CSC108: Introduction to Computer Programming

Lecture 7

Wael Aboulsaadat

Acknowledgment: these slides are based on material by: Velian Pandeliev, Diane Horton, Michael Samozi, Jennifer Campbell, and Paul Gries from CS UoT

Announcements

- Midterm average is 69%
- Test suite for A2
- You can work in pairs!

What have we learnt up till now?

- Statements
  - Variables
  - Logical & Mathematical Operators
  - Assignment Statement
  - if/else Statement
  - while loops
- Types & Type conversion
- I/O
  - print
  - input & raw_input
  - Files

What have we learnt up till now?

- Docstrings
- Code Organization
  - Functions
  - Variable scope & Namespaces & Mutability
  - Classes & Objects
- Data Structures
  - Lists

Functions (revisited)

Function Parameters & Default Values

- We can specify optional parameters by supplying a fall-back default value in the function definition:

```
def calc_rectangle_area(breadth=1, length=1):
    return breadth * length
```

- Above, we are specifying that if calc_rectangle_area is ever called with just no arguments, breadth & length will be assigned 1

- However, if calc_rectangle_are is called with two parameters, the values are passed to the function

Function Overloading

- Defining two functions with the same name but a different number of parameters:

```
def calc_area(length):
        ...

def calc_area(breadth, length):
        ...

calc_area (5)
calc_area (3,8)
```

Tuples

Tuples

- Recall: lists are mutable, ordered collections of elements.
- In Python, lists have an immutable cousin called the tuple ('t-OO-ple' and 't-UH-ple' are both acceptable pronunciations)
- A **tuple is an ordered collection of elements that is** immutable.
- Like lists, tuples can be indexed and iterated over, but they don't have any methods and they can't be changed!

Tuple: syntax

- Tuples are created using the following notation:

```
>>> x = ( 4, 7, 9 )
```

- Where have we seen notation like this before?
- It turns out that formatting strings take tuples:

```
print "%s, %s" % (greeting, name)
```

- Even though tuples are created using round brackets, their elements are still accessed using square brackets:

```
>>> x[0]
4
```

Tuple: usage

- Tuples can be useful when defining things like points in 2-D and 3-D space.

- The name 'tuple' comes from the following series:

  single, double, triple, quadruple, quin**tuple**, sex**tuple...**

- It simply means "an ordered collection of two or more values".

Tuple: limitations

- There are a few things we **cannot do with tuples** that we could do with lists:

1) change elements:

```
x[2] = 8 # WRONG
```

2) use methods that change elements:

```
x.append('seven') # WRONG
```

3) use any methods at all:

```
x.index(4) # WRONG
```

Tuple: capabilities

- We can, however, still do the following:

1) use built-in functions:

```
len(x)
```

2) ask for individual elements:

```
print x[3]
```

3) iterate:

```
for value in x:
    print value
```

Tuple: advantage

- If a function returns a tuple, we can assign separate variables to each element:

```
>>> point = (4, 5, 8)
>>> x, y, z = point
>>> y
5
>>> x, y, z = (3, 2, -1)
```

Tuple: advantage

- If a function returns a tuple, we can assign separate variables to each element:

```
def quadcube(x):
'''Return x squared and x cubed'''
        return (x**2, x**3)

>>> a, b = quadcube(3)
>>> print a
9
>>> print b
27
```

Dictionaries

Dictionary

- A **dictionary** is a **collection of associations**.
- A dictionary entry consists of a key and a value.
- Keys are easily searchable and provide access to the information stored in their corresponding values.
- In a real-world dictionary, words are keys and their definitions are values.

Key	Value
1111	111 Coolway…
1234	218 Nice Road…

Dictionary: syntax

- Dictionaries are defined using the following syntax:

  a_dict = { key1 : value1, key2 : value2 ... }

- A dictionary's keys can be anything as long as they are **immutable objects** **(we don't want keys** changing on the fly!)

- A dictionary's values can be anything (including other dictionaries...).
- To retrieve the value associated with a particular key, we use square brackets:

  a_dict[key1]

Dictionary: example

- Dictionaries themselves are mutable, which means that we can add new key-value pairs as we go:

```
>>> d = {'a': 8, 'b': 4}
```

Key	Value
‘a’	8
‘b’	4

Dictionary: example

- Dictionaries themselves are mutable, which means that we can add new key-value pairs as we go:

```
>>> d = {'a': 8, 'b': 4}
>>> d['a']
8
```

Key	Value
‘a’	8
‘b’	4

Dictionary: insertion

- Dictionaries themselves are mutable, which means that we can add new key-value pairs as we go:

```
>>> d = {'a': 8, 'b': 4}
>>> d['a']
8
>>> d['c'] = 5
>>> d
{'a': 8, 'b': 4, 'c': 5}
>>> d['c']
5
```

Key	Value
‘a’	8
‘b’	4
‘c’	5

Dictionary: lookup

- We can check for membership in dictionaries, but only for keys:

```
>>> d = {'a': 8, 'b': 4}
>>> 'a' in d
True
>>> 8 in d
False
>>> d['c'] = 6
>>> 'c' in d
True
```

Key	Value
‘a’	8
‘b’	4
‘c’	6

Dictionary: deletion

- We can remove a key-value pair from a dictionary using keyword del:

```
>>> d = {'a': 8, 'b': 4}
>>> 'a' in d
True
```

Key	Value
‘a’	8
‘b’	4

Dictionary: deletion

- We can remove a key-value pair from a dictionary using keyword del:

```
>>> d = {'a': 8, 'b': 4}
>>> 'a' in d
True
>>> del d['a']
```

Key	Value
‘a’	8
‘b’	4

Dictionary: deletion

- We can remove a key-value pair from a dictionary using keyword del:

```
>>> d = {'a': 8, 'b': 4}
>>> 'a' in d
True
>>> del d['a']
>>> 'a' in d
False
>>> d
{'b': 4}
```

Key	Value
‘b’	4

Dictionary: keys & uniqueness

- The main use for dictionaries is being able to store values in named spaces that correspond to keys.
- Keys are like indices, except they don't have to be in order, and they don't have to be numbers.
- Dictionary keys are **unique.**
- Because they serve as the lookup mechanism in dictionaries (like indices in lists), each key has to appear only once in a given dictionary.

Dictionary: keys & order

- Note that keys in a dictionary are stored in an arbitrary order.

- There is no guarantee they will come out sorted in any way, or in the order in which you added them.

Dictionary: lookup

- Dictionaries come with some helpful methods.

- If we define a dictionary

```
>>> dict = {'a': 3, 'b': 8}
```

- dict.keys() returns a list of the keys in the dictionary:

```
['a', 'b']
```

- dict.values() returns a list of the values in the dictionary:

```
[3, 8]
```

- dict.items() returns a list of the key-value pairs in the dictionary as tuples: [('a', 3), ('b', 8)]

Dictionary: lookup

```
>>> dict = {'a': 3, 'b': 8}
# dict.get(key) does the same thing as
# dict[key], but it does not fail if the key is not
# in the dictionary:

>>> dict.get('a')
3

>>> dict.get('c')
None
>>> dict['c']
Traceback (most recent call last):.KeyError: 'c'
```

Dictionary: update

```
>>> dict  = {'a': 3, 'b': 8}
>>> dict2 = {'a': 5, 'c': 9}

>>> dict.update(dict2)
# copies values from dict2 to dict1. If any
# keys match, the values from dict2 will
# update those in dict. So, update() can be
# used both to extend a dictionary with new
# key-value pairs and to update a dictionary's
# existing pairs:
# makes dict {'a': 5, 'c': 9, 'b': 8}
```

Dictionary: methods

```
>>>  dict = {'a': 3, 'b': 8}


# dict.clear() empties the dictionary of all
# key-value pairs.
# This may be useful if you'd like to keep
# using the same object, but need to reset its
# contents.


>>> dict.clear()
>>> dict
{}
```

Dictionary: iteration over keys

- Dictionaries are collections, so we can iterate over them using for.

- With dictionaries, the for loop iteration advances over **keys, not key-value pairs!**

```
for key in d:
    print key
```

Dictionary: iteration over keys – bad way!

- This is an equivalent way, but it's **bad style:**

```
for key in d.keys():
    print key
```

- Why is it bad style?

- d.keys() creates and returns an entirely new object: a list of keys, which is an extra step that ties up extra memory.

- These efficiency considerations are going to start to matter in a week or so.

Dictionary: iteration over values – one way

- In dictionaries, we don't have direct access to the values with for. Instead, we use:

```
for value in d.values():
    print value
```

- This gives us access to values, but no good way to tie them back to keys.

Dictionary: iteration over key-value pairs

- The most common iteration technique for dictionaries is to iterate over key-value pairs. We do so as follows:

```
for (key, value) in d.items():
    print key
    print value
```

- Note that we are using a tuple inside a for loop definition to get two variables that change with every pass through the loop.

- This gives us access to keys and values.

Dictionary: advantage

- Fastest access time!

Key	Value
1111	111 Coolway…
1234	218 Nice Road…

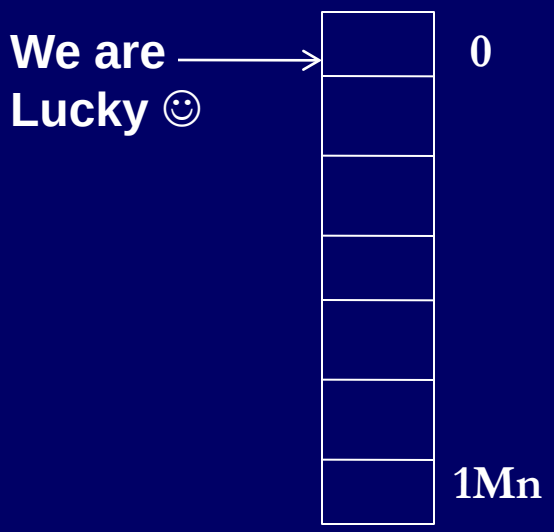
- How dictionary is implemented ?

- Let us compare finding a value in a list to finding a value in dictionary

Dictionary: advantage

- Finding a value in a list

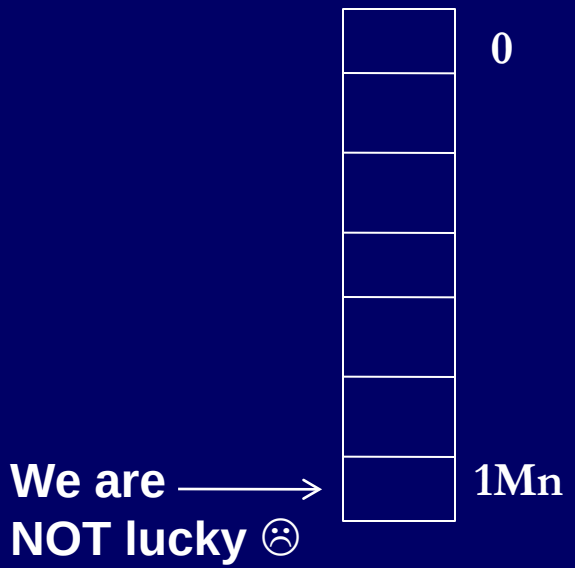
```
def checkValue(lst,Val):
    for element in lst:
        if element == Val:
            return True
    return False
```



Dictionary: advantage

- Finding a value in a list

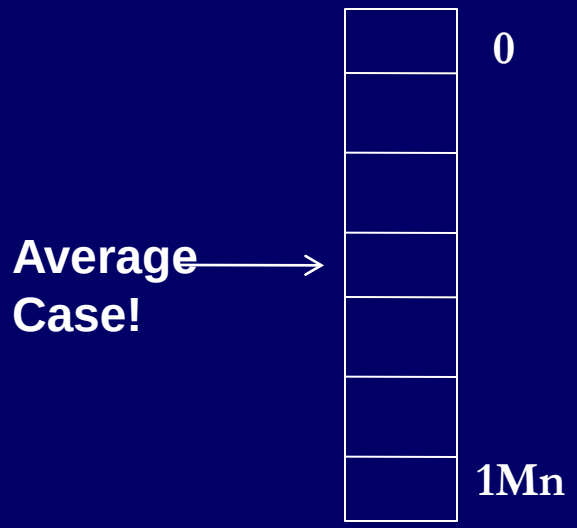
```
def checkValue(lst,Val):
    for element in lst:
        if element == Val:
            return True
    return False
```



Dictionary: advantage

- Finding a value in a list

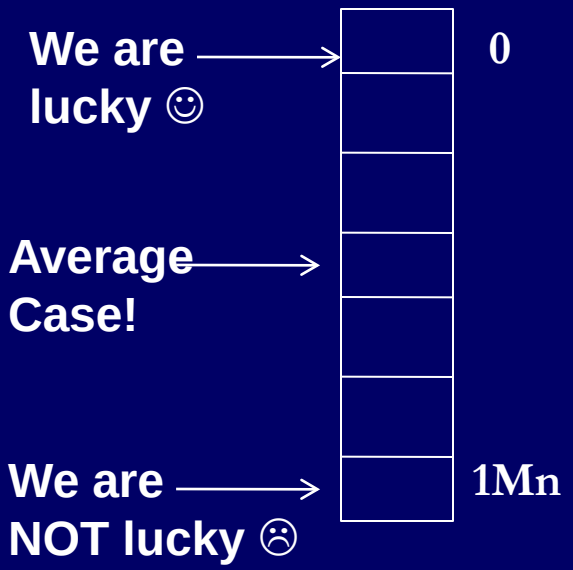
```
def checkValue(lst,Val):
    for element in lst:
        if element == Val:
            return True
    return False
```



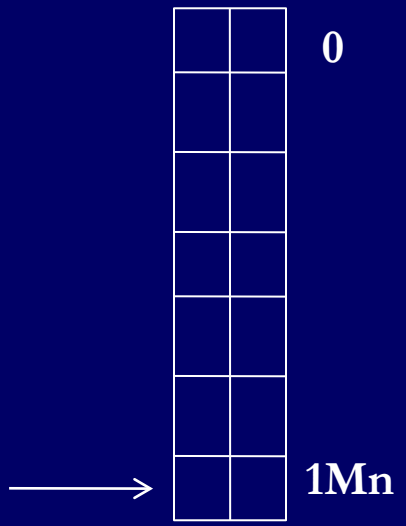
Dictionary: advantage

- Finding a value in a list

```
def checkValue(lst,Val):
    for element in lst:
        if element == Val:
            return True
    return False
```



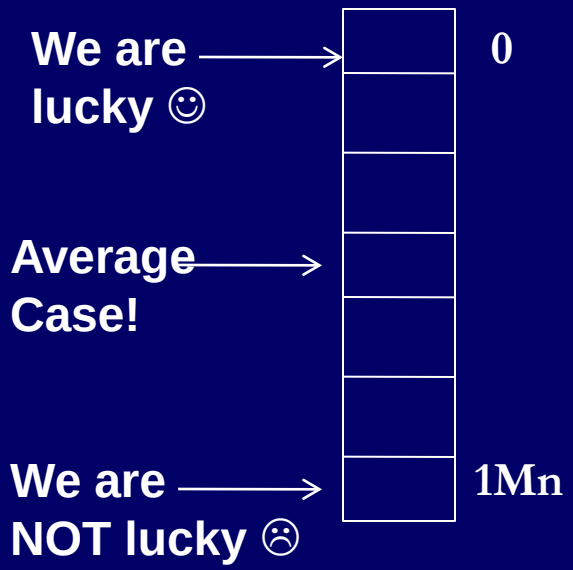
- Finding a value in dictionary d.get(key)
- Finding a value in dictionary does not involve iteration over values.
  Why? Hashing function!



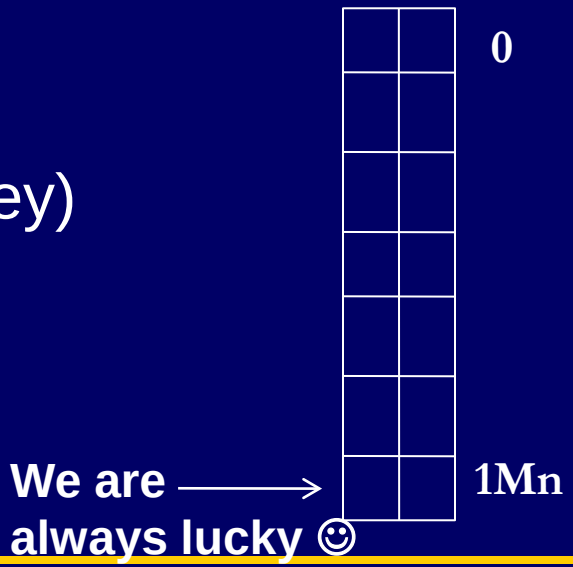
Dictionary: advantage

- Finding a value in a list

```
def checkValue(lst,Val):
    for element in lst:
        if element == Val:
            return True
    return False
```



- Finding a value in dictionary d.get(key)
- Finding a value in dictionary does not involve iteration over values.
  Why? Hashing function!



Dictionary: example

```
phone = {  '555-7632‘ : 'Paul',
           '555-9832‘ : 'Andrew',
           '555-6677‘ : 'Dan',
           '555-2222‘ : 'Paul',
           '555-7343' : 'Diane'}
```

- Suppose we want to create a list of Paul's phone numbers:

```
paulphones = []
for key in phone:
    if phone[key] == 'Paul':
        paulphones.append(key)
```

Dictionary: inverting key-value

```
phone = {  '555-7632‘ : 'Paul',
           '555-9832‘ : 'Andrew',
           '555-6677‘ : 'Dan',
           '555-2222‘ : 'Paul',
           '555-7343' : 'Diane'}
```

- Suppose we want to switch key-values

```
phoneR = {}
for (number, name) in phone.items():
    phoneR[name] = number
```

Dictionary: inverting key-value

```
phone = {  '555-7632‘ : 'Paul',
           '555-9832‘ : 'Andrew',
           '555-6677‘ : 'Dan',
           '555-2222‘ : 'Paul',
           '555-7343' : 'Diane'}
```

- To ensure we don't lose any numbers:

```
phoneR = {}
for (number, name) in phone.items():
    if name not in phoneR:
        phoneR[name] = [number]
    else:
        phoneR[name].append(number)
```

Classes & Objects (revisited)

Inheritance

- Recall how Biological inheritance work!
- We have a similar mechanism in Python
- A class can inherit from another!
- What does it mean code-wise?!

Inheritance

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
class ChildClass(ParentClass):
```

Definition of a class extending student

```
class Student:
    "A class representing a student."

    def __init__(self,n,a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
-------------------------------------------------

class Cs_student (Student):
    "A class extending student."

    def __init__(self,n,a,s):
        Student.__init__(self,n,a) #Call __init__ for student
        self.section_num = s

    def get_age():    #Redefines get_age method entirely
        print "Age: " + str(self.age)
```

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Extending __init__

- Same as for redefining any other method…
  - Commonly, the ancestor's **__init__** method is executed in addition to new commands.
  - You'll often see something like this in the **__init__** method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

This Week's To Do List

- Go through lecture slides – make sure you try the code snippets
- Try the lecture's programs posted on course website