Module 19

Dictionaries

Key-Value Pairs

- Introducing last new 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

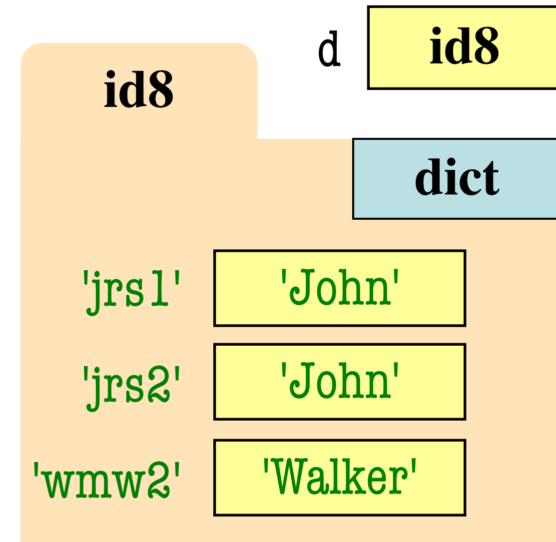
Basic Syntax

- 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 = {'js1':'John','js2':'John',
     'wmw2':'Walker'}
```

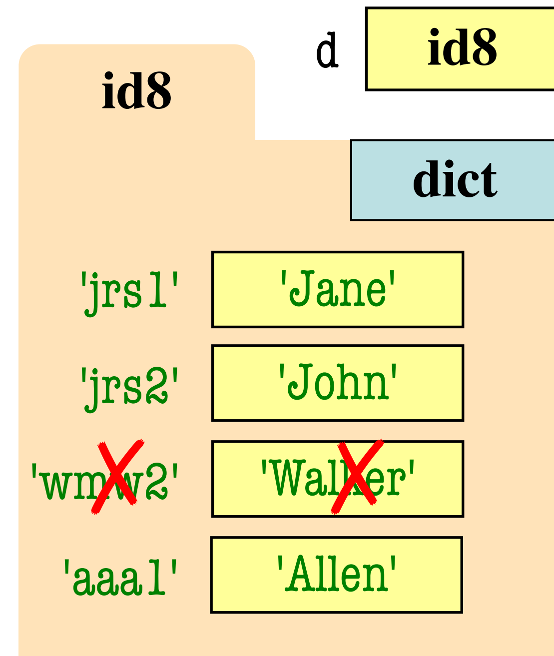


Key-Value order in folder is not important

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
  - **Change:** delete + add

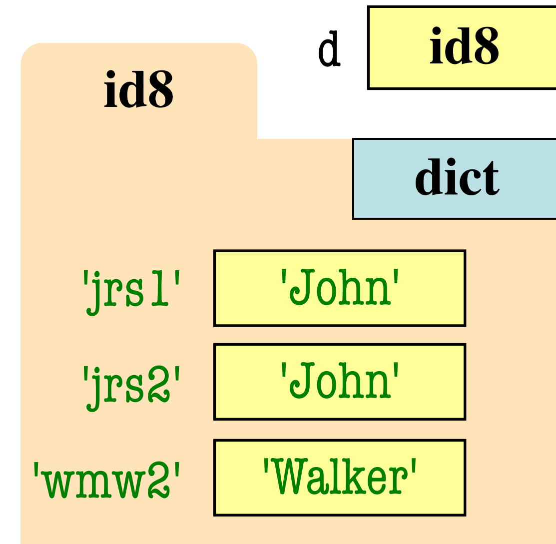
```
d = {'jrs1':'John','jrs2':'John',
     'wmw2':'Walker'}
```



Dictionaries are Represented as Folders

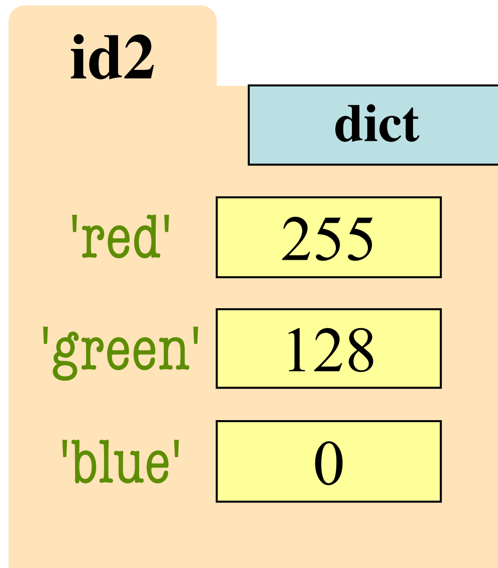
- Need because mutable!
  - Values in variables
  - Keys are off to left
- Looks like objects
  - Esp. if string keys
  - But note the quotes
  - Cannot access with dot
- More flexible type

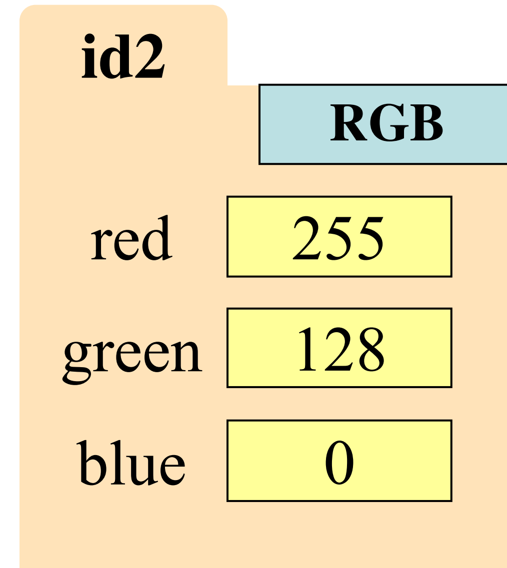
```
d = {'js1':'John','js2':'John',
     'wmw2':'Walker'}
```



Key-Value order in folder is not important

Dicts vs Objects

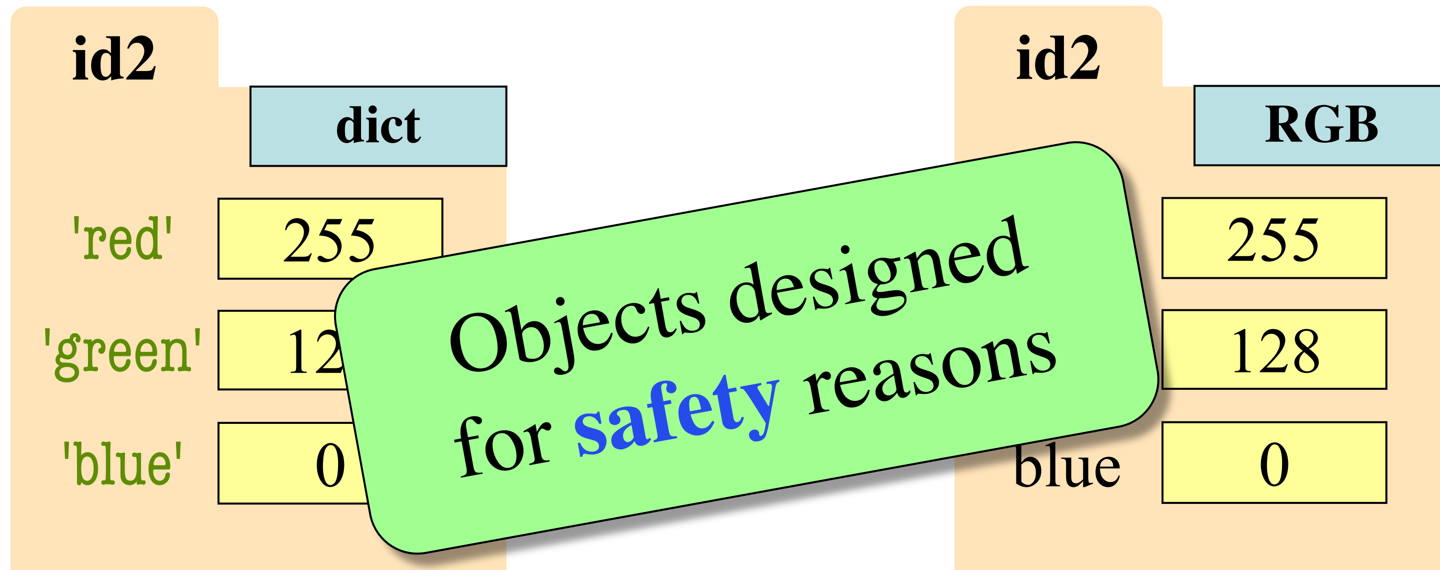




- Can add new variables
- Does not check bounds of the content variables

- 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

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

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()
  - No different normal loop
  - But 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()
  - Returns key-value pairs
  - Elements are tuples
  - Specialized uses

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

Relationship to Standard Lists

- Functions on dictionaries 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

Simple Example

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

    return result
```

Relationship to Standard Lists

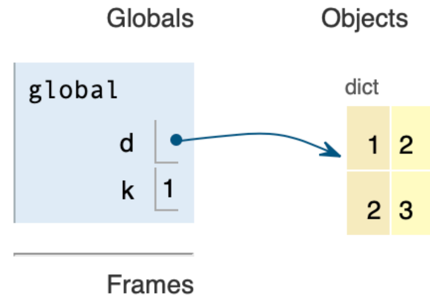
- Restrictions are different than list
  - Okay to loop over dictionary to change
  - You are looping over *keys*, not *values*
  - Like looping over positions
- But you **may not add or remove** keys!
  - Any attempt to do this will fail
  - Have to create a key list if you want to do

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

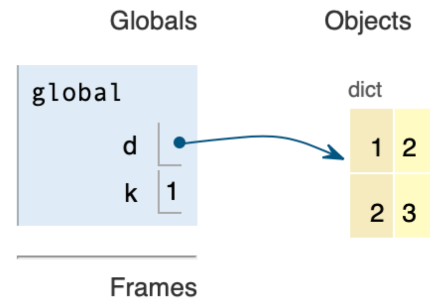



```
1
2  d = {1:2}
3  for k in list(d.keys()):
4      d[k+1] = d[k]+1
```

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

line that has just executed

next line to execute



But This is Okay

```
def give_extra_credit(grades,netids,bonus):
    """Gives bonus points to everyone in sequence netids

    Precondition: grades has netids as keys, ints as values.
    netids is a sequence of strings that are keys in grades
    bonus is an int."""
    # No accumulator.  This is a procedure

    for student in grades:
        if student in netids:              # Test if student gets a bonus
            grades[student] = grades[student]+bonus
```

Could also loop over **netids**

Keyword Expansion

- Last use of dicts is an advanced topic
  - But will see if read Python code online
  - Variation of tuple variation
- An Observation:
  - Functions can be called with assignments
  - These assign parameters to specific variables
  - Can we do this with a *single* argument: a dictionary?
- Purpose of keyword expansion: **kw
  - But only works in certain **contexts**

Tuple Expansion Example

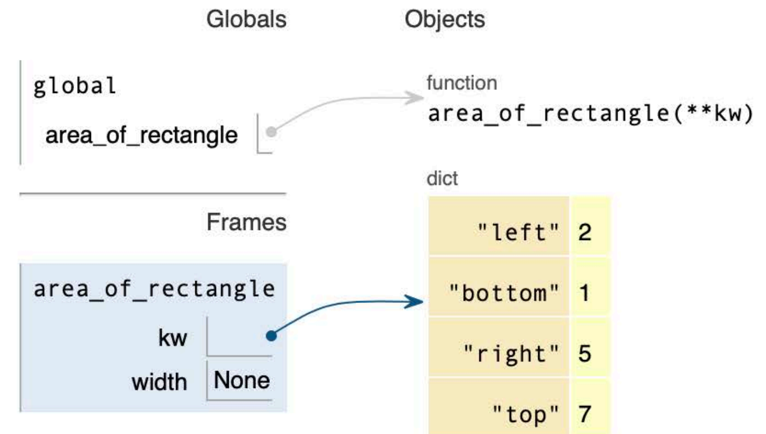
```
>>> def add(x, y)
...       """Returns x+y """
...       return x+y
...
>>> d = {'x':1,'y':2}
>>> add(**d)                    # Assigns to variable with name
3
>>> d = {'x':1,'y':2,'z':3}     # Cannot have extra “variables”
>>> add(**d)                    # Can only have less if optional
ERROR
```

Have to use in **function call**

Also Works in Function Definition

```
1  def area_of_rectangle(**kw):
2      """Returns the area of the specified rectang
3
4      Parameters: left, right, width, center, bott
5      Precondtion: parameters all int or float.  (
6
7      # Compute the width of the rectangle
8      width = None
9      if 'left' in kw and 'right' in kw:
10         width = kw['right']-kw['left']
11     elif 'width' in kw:
12         width = kw['width']
13     elif 'center' in kw:
14         if 'left' in kw:
15             width = 2*(kw['center']-kw['left'])
16         elif 'right' in kw:
17             width = 2*(kw['right']-kw['center'])
18     assert width != None, 'There were not enougł
19
20     # Compute the height of the rectangle
```

<< First | < Back | Step 5 of 31 | Forward > | Last >>



Also Works in Function Definition

```
def area_of_rectangle(**kw):
    """Returns the area of the specified rectangle.

    Params: left,right,width,center,bottom,top,height,middle
    Prec: params all int or float.  Can compute width, height"""
    width = None
    if 'left' in kw and 'right' in kw:
        width = kw['right']-kw['left']
    elif 'width' in kw:
        width = kw['width']
    elif 'center' in kw:
        if 'left' in kw:
            width = 2*(kw['center']-kw['left'])
        elif 'right' in kw:
            width = 2*(kw['right']-kw['center'])
```

Automatically converts all arguments to a dictionary

And similarly for the height

When is This Useful?

- When have a lot of optional arguments
  - GUI libraries infamous for this: TKinter, Kivy
  - Have to specify lots of details for each widget
  - Where located, color, size, and so on
- Also when want flexibility (like example)