A problem:

We have a system where users login with a user-name and password. How do we store this information so that it is easy (and fast) to lookup?

Issues:

Username: A string of characters unique to a particular user.

Password: A non-empty string of characters.   Ideally we should encrypt this string rather than store it as clear-text but we’ll worry about encryption later (maybe).

We’re working for a company that anticipates having millions of users so we want this to be fast.

Operations:

Create new user / password:

- Make sure user-name isn’t already taken
- If not, save the user and password, else report error.

Login existing user (Lookup) :

- Lookup user-name and get the saved password
- Compare the stored password with the one they typed. If match, success, else report error.

How to store this?

So we have many pairs of user-names and associated passwords. How would you store this information?

How would you do the create and lookup operations?

Lists are what we know about so far….

[[“marshall”, “secret”],

 [“sheehan”, “myCat12”],

 [“orourke”, “origami”]]

OR

[“marshall”, “sheehan”, “orourke”]

[“secret”, “myCat12”, “origami”]

Efficiency of lists

Worst-case Scenarios:

If we have millions of users in our list, and the person last in the list logs in, we make millions of comparisons until we find the user.

Similarly if a user mistypes their username, we might search the whole list making millions of comparisons before we realize they aren’t found.

So if our list has N elements, it will take on-the-order-of N comparisons (on worst case) to lookup an item. In computer science, this is called O(n) time.

Python’s Dictionary gives a solution:

Dictionaries *map* words to definitions

- They associate *keys* to *values*

Create a dictionary with 3 key/value pairs:

```
>>>u = {“dave”: “secret”, “sara”: “myCat12”,  “orourke”: “origami”}
>>>type(u)
<class 'dict'>
>>>u[‘sara’]
‘myCat12’
```

*In older versions of Python dictionaries were created like:

```
>>> d = dict([ [“dave”,”secret”], [“sara”, “myCat12”] ])
```

Common dictionary operations

Create Empty dictionary:

```
mydict = {}
```

Add a key/value pair:

```
mydict['dave'] = 'secret'
mydict['sara'] = 'cat12'
```

Overwrite:

```
mydict['dave'] = 'blee'  # the value of 'secret' is replaced with 'blee'
```

Lookup:

```
mydict['Bozo']   #will throw exception if not found
mydict['dave']   #will return secret

if 'dave' in mydict:
    pw = mydict['dave']
```

Dictionaries Disadvantages/Advantages:

Advantage: Convenient Syntax.
Advantage: Simpler to work with than list-of-lists.
Disadvantage: Unordered. Cannot rely on a any ordering
Disadvantage: Dictionaries often require more space than is strictly necessary to store the items

Biggest Advantage:

- Insert: *Constant time - O(c)*
  Lookup: *Constant time - O(c)*
  "Constant" means a small number that holds true if there are 3 users or if there are 3,000,000 users!

How on Earth can it do this?

O(c) just doesn’t seem possible with millions of things.

Here’s the trick:

1. Create a big table with millions of “empty” entries
2. We use a function h(x) that is given the username (the key).

h(x) calculates a unique number based on this key.

This number is then used as an index into the table

Produce an index

So h(x) will take a user-name and will produce a unique index into the list.

h(‘dave’) -> 4

h(‘sara’) -> 1

	key	**value**
0		
1	‘sara’	‘myCat’
4	‘dave’	‘secret’

What is the function h doing?

The function h is called a “hash” function.

In this situation it takes the user-name string and computes an index appropriate for our users list.

```
h(“marshall”) = 852
h(“sara”) = 423
```

```
>>>ord(“m”)
109
>>>ord(“a”)
97
```

Hashing

ord(‘m’) + ord(‘a’) + ord(‘r’) + ord(‘s’) + ord(‘h’) + ord(‘a’) + ord(‘l’) + ord(‘l’) = 852

```
sum = 0
for c in username:
    sum += ord(c)
```

But the sum doesn’t necessarily fit within the bounds of our big list or maybe summing ords never produces large enough numbers to fill it up.

Complete the hash function

So we’ll multiply sum by some large number to make it bigger.

And then we’ll use modulo to make sure the final number fits within the length of our list

```
def hash(key):
    sum = 0
    for c in key:
        sum += ord(c)
    sum *= 345797
    # modulo by length of list N
    sum = sum % N
    return sum
```

Details not to worry about:

Issue: What happens if two different keys hash to the same index?

E.g. two users with the names: “sara” or “raas”

There’s a simple solution to deal with “collisions” which means:

- we have to store both key and value in the table (a “bucket”)
- we store all the buckets at or near the index

Take CSC-212 if you want to know more about how hashing works!

Back to Python dictionaries

It uses hashing to “map” keys to values.

Keys are typically strings but they can be other things too.

Values are typically objects.

For our login problem:

Keys are usernames (string)

Values are passwords (string)

key	value
“Dave”	“secret”
“Sara”	“myCat12”

Review of Operations on dictionaries

Create a new empty dictionary:

```
>>>d = {}
```

Add a Key/Value pair to the dictionary:

```
>>>d['dave'] = 'secret'
```

Is key stored?

```
>>> 'dave' not in d
False
>>> 'dave' in d
True
```

Get value associated with a key:

```
>>>d['dave']
'Secret'

>>>d['bozo']
Traceback (most recent call last):
File "<pyshell#80>", line 1, in
<module>
    D['bozo']
KeyError: 'bozo'
```

Try it for something real

Problem:

We read in a file and want to know how many times each letter occurs in the file.

We did this earlier this semester using lists.

Example of how to work with letter counts

```
letterDict = {}

# we found the letter ‘a’
# increment the count if we’ve seen it before
if ‘a’ in letterDict:
   letterDict[“a”] += 1
# add it to the table with count of 1 if not seen before
else:
   letterDict[“a”] = 1
```

Can you modify the program to use dictionaries instead of two lists?

```
def lc(filename):
    file = open(filename,'r')
    letters = []
    counts = []
    for line in file:
        for letter in line:
            if letter in letters:
                index = letters.index(letter)
                counts[index] += 1
            else:
                letters.append(letter)
                counts.append(1)
    file.close()
    return letters, counts
```

Hint:

```
file = open(filename,'r')
letterDict= {}
```

Sets

A Python object that has the properties of the mathematical Set.

1. An unordered collection of *elements*
2. No duplicate elements

Create like this:

```
a = set()          # for an empty set
b = {“a”,”b”,”c”} # for a set with elements
```

Set operations

A= {a,b,c}

B= {b,d,e}

Operation	operator	method	result
Union	`A | B`	`A.union(B)`	{a,b,c,d,e}
Intersection:	`A & B`	`A.intersection(B)`	{b}
Difference:	`A - B`	`A.difference(B)`	{a,c}
Symmetric Difference	`A ^ B`	`A.symmetric_difference(B)`	{a,c,d,e}

Set operations

Add an element:

```
a = set()
a.add(“dave”)
```

Membership tests:

```
“dave” in a
“Bozo” not in a
```

Cardinality (size):

```
len(a)
```

Remove an element:

```
a.discard(“dave”)
```

```
a.discard(“Bozo”)      #OK

a.remove(“Bozo”)     #Error
```

Iterating through a set

Just like a list:

```
for element in set:
    print (element)
```

Practical use of Sets

The main reason you would use sets in a program is if you want a collection that has no duplicate elements.

Quick way to eliminate dupes from a list:

```
L = [1,2,3,1,2,3,4]
S = set(L)
```

Practical Use of Sets

Example: A website that stores everyone’s favorite movies

favoriteMovies = {“Pulp Fiction”, “Star Wars”, “Amadeus”, ….}

Someone enters Star Wars (again)

favoriteMovies.add(“Star Wars”) will not add another “Star Wars” element

Sets vs Dictionaries

But if we wanted to keep track of how many people liked each movie, we’d use a dictionary

```
D = {}

if “Star Wars” in D:
   D[“Star Wars”] += 1  #another person likes it again
else:
   D[“Star Wars”] = 1
```

Sets vs Dictionaries

Python probably implement sets using the same mechanism that it uses for dictionaries: A hash table!

You can think of a set as a dictionary with only keys and no values.

- though d = {} will give you a dictionary NOT A SET

This is how a set can perform its *add* and *in* operations quickly. These are the O(c) *lookup* and *insert* operations that are properties of hash tables.