Special Topics in Computer Science- CSC 4992

Dictionaries

Tuples

Data Structures

- A *data structure* is a means of organizing several data elements so they can be treated as one thing
- A *sequence* is a data structure in which the elements are accessible by *position* (first .. last)
- A *dictionary* is a data structure in which the elements are accessible by *content*

Examples of Dictionaries

- Dictionary
- Phone book
- Thesaurus
- Encyclopedia
- Cookbook
- World Wide Web

An element is accessed by *content*

This content can be

- A word
- A person's name
- A food type
- A text phrase or an image

Each content is called a *key*

Each associated element is called a *value*

Dictionaries

- Lists index their entries based on the position in the list
- Dictionaries - no order
- So we index the things we put in the dictionary with a "lookup tag"

```
purse = dict()
purse['money'] = 12
purse['candy'] = 3
purse['tissues'] = 75
print (purse)
{'money': 12, 'tissues': 75, 'candy': 3}
print (purse['candy'])
3
purse['candy'] = purse['candy'] + 2'
print (purse)
{'money': 12, 'tissues': 75, 'candy': 5}
```

Characteristics of a Dictionary

- A dictionary is a *set* of keys associated with values
- The keys are unique and need not be ordered by position or alphabetically
- Values can be duplicated
- Keys and values can be of any data types

Examples of Keys and Values

Some hexadecimal (base$_{16}$) digits and their values

`'A'`	`10`
`'B'`	`11`
`'C'`	`12`
`'D'`	`13`
`'E'`	`14`
`'F'`	`15`

A database of Ken's info

`'name'`	`'Ken'`
`'age'`	`65`
`'gender'`	`'M'`
`'occupation'`	`'teacher'`
`'hobbies'`	`['movies', 'gardening']`

Dictionaries in Python

```
hexdigits = {'A':10, 'B':11, 'C':12,
             'D':13, 'E':14, 'F':15}

database = {'name':'Ken', 'age':65,
            'gender':'M', 'occupation':'teacher'}

an_empty_one = {}
```

Syntax:

```
{<key> : <value>, … , <key> : <value>}
```

Acessing a Value with a Key

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database['name']
'Ken'

>>> database['age']
65
```

The subscript expects a key in the dictionary and returns the associated value

```
<dictionary>[<key>]
```

Key Must be in the Dictionary

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database['hair color']

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    database['hair color']
KeyError: 'hair color'
```

Comparing Lists and Dictionaries

- Dictionaries are like Lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print (lst)
[21, 183]
>>> lst[0] = 23
>>> print (lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print (ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print (ddd)
{'course': 182, 'age': 23}
```

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print (lst)
[21, 183]
>>> lst[0] = 23
>>> print (lst)
[23, 183]
```

List

Key	Value
[0]	21
[1]	183

lll

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print (ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print (ddd)
{'course': 182, 'age': 23}
```

Dictionary

Key	Value
['course']	183
['age']	21

ddd

Guard Access with an `if`

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> if 'hair color' in database:
        database['hair color']
```

The **`in`** operator can be used to search any sequence or dictionary

Alternative: Use the **`get`** Method

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database.get('hair color', None)
None
```

If the key (first argument) is in the dictionary, the value is returned

Otherwise, the default value (second argument) is returned

The `for` Loop Visits All Keys

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> for key in database:
        print(key, database[key])
gender M
age 65
name Ken
occupation teacher
```

Inserting a New Key/Value

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database['hair color'] = 'gray'
```

If the key is not already in the dictionary, Python creates one and inserts it with the associated value

Inserting a New Key/Value

```
hexdigits = {'A':10, 'B':11, 'C':12,
             'D':13, 'E':14, 'F':15}


>>> for i in range(10):
        hexdigits[str(i)] = i

>>> hexdigits
{'A':10, 'B':11, 'C':12, 'D':13, 'E':14, 'F':15,
 '1': 1, '0': 0, '3': 3, '2': 2, '5': 5, '4': 4,
 '7': 7, '6': 6, '9': 9, '8': 8}
```

Insert the remaining hexadecimal digits and their integer values

Replacing an Existing Value

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database['age'] = database['age'] + 1
```

If the key is already in the dictionary, Python replaces the associated value with the new one

Removing a key

```
>>> database = {'name':'Ken', 'age':65,
                'gender':'M', 'occupation':'teacher'}

>>> database.pop('age', None)
65
```

The **pop** method removes the key and its associated value and returns this value

Check for Bad Digits

```
>>> validDigits('1111000', 2)    # Base 2
True
>>> validDigits('7181900', 10)   # Base 10
True
>>> validDigits('71819A0', 10)   # Base 10
False
```

```
validDigits(<string of digits>, <integer base>)
```

Implementation

```
def validDigits(digits, base):
    ordzero = ord('0')
    for ch in digits:
        ordch = ord(ch)
        intvalue = ordch - ordzero
        if intvalue < 0 or intvalue >= base:
            return False
    return True
```

This version of the **`validDigits`** function works for bases 2-10

Let's extend it to work for bases 2-16

An Application of a Dictionary

```
def validDigits(digits, base):
    for ch in digits:
        ch = string.upper(ch)
        if not ch in hexdigits:                    # Test 1
            return False
        intvalue = hexdigits[ch]
        if intvalue >= base:                       # Test 2
            return False
    return True
```

Assumes **`hexdigits`** contains the integer values of all 16 digits

Test #1 checks for a non-digit character

Test #2 checks the digit's integer value against the allowed range

No messy conversion needed, just a dictionary lookup

Dictionary Literals (Constants)

- Dictionary literals use curly braces and have a list of key : value pairs
- You can make an empty dictionary using empty curly braces

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print jjj
{'jan': 100, 'chuck': 1, 'fred': 42}
>>> ooo = { }
>>> print (ooo)
{}
>>>
```

Most Common Name?

zhen
zhen
marquard
cwen
csev
zhen
csev
marquard
marquard
csev
cwen
zhen
zhen

Most Common Name?

zhen

marquard

cwen

zhen

zhen

csev

csev

marquard

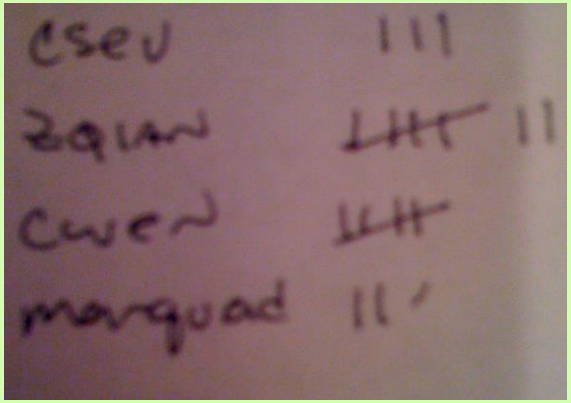
marquard

csev

cwen

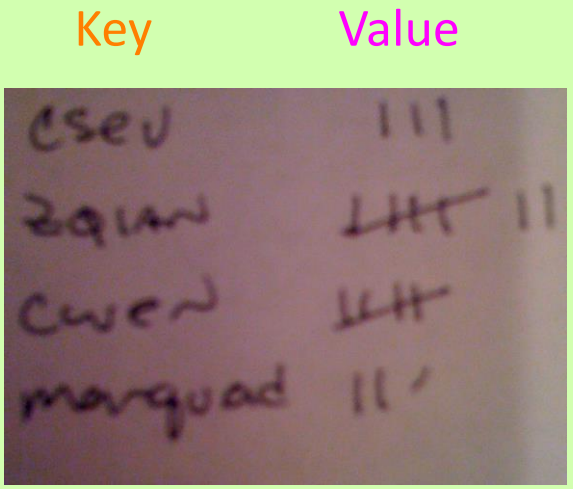
zhen

zhen



Many Counters with a Dictionary

- One common use of dictionary is counting how often we "see" something




```
>>> ccc = dict()
>>> ccc['csev'] = 1
>>> ccc['cwen'] = 1
>>> print ccc
{'csev': 1, 'cwen': 1}
>>> ccc['cwen'] = ccc['cwen'] + 1
>>> print ccc
{'csev': 1, 'cwen': 2}
```

Dictionary Tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> ccc = dict()
>>> print (ccc['csev'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'csev'
>>> print ('csev' in ccc)
False
```

When we see a new name

- When we encounter a new name, we need to add a new entry in the dictionary and if this the second or later time we have seen the name, we simply add one to the count in the dictionary under that name

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    if name not in counts:
       counts[name] = 1
    else :
        counts[name] = counts[name] + 1
print (counts)
```

{'csev': 2, 'zqian': 1, 'cwen': 2}

The get method for dictionaries

- This pattern of checking to see if a key is already in a dictionary and assuming a default value if the key is not there is so common, that there is a method called get() that does this for us

```
if name in counts:
   x = counts[name]
else :
   x = 0
x = counts.get(name, 0)
```

```
{'csev': 2, 'zqian': 1, 'cwen': 2
```

Default value if key does not exist (and no Traceback).

Simplified counting with get()

- We can use get() and provide a default value of zero when the key is not yet in the dictionary - and then just add one

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    counts[name] = counts.get(name, 0) + 1
print counts
```

Default

{'csev': 2, 'zqian': 1, 'cwen': 2}

Counting Pattern

```
counts = dict()
line = input('Enter a line of text:')

words = line.split()

print ('Words:', words)

print ('Counting...')
for word in words:
    counts[word] = counts.get(word,0) + 1
print ('Counts', counts)
```

The general pattern to count the words in a line of text is to split the line into words, then loop through the words and use a dictionary to track the count of each word independently.

Counting Words

Enter a line of text:the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car

Words: ['the', 'clown', 'ran', 'after', 'the', 'car', 'and', 'the', 'car', 'ran', 'into', 'the', 'tent', 'and', 'the', 'tent', 'fell', 'down', 'on', 'the', 'clown', 'and', 'the', 'car']
Counting...

Counts {'and': 3, 'on': 1, 'ran': 2, 'car': 3, 'into': 1, 'after': 1, 'clown': 2, 'down': 1, 'fell': 1, 'the': 7, 'tent': 2}

```
counts = dict()
line = input('Enter a line of text:’)
words = line.split()

print ('Words:', words)
print ('Counting...’)

for word in words:
    counts[word] = counts.get(word,0) + 1
print ('Counts', counts)
```

Enter a line of text:the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car

Words: ['the', 'clown', 'ran', 'after', 'the', 'car', 'and', 'the', 'car', 'ran', 'into', 'the', 'tent', 'and', 'the', 'tent', 'fell', 'down', 'on', 'the', 'clown', 'and', 'the', 'car']
Counting...

Counts {'and': 3, 'on': 1, 'ran': 2, 'car': 3, 'into': 1, 'after': 1, 'clown': 2, 'down': 1, 'fell': 1, 'the': 7, 'tent': 2}

Definite Loops and Dictionaries

- Even though dictionaries are not stored in order, we can write a for loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

```
>>> counts = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for key in counts:
...     print key, counts[key]
...
jan 100chuck 1fred 42
>>>
```

Retrieving lists of Keys and Values

- You can get a list of keys, values or items (both) from a dictionary

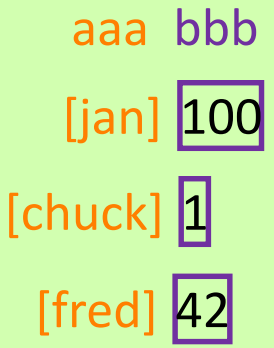
```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print (list(jjj) )
['jan', 'chuck', 'fred']
>>> print (jjj.keys())
['jan', 'chuck', 'fred']
>>> print (jjj.values())
[100, 1, 42]
>>> print( jjj.items())
[('jan', 100), ('chuck', 1), ('fred', 42)]
```

What is a 'tuple'? - coming soon...

Bonus: Two Iteration Variables!

- We loop through the key-value pairs in a dictionary using *two* iteration variables
- Each iteration, the first variable is the key and the the second variable is the *corresponding* value for the key

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for aaa,bbb in jjj.items() :
...         print (aaa, bbb)
...
jan 100
chuck 1
fred 42
>>>
```



Tuples

Tuples are like lists

- Tuples are another kind of sequence that function much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print (x[2])
Joseph
>>> y = ( 1, 9, 2 )
>>> print (y)
(1, 9, 2)
>>> print (max(y))
9
```

```
>>> for iter in y:
...     print (iter)
...
1
9
2
>>>
```

..but.. Tuples are "immutable"

- Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print (x)
[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str'
object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
Assignment
>>>
```

Things not to do with tuples

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:AttributeError: 'tuple' object has no
attribute 'sort'
>>> x.append(5)
Traceback:AttributeError: 'tuple' object has no
attribute 'append'
>>> x.reverse()
Traceback:AttributeError: 'tuple' object has no
attribute 'reverse'
>>>
```

A Tale of Two Sequences

```
>>> l = list()

Methods:
'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort'

>>> t = tuple()

Methods:
'count', 'index'
```

Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So in our program when we are making "temporary variables" we prefer tuples over lists.

Tuples and Assignment

- We can also put a tuple on the left hand side of an assignment statement
- We can even omit the parenthesis

```
>>> (x, y) = (4, 'fred')
>>> print (y)
Fred
>>> (a, b) = (99, 98)
>>> print (a)
99
```

Tuples and Dictionaries

- The items() method in dictionaries returns a list of (key, value) tuples

```
>>> d = dict()
>>> d['csev'] = 2
>>> d['cwen'] = 4
>>> for (k,v) in d.items():
...     print k, v
...
csev 2
cwen 4
>>> tups = d.items()
>>> print tups
[('csev', 2), ('cwen', 4)]
```

Tuples are Comparable

- The comparison operators work with tuples and other sequences If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Sorting Lists of Tuples

- We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary
- First we sort the dictionary by the key using the items() method

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

Using sorted()

We can do this even more directly using the built-in function sorted that takes a sequence as a parameter and returns a sorted sequence

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> d.items()
[('a', 10), ('c', 22), ('b', 1)]
>>> t = sorted(d.items())
>>> t
[('a', 10), ('b', 1), ('c', 22)]

>>> for k, v in sorted(d.items()):
...     print k, v
...
a 10
b 1
c 22
```

Sort by values instead of key

- If we could construct a list of tuples of the form (value, key) we could sort by value
- We do this with a for loop that creates a list of tuples

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> tmp = list()
>>> for k, v in c.items() :
...      tmp.append( (v, k) )
...
>>> print tmp
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> tmp.sort(reverse=True)
>>> print tmp
[(22, 'c'), (10, 'a'), (1, 'b')]
```

```
fhand = open('romeo.txt')
counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0 ) + 1

lst = list()
for key, val in counts.items():
    lst.append( (val, key) )
lst.sort(reverse=True)
for val, key in lst[:10] :
    print key, val
```

The top 10 most common words.