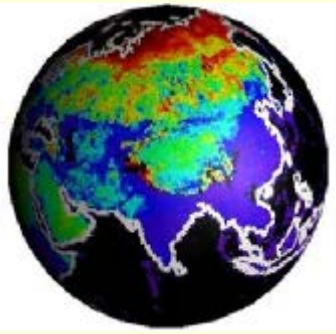


Chapter 11
Dictionaries (字典)

Jihn-Fa Jan, Ph.D.

Associate Professor

Department of Land Economics

National Chengchi University

Dictionary

- A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.
- You can think of a dictionary as a **mapping (映射)** between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item (項目)**.
- As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings. The function **dict** creates a new dictionary with no items. Because **dict** is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

Dictionary (cont'd)

- **The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:**

```
>>> eng2sp['one'] = 'uno'
```

- **This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:**

```
>>> print eng2sp
{'one': 'uno'}
```

  - Python 2: print eng2sp
  - Python 3: print(eng2sp)

- **create a new dictionary with three items:**

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Dictionary (cont'd)

- **But if you print eng2sp, you might be surprised:**

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

  - The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable

- **But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:**

```
>>> print eng2sp['two']
'dos'
```

  - The key 'two' always maps to the value 'dos' so the order of the items doesn't matter.

Dictionary (cont'd)

- **If the key isn't in the dictionary, you get an exception:**

```
>>> print eng2sp['four']
KeyError: 'four'
```

- **The len function works on dictionaries; it returns the number of key-value pairs:**

```
>>> len(eng2sp)
3
```

- **The in operator works on dictionaries; it tells you whether something appears as a *key in the dictionary* (appearing as a value is not good enough).**

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Dictionary (cont'd)

- To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the in operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

- The in operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm, as in Section 8.6.
- As the list gets longer, the search time gets longer in direct proportion.
- For dictionaries, Python uses an algorithm called a hashtable (雜湊表) that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary.

Dictionary (cont'd)

- **Exercise 11.1 Write a function that reads the words in words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.**

```
def make_word_list():
    d = {}
    fin = open('words.txt')
    for line in fin:
        word = line.strip()
        d[word] = ' '
    return d

words = make_word_list()
print len(words)
print 'any' in words
print 'some' in words
print 'aaaaa' in words
```

Dictionary as a set of counters

- **Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:**
  - 1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
  - 2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
  - 3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Dictionary as a set of counters (cont'd)

- **An implementation (實作) is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.**

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

```
>>> s = 'Pepper'
>>> histogram(s)
{'P': 1, 'r': 1, 'e': 2, 'p': 2}
```

  - The name of the function is histogram, which is a statistical term for a set of counters (or frequencies).

Looping and dictionaries

- **If you use a dictionary in a for statement, it traverses (巡訪) the keys of the dictionary. For example, print_hist prints each key and the corresponding value:**

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Looping and dictionaries (cont'd)

Exercise 11.3 Dictionaries have a method called keys that returns the keys of the dictionary, in no particular order, as a list. Modify print_hist to print the keys and their values in alphabetical order.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d

def print_hist(h):
    for c in h:
        print c, h[c]

def print_hist2(h):
    keys = h.keys()
    keys.sort()
    for c in keys:
        print c, h[c]

h = histogram('parrot')
print h
print_hist(h)
print '\nsorted keys and histogram:'
print_hist2(h)
```

```
>>> ================================ RESTART ================================
>>>
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
a 1
p 1
r 2
t 1
o 1

sorted keys and histogram:
a 1
o 1
p 1
r 2
t 1
```

Reverse lookup

- Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup (查找)**.
- But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup (反向查找)**; you have to search.
- Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise ValueError
```

Reverse lookup (cont'd)

- The raise statement causes an exception; in this case it causes a ValueError, which generally indicates that there is something wrong with the value of a parameter.
- If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup and an unsuccessful one:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print k
r
```

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

Reverse lookup (cont'd)

- **The result when you raise an exception is the same as when Python raises one: it prints a traceback (追蹤軌跡) and an error message. The raise statement takes a detailed error message as an optional argument. For example:**

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

Dictionaries and lists

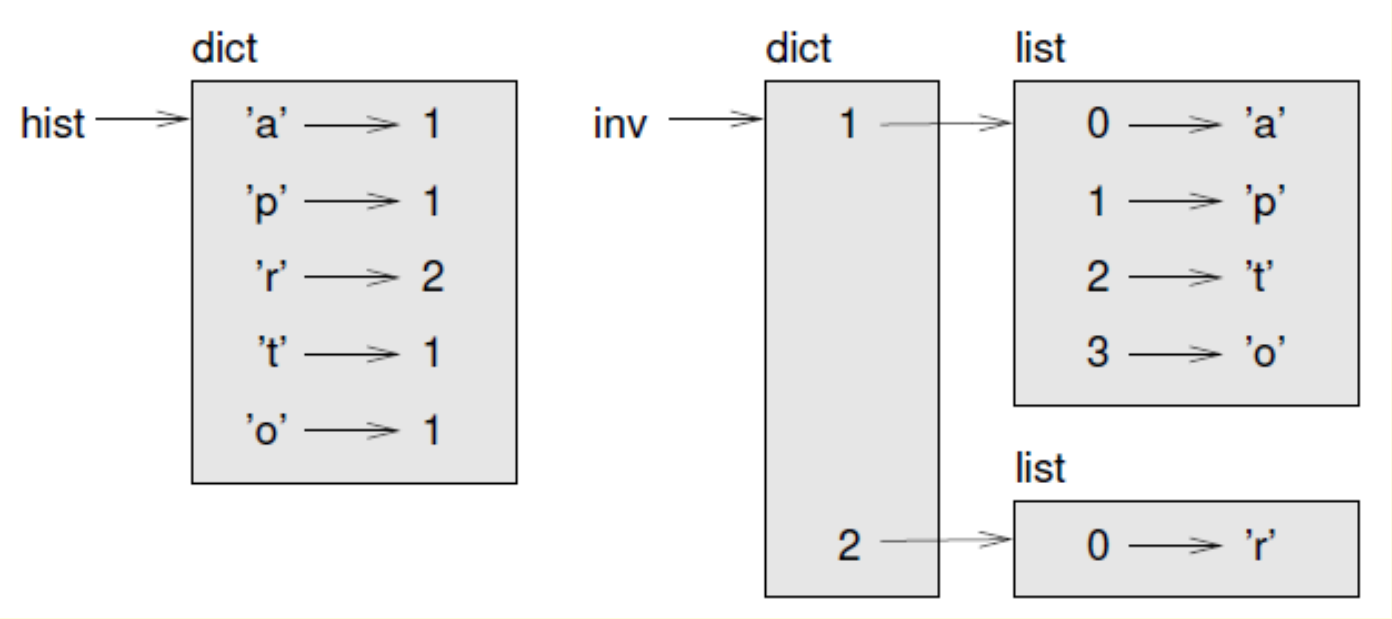
- Lists can appear as values in a dictionary. For example, if you were given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. For example:

```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Dictionaries and lists (cont'd)

- **And here is a diagram showing hist and inv:**



  - A dictionary is represented as a box with the type dict above it and the key-value pairs inside. If the values are integers, floats or strings, I usually draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

Dictionaries and lists (cont'd)

- Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

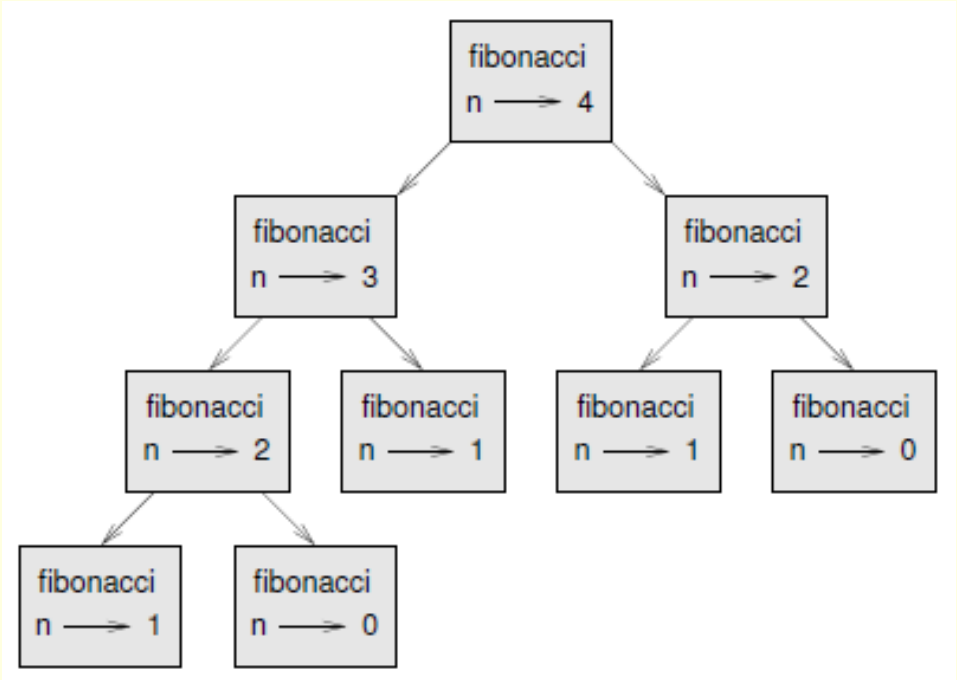
- A dictionary is implemented using a hashtable (可雜湊) and that means that the keys have to be hashable.
- A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

Dictionaries and lists (cont'd)

- This system works fine if the keys are immutable (不可變的). But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.
- That's why the keys have to be hashable, and why mutable types like lists aren't.
- Since dictionaries are mutable, they can't be used as keys, but they *can be used as values*.

Memos

- **If you played with the fibonacci function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. To understand why, consider this call graph (呼叫圖) for fibonacci with n=4:**



Memos (cont'd)

- A call graph shows a set of function frames (函式框), with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacci with n=2 and n=1. And so on.
- Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

Memos (cont'd)

- One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a memo (備忘). Here is an implementation of fibonacci using memos:

```
known = {0:0, 1:1}

def fibonacci(n):
        if n in known:
                return known[n]
        res = fibonacci(n-1) + fibonacci(n-2)
        known[n] = res
        return res

print "fibonacci(3) = ", fibonacci(3)
print "fibonacci(5) = ", fibonacci(5)
print "fibonacci(10) = ", fibonacci(10)
print "fibonacci(20) = ", fibonacci(20)
print "fibonacci(50) = ", fibonacci(50)
```

```
>>>
fibonacci(3) =  2
fibonacci(5) =  5
fibonacci(10) =  55
fibonacci(20) =  6765
fibonacci(50) =  12586269025
```

Global variables (全域變數)

- In the previous example, known is created outside the function, so it belongs to the special frame called __main__. Variables in __main__ are sometimes called global because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.
- It is common to use global variables for flags (旗標); that is, boolean variables that indicate ("flag") whether a condition is true. For example, some programs use a flag named verbose to control the level of detail in the output:

```
verbose = True

def example1():
    if verbose:
        print 'Running example1'
```

Global variables (cont'd)

- **If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:**

```
been_called = False

def example2():
    been_called = True          # WRONG
```

  - But if you run it you will see that the value of been_called doesn't change. The problem is that example2 creates a new local variable named been_called. The local variable (區域變數) goes away when the function ends, and has no effect on the global variable.

Global variables (cont'd)

- To reassign a global variable inside a function you have to declare the global variable before you use it:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

- The global statement tells the interpreter something like, "In this function, when I say been_called, I mean the global variable; don't create a local one."

```
count = 0

def example3():
    count = count + 1           # WRONG
```

If you run it you get:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Global variables (cont'd)

- **Python assumes that count is local, which means that you are reading it before writing it. The solution, again, is to declare count global.**

```
def example3():
    global count
    count += 1
```

- **If the global value is mutable, you can modify it without declaring it:**

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

Global variables (cont'd)

- **So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:**

```
def example5():
    global known
    known = dict()
```

Long integers

- **If you compute fibonacci(50), you get:**

```
>>> fibonacci(50)
12586269025L
```

  - The L at the end indicates that the result is a long integer, or type long.
- **Values with type int have a limited range; long integers can be arbitrarily big, but as they get bigger they consume more space and time.**
- **The mathematical operators work on long integers, and the functions in the math module, too, so in general any code that works with int will also work with long.**
- **Any time the result of a computation is too big to be represented with an integer, Python converts the result as a long integer:**

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
10000000000L
```

Debugging

- **As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:**
  - Scale down the input: If possible, reduce the size of the dataset.
  - Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data and print the type of a value.
  - Write self-checks: Sometimes you can write code to check for errors automatically.
  - Pretty print the output: Formatting debugging output can make it easier to spot an error.