Programming for Engineers in Python

Recitation 5

Agenda

- Birthday problem
- Hash functions & dictionaries
- Frequency counter
- Object Oriented Programming

Random Birthday Problem

- What are the odds that two people will have the same birthday? (1/365~0.27%)
- In a room with 23 people what are the odds that at least two will have the same birthday? (~50%)
- In a room with *n* people, what are the odds?

http://www.greenteapress.com/thinkpython/code/birthday.py

See "Think Python" book for problems and solutions!

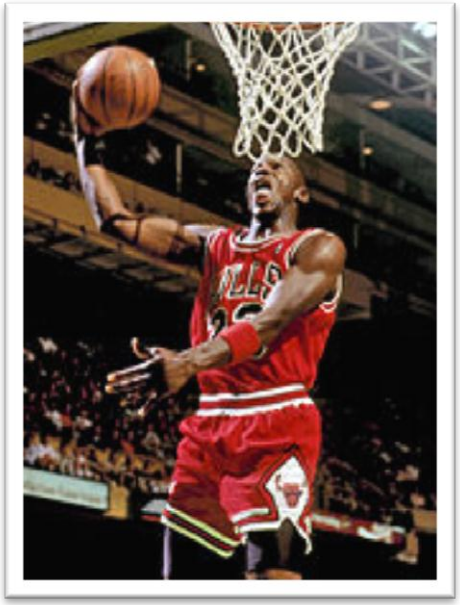
Notes on birthday problem

>>> s = t[:]

- Creates a copy of t in s – changing t will not change s!

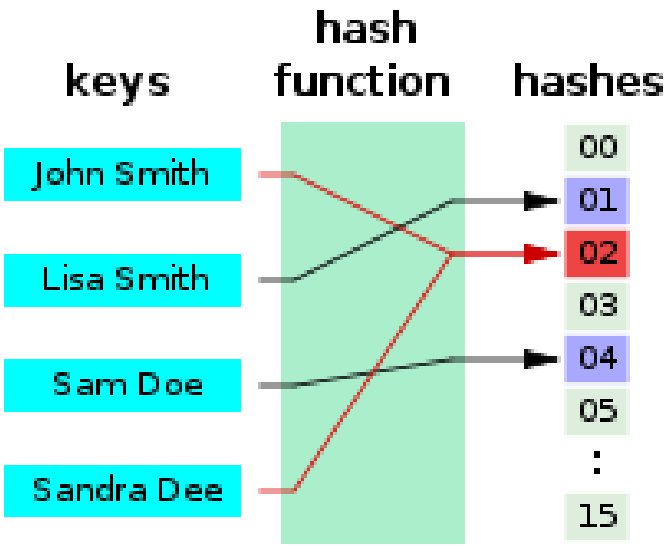
>>> random.randint(a, b)

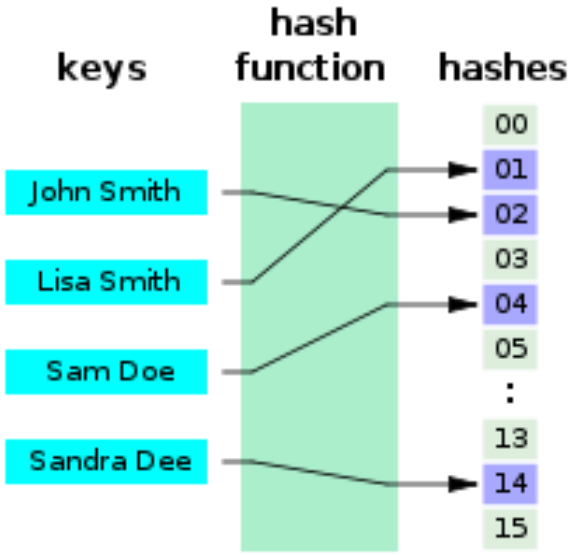
- Returns an integer between a and b, inclusive



Hash functions

- Gets a BIG value and returns a SMALL value
- Consistent: if x==y then hash(x)==hash(y)
- Collisions: there are some x!=y such that hash(x)==hash(y)





http://en.wikipedia.org/wiki/Hash_function

Hash functions

```
>>> hash("yoav")
1183749173
>>> hash(“noga")
94347453
>>> hash(1)
1
>>> hash(“1")
1977051568
```

Dictionary “implementation”

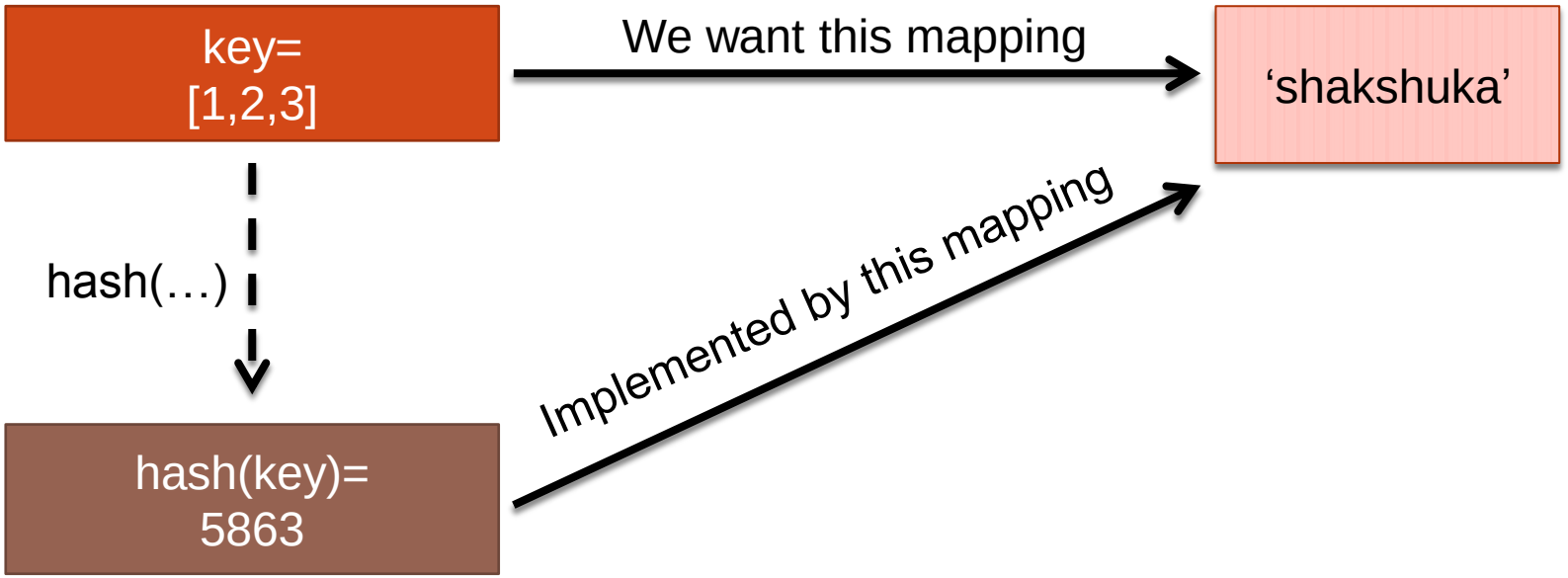
```
>>> dic = {“yoav” : “ram”, “noga” : “levy”}
>>> dic[“yoav”]
‘ram’
```

- What happens under the hood (roughly):

```
>>> hashtable = []
>>> hashtable[hash(“yoav”)]=“ram”
>>> hashtable[hash(“noga”)]=“levy”
>>> hashtable[hash(“yoav”)]
‘ram’
```

For a detailed explanation:
www.laurentluce.com/posts/python-dictionary-implementation/

Why must dictionary keys be immutable?



http://pyfaq.infogami.com/why-must-dictionary-keys-be-immutable

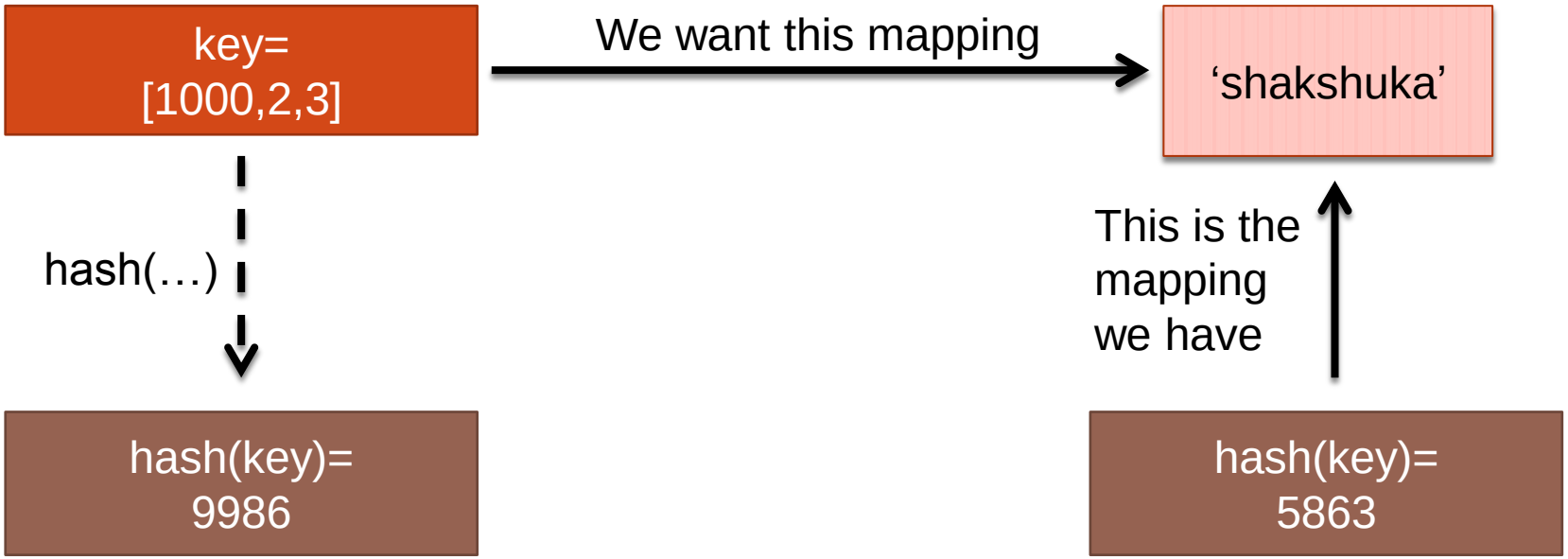
Why must dictionary keys be immutable?

We change an entry in the key:

>>> key[0]=1000

But it doesn't change the mapping!

Now, key doesn't map to the value!



http://pyfaq.infogami.com/why-must-dictionary-keys-be-immutable

Why must dictionary keys be immutable?

- Consider using a list as a key (**this is NOT a real Python code!**)

```
>>> key = [1,2,3]
>>> hash(key)                    # assuming key was hashable – it is not!
2011
>>> dic = {key : "shakshuka"}  # hashtable[2011] = "shakshuka"
>>> key[0] = 1000
>>> hash(key)
9986
>>> dic[key]                     # hashtable[1983]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    dic[key]
KeyError: '[1000,2,3]'
```

- The key [1000,2,3] doesn't exist!

http://pyfaq.infogami.com/why-must-dictionary-keys-be-immutable

Frequency Counter

- Substitution cyphers are broken by frequency analysis
- So we need to learn the frequencies of English letters
- We find a long text and start counting
- Which data structure will we use?

s u p e r c a l i f r a g i l i s t i c e x p i a l i d o c i o u s

Frequency Counter

```
str1 = 'supercalifragilisticexpialidocious‘
# count letters
charCount = {}

for char in str1:
    charCount[char] = charCount.get(char, 0) + 1

# sort alphabetically
sortedCharTuples = sorted(charCount.items())
```

Returned if char is not in the dictionary

Frequency Counter

```
# print
for charTuple in sortedCharTuples:
   print charTuple[0] , ‘ = ‘, charTuple[1]
```

```
a = 3
c = 3
d = 1
e = 2
f =  1
g = 1
…
```

The Table Does Not Lie

- We will construct a dictionary that holds the current table of the basketball league and a function that changes the table based on this week results
- Initialize the table:

```
>>> table = {}
>>> for team in team_list:
        table[team] = 0
>>> table
{'jerusalem': 0, ‘tel-aviv': 0, 'ashdod': 0,
```

טבלת ליגת לוטו
מחזור 5

#	שם הקבוצה	נצ'	הפ'	+/-	נק'
.1	אלדן אשקלון	4	1	+15	9
.2	גלבוע גליל	3	2	+2	8
.3	אלקטרה ת"א	4	0	+70	8
.4	הפועל י-ם	3	2	-3	8
.5	מ.כ הבקעה	3	1	-7	7
.6	מכבי בזן חיפה	2	3	+17	7
.7	הפועל חולון	2	3	+13	7
.8	מכבי ראשלצ	2	2	+26	6
.9	השרון הרצליה	1	4	-65	6
.10	מכבי אשדוד	1	3	-46	5
.11	גרינטופס נתניה	0	4	-22	4

Changing the table

- League results is a list of tuples - (winner, loser):

```
>>> results= [(“galil”,”tel-aviv”),(
“herzelia”,”ashdod”),…]
```

- We define a function to update the league table:

```
def update_table(results):
        for winner,loser in results:
                table[winner] = table[winner]+2
                table[loser] = table[loser]+1
```

Comparing teams

- The table must be sorted by points, we need a compare function:

```
def cmp_teams(team1, team2):
        pointCmp = cmp(team1[1],team2[1])
        if pointCmp != 0:
                return pointCmp
        else:
                return -cmp(team1[0],team2[0])
>>> cmp_teams(('galil', 2),('ashdod', 1))
1
>>> cmp_teams(('galil', 2),('ashdod', 2))
-1
>>> cmp_teams(('galil',2),('tel-aviv', 2))
1
```

Showing the table

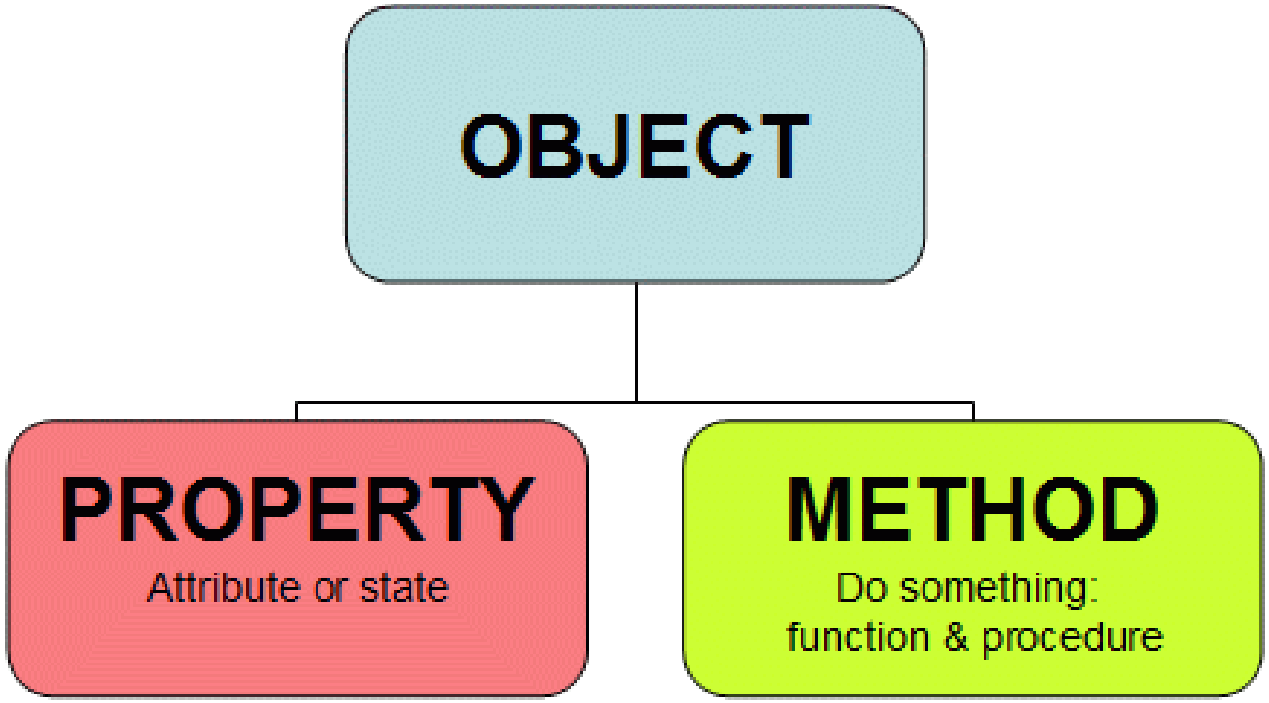
- The table must be sorted by points, in descending order:

```
>>> s_teams = sorted(table.items(),cmp_teams,reverse=True)
>>> for team, points in s_teams:
        print points,"|",team.capitalize()
2 | Galil
2 | Herzelia
2 | Jerusalem
1 | Ashdod
1 | Tel-aviv
```

Object Oriented Programing

- Started in class – more next week
- Model human thinking
  - We perceive the world as objects and relations
- Encapsulation
  - Keep data and relevant functions together
- Modularity
  - Replace one part without changing the others
- Reusability
  - Use the same code for different programs
- Inheritance
  - Extend a piece of code without changing it

Objects


OBJECT
PROPERTY
Attribute or state
METHOD
Do something:
function & procedure

Classes

- Definition

```
class ClassName (object):
    statement1
    statement2
    ...
```

- Initialization

```
>>> foo = Foo()
>>> foo # this is an instance of class Foo
<__main__.Foo instance at 0x01C60530>
>>> type(foo)
<type 'instance'>
```

__init__ - the constructor

- __init__ is used to initialize an instance
- Can have arguments other than *self*

```
class Student():
        def __init__(self,name):
                self.name = name

>>> student = Student('Yossi')
>>> student.name
'Yossi'
```

self

- *self* is a reference to the instance itself
- Used to refer to the data and methods
- The first argument of a method is always *self*, but there's no need to give it to the method

self – cont.

```
class ClassExample():
        def get_self(self):
            return self
>>> example = ClassExample()
>>> example
<__main__.ClassExample instance at 0x01C00A30>
>>> example.get_self()
<__main__.ClassExample instance at 0x01C00A30>
>>> ClassExample.get_self(example )
<__main__.ClassExample instance at 0x01C00A30>
>>> ClassExample.get_self()
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    ClassExample.get_self()
TypeError: unbound method get_self() must be called with
ClassExample instance as first argument (got nothing instead)
```

Example – Bank Account

- Code: https://gist.github.com/1399827

```
class BankAccount(object):
    def __init__(self, initial_balance=0):
        self.balance = initial_balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def is_overdrawn(self):
        return self.balance < 0
>>> my_account = BankAccount(15)
>>> my_account.withdraw(5)
>>> my_account.deposit(3)
>>> print “Balance:”,my_account.balance,”,
Overdraw:”,my_account.is_overdrawn()
Balance: 13 , Overdraw: False
```

Multimap

- A dictionary with more than one value for each key
- We already needed it once or twice and used:

```
>>> lst = d.get(key, [])
>>> lst.append(value)
>>> d[key] = lst
```

- Now we create a new data type

Multimap

- Use case – a dictionary of countries and their cities:

```
>>> m = Multimap()
>>> m.put('Israel','Tel-Aviv')
>>> m.put('Israel','Jerusalem')
>>> m.put('France','Paris')
>>> m.put_all('England',('London','Manchester','Moscow'))
>>> m.remove('England','Moscow')

>>> print m.get('Israel')
['Tel-Aviv', 'Jerusalem']
```

Code: https://gist.github.com/1397685

Advanced example: HTTP Server

- We will write a simple HTTP server
- The server allows to browse the file system through the browser
- Less than 20 lines, including imports!
- Code: https://gist.github.com/1400066
- Another one - this one writes the time and the server's IP address: https://gist.github.com/1397725