Lecture 17

Dictionaries

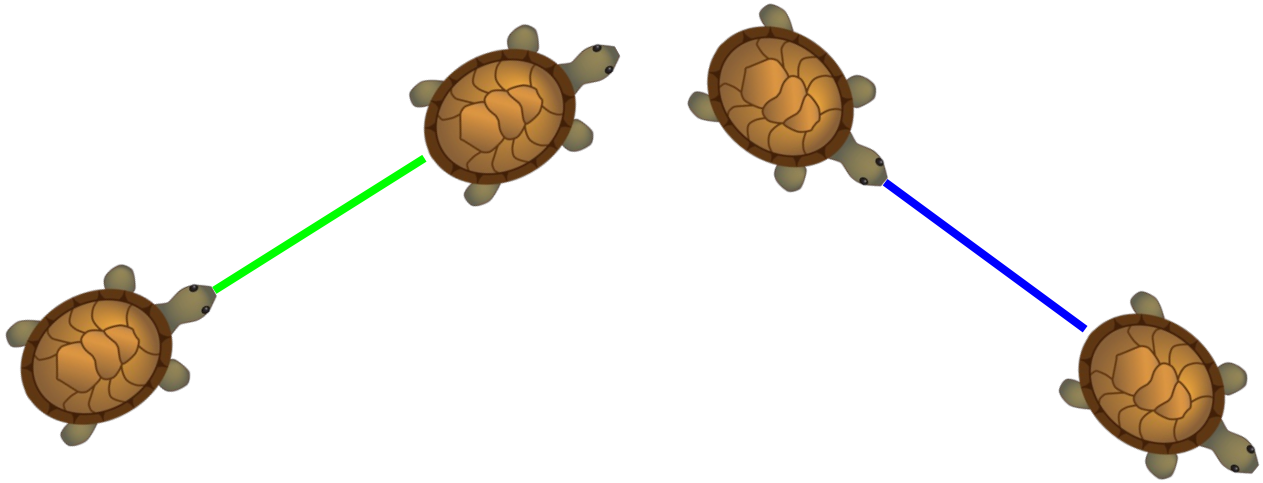
Announcements for This Lecture

Optional Videos

- View the lesson videos
  - **Videos 19.1-19.7** today
  - **Videos 20.1-20.8** Tue
  - **Videos 20.9-20.10** Thu

Assignment 4

- Should be working on it now
  - Tasks 1-3 by Saturday
  - Task 4 by Monday
  - Task 5 by Wednesday



Key-Value Pairs

- The last built-in type: dictionary (or dict)
  - One of the most important in all of Python
  - Like a list, but built of key-value pairs
- **Keys:** Unique identifiers
  - Think social security number
  - At Cornell we have netids: jrs1
- **Values:** Non-unique Python values
  - John Smith (class '13) is jrs1
  - John Smith (class '16) is jrs2

Idea: Lookup values by keys

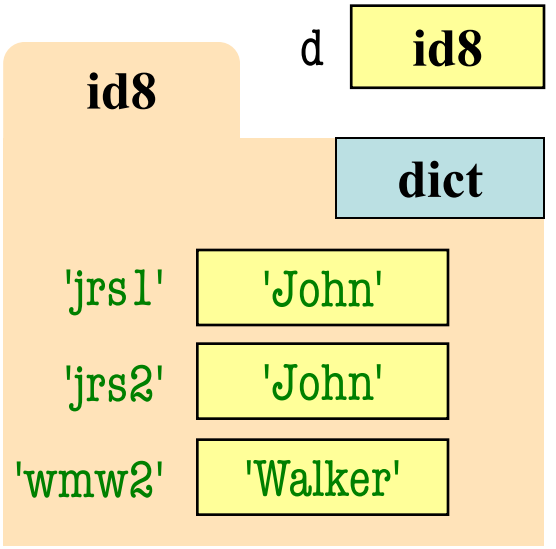
Basic Syntax

- Create with format: `{k1:v1, k2:v2, ...}`
  - Both keys and values must exist
  - **Ex:** `d={'jrs1':'John','jrs2':'John','wmw2':'Walker'}`
- **Keys** must be **non-mutable**
  - ints, floats, bools, strings, tuples
  - **Not** lists or custom objects
  - Changing a key's contents hurts lookup
- **Values** can be **anything**

Using Dictionaries (Type **dict**)

- Access elts. like a list
  - d['jrs1'] evals to 'John'
  - d['jrs2'] does too
  - d['wmw2'] evals to 'Walker'
  - d['abc1'] is an **error**
- Can test if a key exists
  - 'jrs1' in d evals to True
  - 'abc1' in d evals to False
- But cannot slice ranges!

```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```

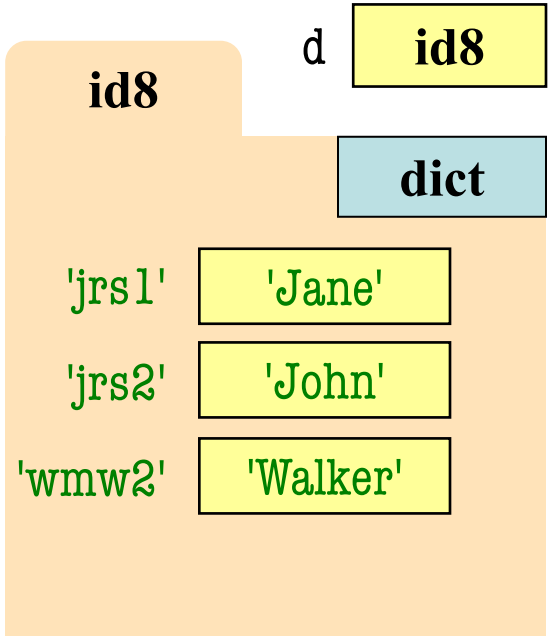


Key-Value order in folder is not important

Dictionaries Can be Modified

- **Can reassign values**
  - d['jrs1'] = 'Jane'
  - Very similar to lists
- Can add new keys
  - d['aaa1'] = 'Allen'
  - Do not think of order
- Can delete keys
  - del d['wmw2']
  - Deletes both key, value

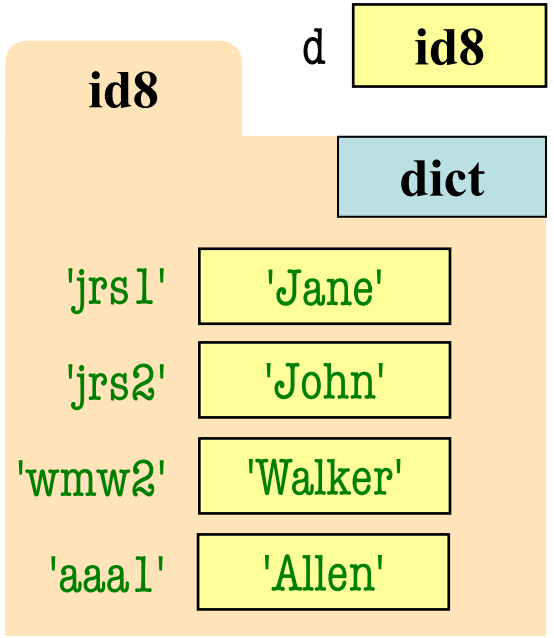
```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```



Dictionaries Can be Modified

- Can reassign values
  - d['jrs1'] = 'Jane'
  - Very similar to lists
- **Can add new keys**
  - d['aaa1'] = 'Allen'
  - Do not think of order
- Can delete keys
  - del d['wmw2']
  - Deletes both key, value

```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```

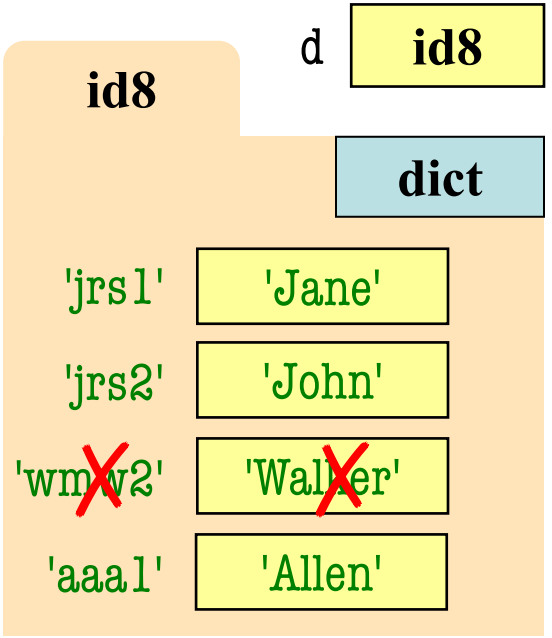


Dictionaries Can be Modified

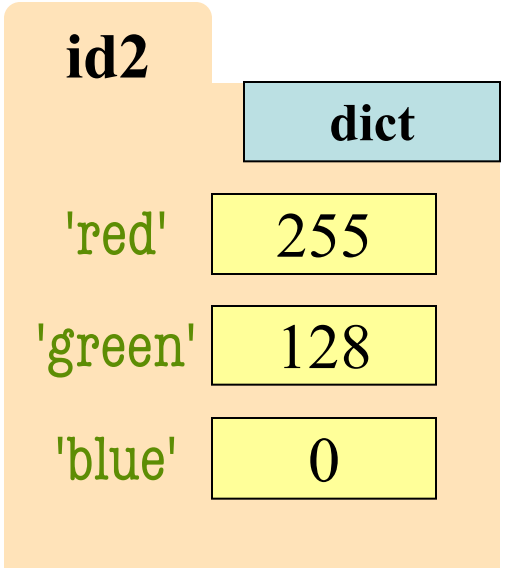
- Can reassign values
  - d['jrs1']
  - Very similar to lists
- Can add new keys
  - d['aaa1'] = 'Allen'
  - Do not think of order
- **Can delete keys**
  - del d['wmw2']
  - Deletes both key, value

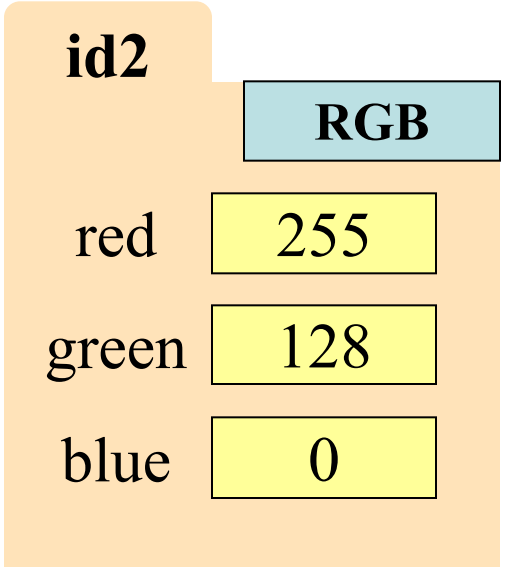
Change key = Delete + Add

jrs2':'John',
wmw2':'Walker'}



Dicts vs Objects

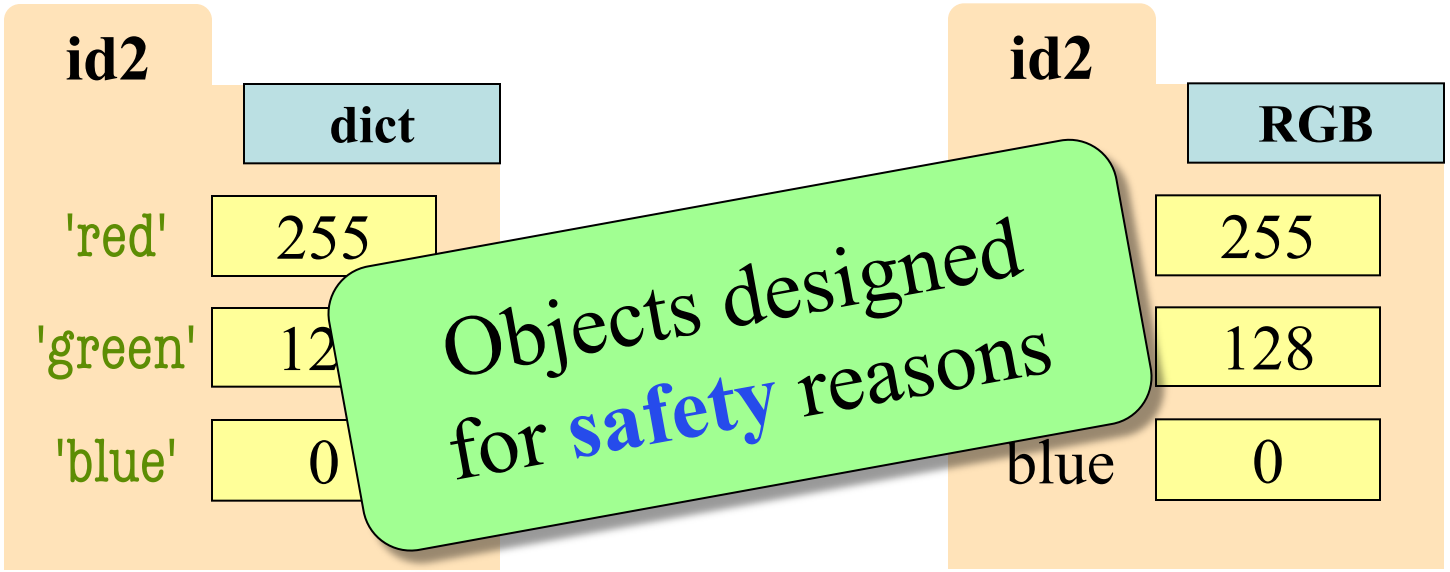




- Can add new variables
- Does not check bounds of the content variables

- Variables fixed (sort-of)
- Possibly checks bounds of the content variables

Dicts vs Objects



- Can add new variables
- Does not check bounds of the content variables

- Variables fixed (sort-of)
- Possibly checks bounds of the content variables

Question Time!

- `d = {'0':'A', 0:'B', 1:'C' }`
- What is the value `d[0]`?

A: `'A'`
B: `'B'`
C: `'C'`
D: `ERROR`
E: I don't know

Question Time!

- `d = {'0':'A', 0:'B', 1:'C' }`
- What is the value `d[0]`?

A: `'A'`
B: `'B'`
C: `'C'`
D: `ERROR`
E: I don't know

- `d = {0:'A', 1:'B', 2:'C'}`
- What is the value `d[0:2]`?

A: `{0:'A', 1:'B'}`
B: `{0:'A', 1:'B', 2:'C'}`
C: `['A','B']`
D: `ERROR`
E: I don't know

Question Time!

- `d = {'0':'A', 0:'B', 1:'C' }`
- What is the value d[0]?

A: 'A'
B: 'B'
C: 'C'
D: ERROR
E: I don't know

- `d = {0:'A', 1:'B', 2:'C'}`
- What is the value d[0:2]?

A: {0:'A', 1:'B'}
B: {0:'A', 1:'B', 2:'C'}
C: ['A','B']
D: ERROR
E: I don't know

Dictionaries: Iterable, but not Sliceable

- Can loop over a dict
  - Only gives you the keys
  - Use key to access value

```
for k in d:
    # Loops over keys
    print(k)      # key
    print(d[k])   # value
```

- Can iterate over values
  - **Method:** d.values()
  - But no way to get key
  - Values are not unique

```
# To loop over values only
for v in d.values():
    print(v)      # value
```

Other Iterator Methods

- **Keys:** d.keys()
  - Sames a normal loop
  - Good for *extraction*
  - keys = list(d.keys())

```
for k in d.keys():
    # Loops over keys
    print(k)      # key
    print(d[k])   # value
```

- **Items:** d.items()
  - Gives key-value pairs
  - Elements are tuples
  - Specialized uses

```
for pair in d.items():
    print(pair[0]) # key
    print(pair[1]) # value
```

Other Iterator Methods

- **Keys:** d.keys()
  - Sames a normal loop
  - Good for *extraction*
  - keys

So mostly like loops over lists

- **Items:** d.items()
  - Gives key-value pairs
  - Elements are tuples
  - Specialized uses

```
for k in d.keys():
    # Loops over keys
    print(k)      # key
    print(d[k])   # value
```

```
for pair in d.items():
    print(pair[0]) # key
    print(pair[1]) # value
```

Dictionaries and Fruitful Functions

- Dictionaries handled similar to lists
  - Go over dictionary (keys) with *for-loop*
  - Use *accumulator* to gather the results
- Only difference is how to access value
  - Remember, loop variable is **keys**
  - Use **keys** to access the **values**
  - But otherwise the same

Dictionary Loop with Accumulator

```
def max_grade(grades):
    """Returns max grade in the grade dictionary

    Precondition: grades has netids as keys, ints as values"""
    maximum = 0                    # Accumulator


    # For each student
        # if student grade exceeds maximum
            # make that grade the new maximum


    return maximum
```

Dictionary Loop with Accumulator

```
def max_grade(grades):
    """Returns max grade in the grade dictionary

    Precondition: grades has netids as keys, ints as values"""
    maximum = 0                    # Accumulator
    # Loop over keys
    for k in grades:
        if grades[k] > maximum:
            maximum = grades[k]

    return maximum
```

Another Example

```
def netids_above_cutoff(grades,cutoff):
    """Returns list of netids with grades above or equal cutoff

    Precondition: grades has netids as keys, ints as values.
    cutoff is an int."""
    result = []                          # Accumulator

    # For each student
        # if student's grade is above cutoff
            # add student to the result

    return result
```

Another Example

```
def netids_above_cutoff(grades,cutoff):
    """Returns list of netids with grades above or equal cutoff

    Precondition: grades has netids as keys, ints as values.
    cutoff is an int."""
    result = []                          # Accumulator

    for k in grades:
        if grades[k] >= cutoff:
            result.append(k)             # Add key to the list result

    return result
```

Dictionaries and Mutable Functions

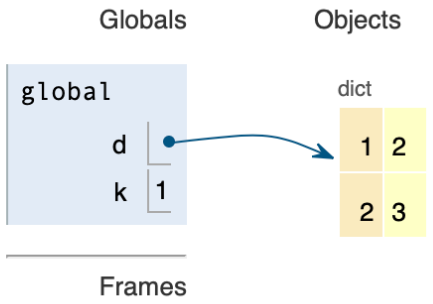
- Restrictions are different than list
  - Okay to loop over dictionary to change
  - You are looping over *keys*, not *values*
  - Like looping over positions
- But you **may not add or remove** keys!
  - Any attempt to do this will fail
  - Have to create a key list if you want this

A Subtle Difference

```
1
2 d = {1:2}
3 for k in d.keys():
4     d[k+1] = d[k]+1
```

<< First | < Back | Program terminated | Forward > | Last >>

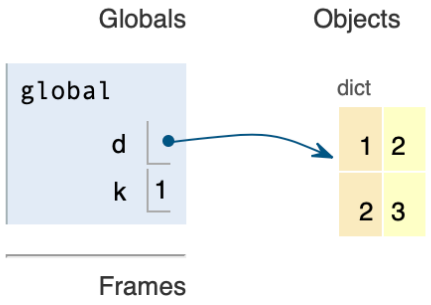
RuntimeError: dictionary changed size during iteration




```
1
2 d = {1:2}
3 for k in list(d.keys()):
4     d[k+1] = d[k]+1
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed
next line to execute



But This is Okay

```
def add_bonus(grades,bonus):
    """Gives bonus points to everyone in grades

    Precondition: grades has netids as keys, ints as values.
    bonus is an int."""
    # No accumulator.  This is a procedure


    for student in grades:
        # Modifies the dictionary, but does not change keys
        grades[student] = grades[student]+bonus
```

Another Example

```
def merge(dict1,dict2):
    """Updates dict1 to include the contents of dict2.

    If a key is already in dict1, then assign the max of dict1, dict2

    Precondition: dict1, dict2 have str as keys, int as values."""
    for key in dict2:
        # Looping over dict2; safe to modify dict1
        if key in dict1:
            dict1[key] = max(dict1[key],dict2[key])
        else:
            dict1[key] = dict2[key]
```

Nesting Dictionaries

- Remember, values can be anything
  - Only restrictions are on the keys
- Values can be lists (**Visualizer**)
  - `d = {'a':[1,2], 'b':[3,4]}`
- Values can be other dicts (**Visualizer**)
  - `d = {'a':{'c':1,'d':2}, 'b':{'e':3,'f':4}}`
- Access rules similar to nested lists
  - **Example:** `d['a']['d'] = 10`

Example: JSON File

```
{
   "wind" : {
      "speed" : 13.0,
      "crosswind" : 5.0
      },
   "sky" : [
      {
         "cover" : "clouds",
         "type" : "broken",
         "height" : 1200.0
      },
      {
         "type" : "overcast",
         "height" : 1800.0
      }
   ]
}
```

Nested Dictionary

Nested List

Nested Dictionary

- **JSON:** File w/ Python dict
  - Actually, minor differences
- weather.json:
  - Weather measurements at Ithaca Airport (2017)
  - **Keys**: Times (Each hour)
  - **Values**: Weather readings
- This is a *nested* JSON
  - Values are also dictionaries
  - Containing more dictionaries
  - And also containing lists

JSONs vs Dictionaries

- JSONs *look* like dictionaries, but are not same
  - JSONs are strings (to send over internet)
  - Dictionaries are a type with its own operations
- But you can go back and forth between them

```
>>> import json            # The json module in Python
>>> d = json.loads(s)      # Converts JSON s to dict d
>>> s = json.dumps(d)      # Converts dict d to JSON s
```

- So we often think of the two as the same
  - JSON is to dict as CSV is to nested lists

Navigating this File

```
{
   "wind" : {
      "speed" : 13.0,
      "crosswind" : 5.0
      },
   "sky" : [
      {
         "cover" : "clouds",
         "type" : "broken",
         "height" : 1200.0
      },
      {
         "type" : "c
         "height" : 1800.0
      }
   ]
}
```

Access this value

- Let d be the dict to left
- Need to access a value
- How do we do it?

A: d['height']
B: d['height']['sky']
C: d['sky']['height']
D: d['sky'][0]['height']
E: I don't know

Navigating this File

```
{
   "wind" : {
      "speed" : 13.0,
      "crosswind" : 5.0
      },
   "sky" : [
      {
         "cover" : "clouds",
         "type" : "broken",
         "height" : 1200.0
      },
      {
         "type" : "c
         "height" : 1800.0
      }
   ]
}
```

This is a list

Access this value

- Let d be the dict to left
- Need to access a value
- How do we do it?

A: d['height']
B: d['height']['sky']
C: d['sky']['height']
D: d['sky'][0]['height']
E: I don't know

Dictionaries and Recursion

- Dictionaries are **not sliceable**
  - Makes it difficult to do divide and conquer
  - So rare to be used in recursion by itself
  - Often the *answer* to a recursion, not the *input*
- However, the **key list** is sliceable
  - Can recurse on key list, not the dict
  - This requires a helper function
  - Helper is recursive, not the main function

The Recursive Version

```
def max_grade(grades):
    """Returns max grade in the grade dictionary

    Precondition: grades has netids as keys, ints as values"""

    # WE CANNOT SLICE A DICTIONARY
    # We need to pull out keys and use a recursive helper
    netids = list(grades.keys())

    return max_grade_helper(netids,grades)
```

The Recursive Version

```
def max_grade_helper(netids,grades):
    """Returns max grade among given netids

    Precond: netids a list of keys in grades, grades a dict w/ int values"""
    # Process small data
    if len(netids) <= 1:
        return grades[netids[0]] if len(netids) == 1 else 0

    # Break it up into left and right



    # Combine the answers
```

The Recursive Version

```
def max_grade_helper(netids,grades):
    """Returns max grade among given netids

    Precond: netids a list of keys in grades, grades a dict w/ int values"""
    # Process small data
    if len(netids) <= 1:
        return grades[netids[0]] if len(netids) == 1 else 0

    # Break it up into left and right
    left  = grades[netids[0]]
    right = max_grade_helper(netids[1:],grades)

    # Combine the answers
```

The Recursive Version

```
def max_grade_helper(netids,grades):
    """Returns max grade among given netids

    Precond: netids a list of keys in grades, grades a dict w/ int values"""
    # Process small data
    if len(netids) <= 1:
        return grades[netids[0]] if len(netids) == 1 else 0

    # Break it up into left and right
    left  = grades[netids[0]]
    right = max_grade_helper(netids[1:],grades)

    # Combine the answers
    return max(left,right)
```