Dictionaries and Sorting

Python 2

home (../handouts.html)

Introduction: Dictionaries for Paired Data

dicts pair unique keys with associated values.

Much of the data we use tends to be *paired*:

- **companies** paired with **annual revenue** for each
- **employees** with **contact information** for each
- **students** with **grade point averages**
- **dates** with the **high temperature** for each
- **web pages** with the **number of times** each was accessed

To store paired data, we use a Python container called a *dict*, or dictionary. The dict contains *keys* paired with *values*.

```
customer_balances = { 'jk125': 493.95,      # spacing for clarity
                      'xx3':   122.03,
                      'jp9':   23238.72 }

print customer_balances          # { 'jk125': 493.95, 'xx3': 122.03, 'jp9'
print type(customer_balances)    # <type 'dict'>
```

dicts have some of the same features as other containers. When standard container operations are applied, the *keys* are used:

```
print len(customer_balances)        # 3 keys in dict

print 'xx3' in customer_balances    # True:  the key is there

for yyy in customer_balances:
    print yyy                       # prints each key in customer_balances

print customer_balances['jp9']      # 23238.72
```

Objectives for the Unit: Dictionaries

This *key/value pairs* container allows us to summarize data in powerful ways:

- Use a dict to store and report a value for each a unique collection of keys, such as date to price, contract to expiration, device to uptime, etc.
- Use a dict to take a "running sum" or "running count", such as summing up revenue by month (based on daily revenue data), counting the number of times a web page was visited, counting the number of error requests each month, etc.

Summary for Object: Dictionary (dict)

A dictionary (or *dict*) is an *unordered collection* of *unique key/value pairs* of objects.

The keys in a dict are *unordered* and *unique*. In this way it is like a **set**.

A dict key can be used to obtain the assocaiated *value* ("addressible by key"). In this way it is like a **list** or **tuple** (which use an *integer index* to obtain a value).

initialize a dict

```
mydict = {}                          # empty dict

mydict = {'a':1, 'b':2, 'c':3}       # dict with str keys and int values
```

add a key/value pair to a dict

```
mydict['d'] = 4                  # setting a new key and value

print mydict                     # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

read a value based on a key

```
dval = mydict['d']              # value for 'd' is 4

xxx = mydict['c']               # value for 'c' is 3
```

Note in the standard container features below, only the *keys* are used:

loop through a dict and read keys and values

```
mydict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key in mydict:
    print key          # prints 'a', then 'c', then 'b', then 'd'
```

check for key membership

```
mydict = {'a': 1, 'b': 2, 'c': 3}

if 'a' in mydict:
    print "'a' is a key in mydict"
```

sort a dict's keys

```
mydict = {'xenophon': 10, 'abercrombie': 5, 'denusia': 3}

mykeys = sorted(mydict)         # ['abercrombie', 'denusia', 'xenophon']
```

retrieve list of keys or list of values

```
mydict = {'a': 1, 'b': 2, 'c': 3}

yyy = mydict.keys()                  # ['a', 'c', 'b']

zzz = mydict.values()                # [1, 3, 2]
```

Summary dict Method: get()

The **dict get()** method returns a value based on a key. An optional 2nd

argument provides a *default* value if the key is missing.

```
mydict = {'a': 1, 'b': 2, 'c': 3}

xx = mydict.get('a', 0)           # 1 (key exists so paired value is retu

yy = mydict.get('zzz', 0)         # 0 (key does not exist so default valu
```

The default value is your choice.

This method is sometimes used as an alternative to testing for a key in a dict before reading it -- avoiding the **KeyError** exception that occurs when trying to read a nonexistent key.

Summary dict Methods: keys() and values()

keys() returns a list of keys; *values()* returns a list of values.

These methods are used for advanced manipulation of a **dict**.

dict keys() method returns a list of keys

```
mydict = {'a': 1, 'b': 2, 'c': 3}

yyy = mydict.keys()                  # ['a', 'c', 'b']
```

We might want a list of a dict's keys for advanced manipulation of a dictionary. We can retrieve the list for sorting, for testing for membership, for checking length, BUT these can also accomplished by using the dictionary directly: a dict used in a "listy" context always works with its keys.

dict values() method returns a list of values in the dict

```
zzz = mydict.values()                # [1, 3, 2]
```

This method, like, **keys()** is also less often used -- to test for membership or frequency among values, for example.

Summary Exception: KeyError

The **KeyError** exception indicates that the requested key does not exist in the dictionary.

```
mydict = {'a': 1, 'b': 2, 'c': 3}

xx = mydict['a']                    # 1

yy = mydict['NARNAR']               # KeyError:  'NARNAR'
```

The above code results in this exception:

```
Traceback (most recent call last):
  File "./keyerrortest.py", line 7, in
    yy = mydict['NARNAR']
KeyError: 'NARNAR'
```

As you can see, the line of code and the key involved in the error are displayed. If you get this error the debugging procedure should be to check to see first whether the key seems to be in the dict or not; if it is, you may want to check the type of the object, since string **'1'** is not the same as int **1**. String case also matters when Python is looking for a key -- the object value must match the dict's key exactly.

One way to handle an error like this is to test the dict ahead of time using the **in** operator:

```
mydict = {'a': 1, 'b': 2, 'c': 3}

if 'NARNAR' not in mydict:
    yy = 0
else:
    yy = mydict['NARNAR']    # else: block only executed
                             # if key NARNAR exists in the dict
```

Another approach is the use the **dict get()** method:

```
mydict = {'a': 1, 'b': 2, 'c': 3}

yy = mydict.get('a', 0)            # 1

zz = mydict.get('zzz', 0)          # 0
```

Summary Task: read and write to a dict

Subscript syntax is used to add or read key/value pairs. The dict's *key* is the key!

Note well: *the syntax is the same* for setting a key/value pair *or* getting a value based on a key.

Setting a key/value pair in a dict

```
mydict = {}
mydict['a'] = 1         # set a key and value in the dict
mydict['b'] = 2         # same

print mydict            # {'a': 1, 'b': 2}
```

Getting a value from a dict based on a key

```
val  = mydict['a']       # get a value using a key:  1
val2 = mydict['b']       # get a value using a key:  2
```

Summary Task: Looping through a Dict with *for*

Looping through a dict means looping through its *keys*.

```
mydict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key in mydict:
    print key        # prints 'a', then 'c', then 'b', then 'd'
```

As with all standard container features (e.g. **in**, **for**, **sorted()**) the **in** operator test the **keys** of the dict.

Of course having a key means we can get the value -- here we loop through and print each key *and* value in the dict:

```
mydict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

for key in mydict:
    print "key: {}; value: {}".format(key, mydict[key])

    ### key: a; value 1
    ### key: c; value 3
    ### key: b; value 2
    ### key: d; value 4
```

Summary Task: Checking for Membership in a Dict with *in*

Checking membership in a dict means checking for the presence of a *key*.

```
mydict = {'a': 1, 'b': 2, 'c': 3}

if 'a' in mydict:
    print "'a' is a key in mydict"
```

As with all standard container features (e.g. **in**, **for**, **sorted()**) the **in** operator test the **keys** of the dict.

Summary Task: Dictionary Size with len()

len() counts the *pairs* in a dict.

```
mydict = {'a': 1, 'b': 2, 'c': 3}

print len(mydict)               # 3 (number of keys in dict)
```

Summary Task: Obtaining List of Keys, List of Values, List of key/value Items

keys(), **values()** return a list of objects; **items()** returns a list of *2-element tuples*.

keys(): return a list of keys in the dict

```
mydict = {'a': 1, 'b': 2, 'c': 3}

these_keys = mydict.keys()
print these_keys                        # ['a', 'c', 'b']
```

Of course once we have the keys, we can loop through and get the value for each key.

values(): return a list of values in the dict

```
these_values = mydict.values()
print these_values                    # [1, 3, 2]
```

The values cannot be used to get the keys - it's a one-way lookup from the keys. However, we might want to check for membership in the values, or sort

the values, or some other less-used approach.

The dict items() method: pairs as a list of 2-element tuples

```
these_items = mydict.items()
print these_items                    # [('a', 1), ('c', 3), ('b', 2)]
```

There are three elements here: three tuples. Each tuple contains two elements: a key and value pair from the dict. There are a number of reasons we might wish to use a structure like this -- for example, to sort the dictionary and store it in sorted form. As you know, a dictionary's keys are unordered, but a lists are not. A list of tuples can also be manipulated in other ways pertaining to a list. It is a convenient structure that is preferred by some developers as an alternative to working with the keys.

Review Summary Task: Sorting a Container with sorted()

With a list, tuple or set, **sorted()** returns a list of sorted elements

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

slist = sorted(namelist)          # ['avram', 'jo', 'michael', 'pete', 'z
```

Remember that no matter what container is passed to **sorted()**, the function returns list -- even a string!

Summary Task: Reversing a Sort

reverse=True, a special arg to **sorted()**, reverses the order of a sort.

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

slist = sorted(namelist, reverse=True)     # ['zeb', 'pete', 'michael', 'j
```

reverse is called a *keyword argument* -- it is no different from a regular *positional* arguments we have seen before; it is simply notated differently.

Summary Task: Sorting a Dict (sorting its *Keys*)

sorted() returns a sorted list of a dict's *keys*

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
keys = bowling_scores.keys()
keys = sorted(keys)
print keys                          # [ 'janice', 'jeb', 'mike', 'zeb' ]
for key in keys:
  print key + " = " + str(bowling_scores[key])
```

Or, we can even loop through the sorted dictionary keys directly:

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
for key in sorted(bowling_scores.keys()):
  print key + " = " + str(bowling_scores[key])
```

Practical: Build Up a Dict from Two Fields in a File

As with all containers, we loop through a data source, select and add to a dict.

```
ids_names = dict()                                  # initialize an empty
for line in open('student_db.txt'):
    id, address, city, state, zip = line.split(':') # note "multi-target a
    ids_names[id] = state                           # key id is paired to

print "here are ids and names from the students.txt file:  "
for id in ids_names:
    print "id " + id + " is from this state:  " + ids_names[id]

print "here is the state for student 'jb29':  "
print ids_names['jb29']                             #  NJ
```

Practical: Build a "Counting" or "Summing" Dictionary

We can use a dict's keys and associated values to create an aggregation (correlating a sum or count to each of a collection of keys).

```
state_count = dict()                                  # initialize an empt
for line in open('/Users/dblaikie/Desktop/python_data/student_db.txt'):
    items = line.split(':')
    state = items[3]
    if state not in state_count:
        state_count[state] = 0
    state_count[state] = state_count[state] + 1

print "here is the count of states from the students.txt file:  "
for state in state_count :
    print "{}:  {} occurrences".format(state, state_count[state])

print "here is the count for 'NY':  "
print state_count['NY']                               # 4
```

Practical: Build and Read a Dict of Lists

A list can be added to a dict just like any other object.

Here we're keying a dict to student id, and setting a list of the remaining

elements as its value:

```
ids_data = dict()                       # initialize an empty dict
for line in open('student_db.txt'):
    item_list = line.split(':') # note "multi-target assignment" of 5 elem
    id = item_list[0]
    data = item_list[1:]
    ids_data[id] = data                 # key id is paired to student's stat

print "here is the data for id 'jb29':  ", ids_names['jb29']    #
```

Practical: Working with dict items()

dict items() produces a list of 2-item tuples. **dict()** can convert back to a dictionary.

```
mydict = {'a': 1, 'b': 2, 'c': 3}
these_items = mydict.items()
print these_items                       # [('a', 1), ('c', 3), ('b', 2)]


newdict = dict(these_items)

print newdict                           # {'a': 1, 'b': 2, 'c': 3}
```

2-item tuples can be sorted and sliced, so they are a handy alternate structure.

zip() zips up paralell lists into tuples

```
list1 = ['a', 'b', 'c', 'd']
list2 = [ 1,   2,   3,   4 ]

tupes = zip(list1, list2)

newdict = dict(tupes)                   # [('a', 1), ('c', 3), ('b', 2), ('c
```

Occasionally we are faced with lists that relate to each other one a 1-to-1

basis... or, we sometimes even shape our data into this form. Paralell lists like these can be zipped into multi-item tuples.