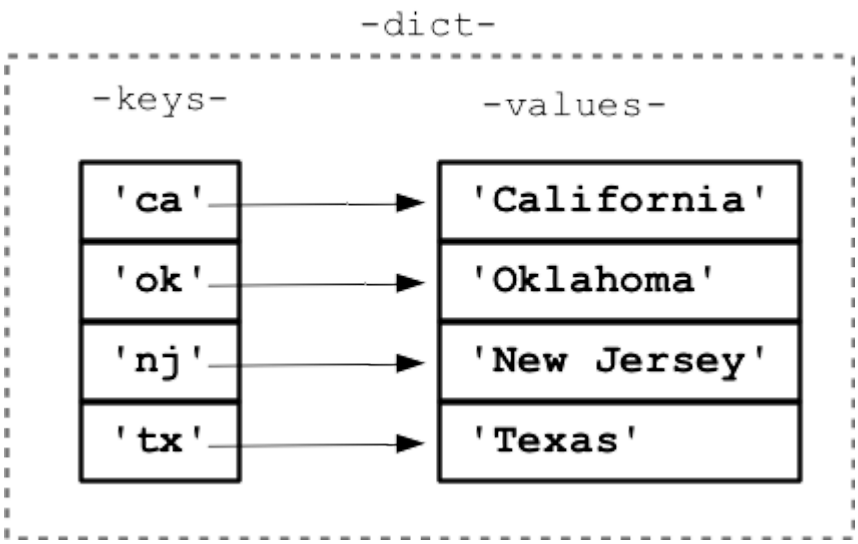
Python Dict

The dict "dictionary" type is very important because it takes in disorganized data, organizes it, and it is **fast**. The dict here is based on the "hash table" data structure. You will learn how to build and analyze a hash table in CS106B, here we're happy to just use them. Many important algorithms leverage hash tables in some way since they can handle large data sets and remain fast. 'dict' is the name of the Python dict type, so we'll use just 'd' as a generic variable name.

```
d = {}                      # Create empty dict
d['ca'] = 'California'      # 1. Set key/value pairs into
dict
d['ok'] = 'Oklahoma'
d['nj'] = 'New Jersey'
d['tx'] = 'Texas'
val = d['nj']               # 2. Retrieve value by key
val = d['xx']               # fails with KeyError
check = 'nj' in d           # 3. in check -> True
```



1. `d['ca'] = 'California'` - with equal-sign: d[key] = xxx creates a key/value pair in the dict. If a pair was in there already, it is overwritten.

2. `val = d['nj']` - referring to d[key] retrieves the value for that key, or is a KeyError. Code needs to check that the key is in before doing [ ] retrieval.

3. `if 'nj' in d:` - use **in** to check if a key is in the dict or not. This sort of 'in' works for lists too, but for dicts it's much faster.

Key point: dicts (hash tables) are **fast**. Even if the dict has 1 million key/value pairs in it, performing the get/set/in of single key is very fast. If the dict grows to hold 10 million key/value pairs, the speed is largely unchanged.

Strategy: therefore if we are reading a file or a network taking in disorganized data, load the data into a dict choosing the key and value definitions we want to output. The data will become organized by that key, even though it was in random order as it was read.

The type used as key should be "immutable", often a string or int (or tuple when we get to those).

Dict Counting

Here is the canonical logic to load up a dict - in this example the code counts the number of occurrences of each word, but many useful dict operations will follow this basic loop/in/out pattern.

```
def strs_counts(strs):
    """
    Given a list of strings, return a 'counts' dict of
    str->count key/value pairs
    """
    counts = {}
    for s in strs:
        # fix up not-in case
        if not s in counts:
            counts[s] = 0
        # invariant: at this line, s is in
        counts[s] += 1
        # alternate 'else' solution:
        #if not s in counts:
        #    counts[s] = 1
        #else:
        #    counts[s] += 1
    return counts
```

Getting Data out of Dict

1. `len(d)` - as you would guess, the number of key/value pairs in the dict

2. `d.get(key)` - retrieves the value for a key, but if the key is not there, returns None by default (vs. throwing an error like [ ]). A 2 parameter form `d.get(key, missing-value)` specifies what value to return if the key is missing. This forms an alternative to writing if/in logic to check if the key is in.

3. `d.keys()` - returns an iterable of all the keys in dict (in a random order). Can loop over this to get each key. The keys alone, no values.

4. `d.values()` - returns an iterable of all the values in dict (in a random order). Can loop over this to get each value.

5. `d.items()` returns an iterable of the key,value pairs. This works with a particular sort of double-variable loop, see below. If you want to look at all of the key,value pairs, this is the most direct way.

The most common pattern for looping over a dict, using sorted() function which returns a linear collection sorted into increasing order:

```
def print_counts2(counts):
    # sort the keys for nicer output
    for key in sorted(counts.keys()):
        print(key, counts[key])
```

The double-variable key,value loop (more detailed explanation in the tuples section below)

```
def print_counts3(counts):
    # key,value loop over .items()
    # unsorted
    for key,value in counts.items():
        print(key, value)
```

Tuples and Dicts

One handy use of tuples is the dict.items() function, which returns the entire contents of the dict as an list of len-2 (key, value) tuples.

```
>>> d = {'a':1, 'd':4, 'c':2, 'b':3}
>>> d
{'a': 1, 'd': 4, 'c': 2, 'b': 3}
>>> d.items()
dict_items([('a', 1), ('d', 4), ('c', 2), ('b', 3)])
# (key, value) tuples
>>> sorted(d.items())   # same tuples, sorted by key
[('a', 1), ('b', 3), ('c', 2), ('d', 4)]
```

Sorting of tuples goes left-to-right within each tuple -- first sorting by all the [0] values across the tuples, then by [1], and so on.

Once all the data has been loaded into a dict, it's natural to have a process-all-the-data phase. This can be written as a loop over the above d.items()

list. The loop syntax below takes one tuple off the list for each iteration, setting the two variable, key and value each time:

```
# Example: for loop setting key/value for each
iteration
for key,value in d.items():
    # use key and value in here
```

This is a special version of the for loop, where there are multiple variables, and the number of variables matches the size of a tuples coming off the list. The above example, looping key,value over dict.items() is probably the most common use of this multiple-variable variant of the for loop.