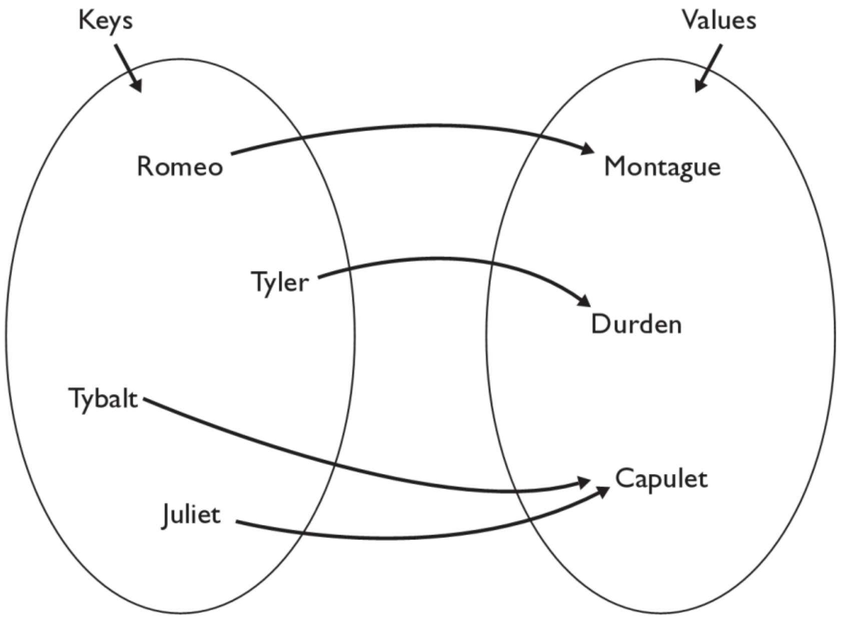
- What structure is appropriate for this problem?

Dictionaries

- **dictionary**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "map", "associative array", "hash"
- basic dictionary operations:
  - Add a mapping from a key to a value.
  - Retrieve a value mapped to a key.
  - Remove a given key and its mapped value.



Creating dictionaries



- Creating a dictionary
  - {**key** : **value, …, keyn : valuen**}

```
my_dict = {"Romeo": "Montague",
           "Tyler" :"Durden",
           "Tybalt" : "Capulet",
           "Juliet" :"Capulet" }
```

my_dict[**key**] = **value**

adds a mapping from the given key to the given value;
if the key already exists, replaces its value with the given one

Accessing values:

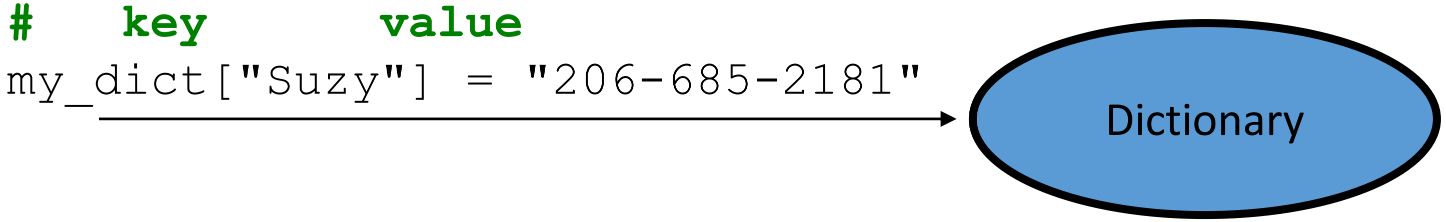
- my_dict[**key**]

returns the value mapped to the given key (error if key not found)

`my_dict["Juliet"]` **produces** `"Capulet"`

Using dictionaries

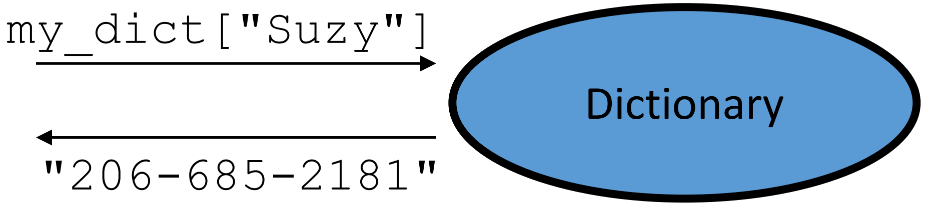
- A dictionary allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).



```
#    key         value
my_dict["Suzy"] = "206-685-2181"
```

  - Later, we can supply only the key and get back the related value:

Allows us to ask: *What is Suzy's phone number?*



```
my_dict["Suzy"]
"206-685-2181"
```

Maps and tallying

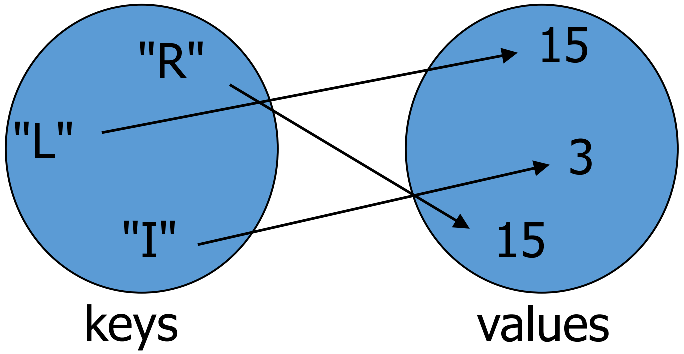
- a map can be thought of as generalization of a tallying list
  - the "index" (key) doesn't have to be an `int`
  - count digits: `22092310907`

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

```
# (Roosevelt), (L)andon, (I)ndependent
```

  - count votes: `"RLLLLLLRRRRLLLLLLRLRRIRLRRIRLLRIR"`

key	"R"	"L"	"I"
value	15	15	3



Dictionary methods

`items()`	return a new view of the dictionary’s items ((key, value) pairs)
`pop(`**key**`)`	removes any existing mapping for the given key and returns it (error if key not found)
`popitem()`	removes and returns an arbitrary (key, value) pair (error if empty)
`keys()`	returns the dictionary's keys
`values()`	returns the dictionary's values

You can also use `in`, `len()`, etc.

`items`, `keys` and `values`

- `items` function returns tuples of each key-value pair
    - can loop over the keys in a for loop

```
ages = {}
ages["Merlin"] = 4
ages["Chester"] = 2
ages["Percival"] = 12
for cat, age in ages.items():
    print(cat + " -> " + str(age))
```

- `values` function returns all values in the dictionary
    - no easy way to get from a value to its associated key(s)
- `keys` function returns all keys in the dictionary