

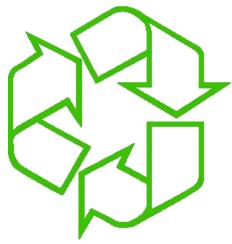
BBW 101

Introduction to Programming!

Lecture #05 – Collections File I/O



Last time... Control Flow, Functions



Repeating yourself

```
for f in [30,40,50]:  
    print(f, (f-32)/9.0*5)
```

```
counter = 1  
while counter <= n:  
    s = s + counter  
    counter += 1
```

Making decisions

```
if val < 0:  
    result = - val  
else:  
    result = val
```

```
if height > 100:  
    print("space")  
elif height > 50:  
    print("mesosphere")  
elif height > 20:  
    print("stratosphere")  
else:  
    print("troposphere")
```



```
def dbl_plus(x):  
    return 2*x + 1
```

Lecture Overview

- Collections
 - Lists
 - Sets
 - Tuples
 - Dictionaries
- File I/O

Disclaimer: Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Lecture Overview

- Collections
 - Lists
 - Sets
 - Tuples
 - Dictionaries
- File I/O

Disclaimer: Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Data Structures

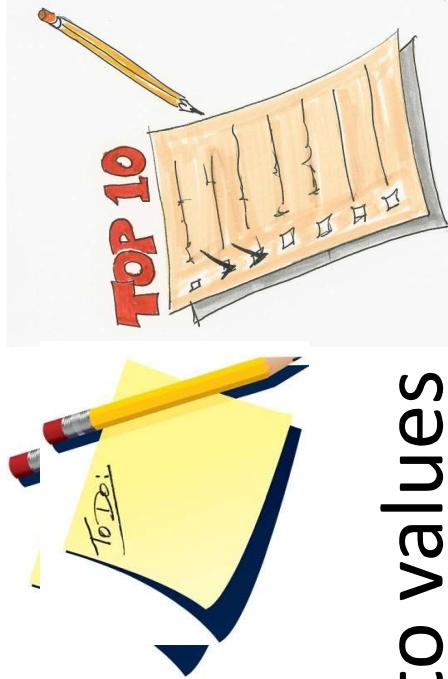
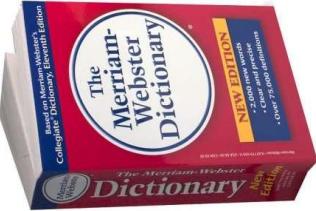
- A *data structure* is way of organizing data
 - Each data structure makes certain operations convenient or efficient
 - Each data structure makes certain operations inconvenient or inefficient
- Example: What operations are efficient with:
 - a file cabinet sorted by date?
 - a shoe box?



A Collection Groups Similar Things

- List: ordered
 - Set: unordered, no duplicates
 - Tuple: unmodifiable list
 - Dictionary: maps from values to values

Example: word → definition



Lecture Overview

- Collections
 - Lists
 - Tuples
 - Sets
 - Dictionaries
- File I/O

Disclaimer: Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

What is a List?

- A list is an ordered sequence of values, where each value is identified by an index.
- What operations should a list support efficiently and conveniently?
 - Creation
 - Querying
 - Modification

List Creation

```
a = [ 3, 1, 2*2, 1, 10/2, 10-1 ]
```

```
b = [ 5, 3, 'hi' ]
```

```
c = [ 4, 'a', a ]
```

```
a = [3, 4, 5]
```

- Use square brackets to specify a list.
- Separate each element with a comma.
- The empty list is written as [].



List Example - 1

```
L = ['I did it all', 4, 'Love']
```

```
for i in range(len(L)):  
    print(L[i])
```

```
>> I did it all  
>> 4  
>> Love
```

List Example - 2

```
Techs = ['MIT' , 'Caltech']  
Ivys = ['Harvard' , 'Yale' , 'Brown']  
  
Univs = [Techs , Ivys]  
Univs1 = [[ 'MIT' , 'Caltech'] , [ 'Harvard' , 'Yale' , 'Brown']]
```

```
print('Univs =' , Univs)  
print('Univs1 =' , Univs1)  
print(Univs == Univs1)
```

```
>> Univs = [[ 'MIT' , 'Caltech'] , [ 'Harvard' , 'Yale' , 'Brown']]  
>> Univs1 = [[ 'MIT' , 'Caltech'] , [ 'Harvard' , 'Yale' , 'Brown']]  
>> True
```



List Querying

- Extracting part of the list:
 - Single element: `mylist[index]`
 - Sublist (“slicing”): `mylist[startidx : endidx]`
- Find/lookup in a list
 - `elt in mylist`
 - Evaluates to a boolean value
- `mylist.index(x)`
 - Return the int index in the list of the first item whose value is x. It is an error if there is no such item.
- `list.count(x)`
 - Return the number of times x appears in the list.

List Mutation

- Insertion
- Removal
- Replacement
- Rearrangement

List Insertion

- **mylist.append(x)**
 - Extend the list by inserting x at the end
- **mylist.extend(L)**
 - Extend the list by appending all the items in the argument list
- **mylist.insert(i, x)**
 - Insert an item before the a given position.
 - a.insert(0, x) inserts at the front of the list
 - a.insert(len(a), x) is equivalent to a.append(x)

List Removal

- `list.remove(x)`
 - Remove the first item from the list whose value is `x`
 - It is an error if there is no such item
- `list.pop([i])`
 - Remove the item at the given position in the list, and return it.
 - If no index is specified, `a.pop()` removes and returns the last item in the list.

Notation from the Python Library Reference:
The square brackets around the parameter, “[i]”, means the argument is *optional*.
It does *not* mean you should type square brackets at that position.

List Replacement

- `mylist[index] = newvalue`
- `mylist[start : end] = newsublist`
 - Can change the length of the list
 - `mylist[start : end] = []` # removes multiple elements
 - `a[len(a) :] = l` # is equivalent to `a.extend(l)`

List Rearrangement

- **list.sort()**
 - Sort the items of the list, in place.
 - “in place” means by modifying the original list, not by creating a new list.
- **list.reverse()**
 - Reverse the elements of the list, in place.

How to Evaluate a List Expression

There are two new forms of expression:

- **[*a*, *b*, *c*, *d*]** *list creation*

– To evaluate:

- evaluate each element to a value, from left to right
- make a list of the values

– The elements can be arbitrary values, *including lists*

- ["a", 3, 3.14*r*r, fahr_to_cent(-40), [3+4, 5*6]]

List expression

- ***a*[*b*]** *list indexing* or dereferencing

– Index expression

– To evaluate:

- evaluate the list expression to a value
- evaluate the index expression to a value
- if the list value is not a list, execution terminates with an error
- if the element is not in range (not a valid index), execution terminates with an error
- the value is the given element of the list value (counting from **zero**)

List Expression Examples

What does this mean (or is it an error)?

```
[ "four" , "score" , "and" , "seven" , "years" ] [2]
```

```
[ "four" , "score" , "and" , "seven" , "years" ] [0,2,3]
```

```
[ "four" , "score" , "and" , "seven" , "years" ] [[0,2,3]]
```

```
[ "four" , "score" , "and" , "seven" , "years" ] [[0,2,3][1]]
```

List Expression Examples

```
>>> ["four", "score", "and", "seven", "years"] [2]
'and'
```

```
>>> ["four", "score", "and", "seven", "years"] [0,2,3]
```

TypeError: list indices must be integers or slices, not tuple

```
>>> ["four", "score", "and", "seven", "years"] [[0,2,3]]
```

TypeError: list indices must be integers or slices, not list

```
>>> ["four", "score", "and", "seven", "years"] [[0,2,3][1]]
'and'
```

Exercise: List Lookup

```
def index(somelist, value):
    """Return the position of the first occurrence of
    the element value in the list somelist.
    Return None if value does not appear in
    somelist."""
    i = 0
    for c in somelist:
        if c == value:
            return i
        i = i + 1
    return None
```

Exercise: List Lookup

```
def index(somelist, value):
    """Return the position of the first occurrence of
    the element value in the list somelist.
    Return None if value does not appear in
    somelist."""
    pass
```

Examples:

```
gettysburg = ["four", "score", "and",
              "seven", "years", "ago"]
index(gettysburg, "and") => 2
index(gettysburg, "years") => 4
```

Fact: `mylist[index(mylist, x)] == x`

List Slicing

- `mylist[startindex : endindex]` evaluates to a **sublist** of the original list
 - `mylist[index]` evaluates to an **element** of the original list
- Arguments are like those to the **range** function
 - `mylist[start : end : step]`
 - start index is inclusive, end index is exclusive
 - All 3 indices are *optional*
- Can assign to a slice: `mylist[s : e] = yourlist`

List Slicing Examples

```
test_list = ['e0', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6']
```

From e2 to the end of the list:

From beginning up to (but not including) e5:

Last element:

Last four elements:

Everything except last three elements:

Reverse the list:

Get a copy of the whole list:

```
test_list[2:]
```

```
test_list[:5]
```

```
test_list[-1]
```

```
test_list[-4:]
```

```
test_list[:-3]
```

```
test_list[::-1]
```

```
test_list[:]
```

Lecture Overview

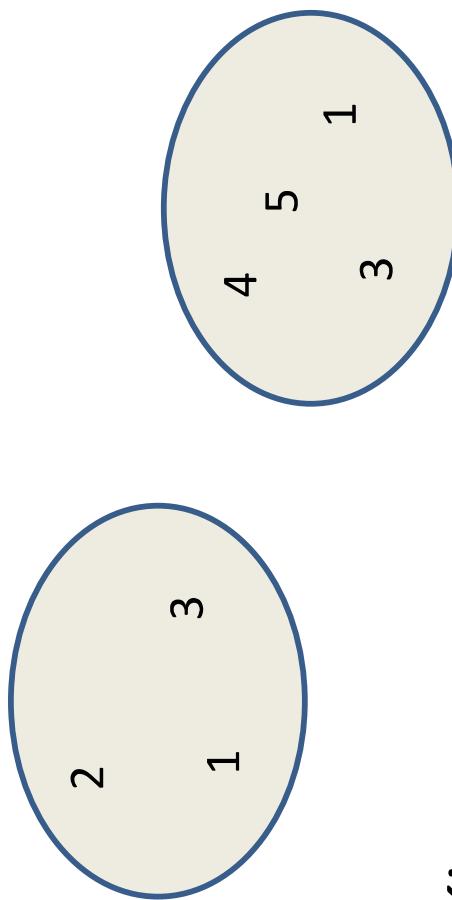
- Collections
 - Lists
 - Sets
 - Tuples
 - Dictionaries
- File I/O

Disclaimer: Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Sets

- Mathematical set: a collection of values, without duplicates or order
 - Order does not matter
 $\{ 1, 2, 3 \} == \{ 3, 2, 1 \}$
 - No duplicates
 $\{ 3, 1, 4, 1, 5 \} == \{ 5, 4, 3, 1 \}$
 - For every data structure, ask:
 - How to create
 - How to query (look up) and perform other operations
 - (Can result in a new set, or in some other datatype)
 - How to modify
- Answer: <http://docs.python.org/3/library/stdtypes.html#set>



Creating a Set

- Construct from a list:

```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
empty = set([])
```



Set Operations

- ```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
```
- membership  $\in$  Python: `in` 4 in prime  $\Rightarrow$  False
  - union  $\cup$  Python: `|` odd | prime  $\Rightarrow \{1, 2, 3, 5\}$
  - intersection  $\cap$  Python: `&` odd & prime  $\Rightarrow \{3, 5\}$
  - difference \ or - Python: `-` odd - prime  $\Rightarrow \{1\}$

Think in terms of set operations,  
*not in terms of iteration and element operations*

- Shorter, clearer, less error-prone, faster

Although we can do iteration over sets:

```
iterates over items in arbitrary order
for item in myset:
 ...
```

But we cannot index into a set to access a specific element.

# Modifying a Set

- Add one element to a set:

```
myset.add(newelt)
myset = myset | set([newelt])
```

- Remove one element from a set:

```
myset.remove(elt) # elt must be in myset or raises err
myset.discard(elt) # never errs
```

What would this do?

```
myset = myset - set([newelt])
```

- Choose and remove some element from a set:

```
myset.pop()
```

# Practice with Sets

```
z = set([5,6,7,8])
y = set([1,2,3,"foo",1,5])
k = z & y
j = z | y
m = y - z
z.add(9)
```

```
z: {8, 9, 5, 6, 7}
y: {1, 2, 3, 5, 'foo'}
k: {5}
j: {1, 2, 3, 5, 6, 7, 8, 'foo'}
m: {1, 2, 3, 'foo'}
```



# List VS. Set Operations (1)

Find the common elements in both list1 and list2:

```
out1 = []
for i in list2:
 if i in list1:
 out1.append(i)
```

or

```
out1 = [i for i in list2 if i in list1]
```

Find the common elements in both set1 and set2:

```
set1 & set2
```

Much shorter, clearer, easier to write!

# List VS. Set Operations (2)

Find the elements in either list1 or list2 (or both)  
(without duplicates):

```
out2 = list(list1) # make a copy
for i in list2:
 if i not in list1: # don't append elements
 out2.append(i) # already in out2
```

Or

```
out2 = list1+list2 # out1 (from previous example),
for i in out1: # common elements in both lists
 out2.remove(i) # Remove common elements
```

Find the elements in either set1 or set2 (or both):

```
set1 | set2
```

# List VS. Set Operations (3)

Find the elements in either list but not in both:

```
out3 = []
for i in list1+list2:
 if i not in list1 or i not in list2:
 out3.append(i)
```

Find the elements in either set but not in both:

```
set1 ^ set2 # symmetric difference
```

# Set Elements

- Set elements must be **immutable** values
    - int, float, bool, string, *tuple*
    - *not*: list, set, dictionary
  - Goal: only set operations change the set
    - after “`myset.add(x)`”, `x in myset`  $\Rightarrow$  True
    - `y in myset` always evaluates to the same value
- Both conditions should hold until `myset` itself is changed

# Set Elements

- Mutable elements can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
myset = { list1 } ← Hypothetical; actually illegal in Python
 TypeError: unhashable type: 'list'

list1 in myset ⇒ True
list3 in myset ⇒ True
list2.append("c") ← modifying myset “indirectly” would
 lead to different results

list1 in myset ⇒ ???
list3 in myset ⇒ ???
```

# Lecture Overview

- Collections
  - Lists
  - Sets
  - Tuples
  - Dictionaries
- File I/O

**Disclaimer:** Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

# Tuples

- Like strings, **tuples** are ordered sequences of elements.
- The individual elements can be of any type, and need not be of the same type as each other.
- Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.
- Tuples differ from lists in one hugely important way:
  - Lists are mutable. In contrast, tuples are immutable.

```
• t1 = ()
t2 = (1, 'two', 3)
print(t1)
print(t2)
```

```
>> ()
>> (1, 'two', 3)
```



# Tuples

- Like strings, tuples can be concatenated, indexed, and sliced.

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print(t2)
print((t1 + t2))
print((t1 + t2)[3])
print((t1 + t2)[2:5])

>> ((1, 'two', 3), 3.25)
>> (1, 'two', 3, (1, 'two', 3), 3.25)
>> (1, 'two', 3)
>> (3, (1, 'two', 3), 3.25)
```

# Tuples

- A for statement can be used to iterate over the elements of a tuple.
- The following code prints the common divisors of 20 and 100 and then the sum of all the divisors.

```
def findDivisors (n1, n2) :
 """Assumes n1 and n2 are positive ints
 Returns a tuple containing all common divisors
 of n1 & n2"""\n divisors = () #the empty tuple\n for i in range(1, min(n1, n2) + 1) :
 if n1%i == 0 and n2%i == 0:
 divisors = divisors + (i,)\n return divisors
```

```
divisors = findDivisors (20, 100)
print(divisors)
total = 0
for d in divisors:
 total += d
print(total)
>> (1, 2, 4, 5, 10, 20)
>> 42
```



# Lecture Overview

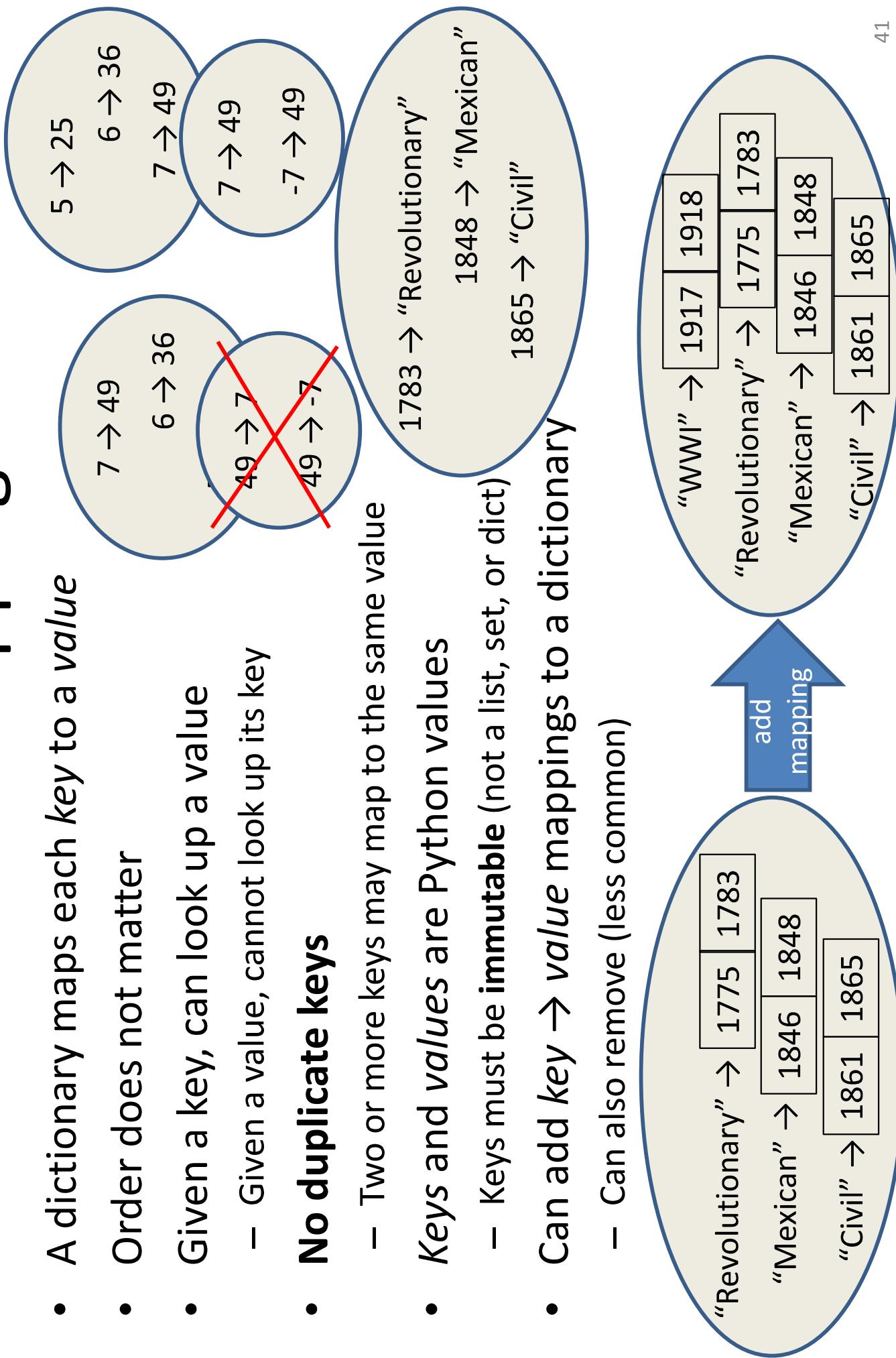
- Collections
  - Lists
  - Sets
  - Tuples
  - Dictionaries
- File I/O

**Disclaimer:** Much of the material and slides for this lecture were borrowed from

—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

# Dictionarys or Mappings

- A dictionary maps each *key* to a *value*
- Order does not matter
- Given a *key*, can look up a value
  - Given a value, cannot look up its key
- **No duplicate keys**
  - Two or more keys may map to the same value
- *Keys* and *values* are Python values
  - Keys must be **immutable** (not a list, set, or dict)
- Can add *key* → *value* mappings to a dictionary
  - Can also remove (less common)



# Dictionary Syntax in Python

```
d = {}
d = dict()
```

Two different ways  
to create an empty  
dictionary

```
us_wars_by_end = {
 1783: "Revolutionary",
 1848: "Mexican",
 1865: "Civil"}
```

1783 → "Revolutionary"

1848 → "Mexican"

1865 → "Civil"

```
us_wars_by_name = {
 "Civil" : [1861, 1865],
 "Mexican" : [1846, 1848],
 "Revolutionary" : [1775, 1783]
}
```

"Revolutionary" → 

|      |      |
|------|------|
| 1775 | 1783 |
|------|------|

"Mexican" → 

|      |      |
|------|------|
| 1846 | 1848 |
|------|------|

"Civil" → 

|      |      |
|------|------|
| 1861 | 1865 |
|------|------|

Syntax just like arrays, for accessing and setting:

```
us_wars_by_end[1783] ⇒ 'Revolutionary'
us_wars_by_end[1783][1:10] ⇒ 'evolution'
us_wars_by_name["WWI"] = [1917, 1918]
```



# Creating a Dictionary

```
>>> state = { "Atlanta" : "GA" , "Seattle" : "WA" }
```

“Atlanta” → “GA”  
“Seattle” → “WA”

```
>>> phonebook = dict()
>>> phonebook ["Alice"] = "206-555-4455"
>>> phonebook ["Bob"] = "212-555-2211"
```

“Alice” → “206-555-4455”

“Bob” → “212-555-1212”

```
>>> atomicnumber = {}
>>> atomicnumber ["H"] = 1
>>> atomicnumber ["Fe"] = 26
>>> atomicnumber ["Au"] = 79
```

“H” → 1  
“Fe” → 26  
“Au” → 79

# Accessing a Dictionary

```
>>> atomicnumber = { "H":1, "Fe":26, "Au":79}
>>> atomicnumber["Au"]
79
```

```
>>> atomicnumber["B"]
```

**KeyError: 'B'**

```
>>> "B" in atomicnumber
False
```

“H” → 1  
“Fe” → 26  
“Au” → 79

Good for iteration (for loops)

```
>>> atomicnumber.keys()
```

```
['H', 'Au', 'Fe']
```

```
>>> atomicnumber.values()
```

```
[1, 79, 26]
```

```
>>> atomicnumber.items()
```

```
[('H', 1), ('Au', 79), ('Fe', 26)]
```

for key in mymap.keys():
 val = mymap[key]
 ... use key and val

for key in mymap:
 val = mymap[key]
 ... use key and val

for (key, val) in mymap.items():
 ... use key and val

# Iterating Through a Dictionary

```
atomicnumber = {"H":1, "Fe":26, "Au":79}
```

```
Print out all the keys:
for element_name in atomicnumber.keys():
 print(element_name)
```

H  
Fe  
Au

```
Another way to print out all the keys:
for element_name in atomicnumber:
 print(element_name)
```

H  
Fe  
Au

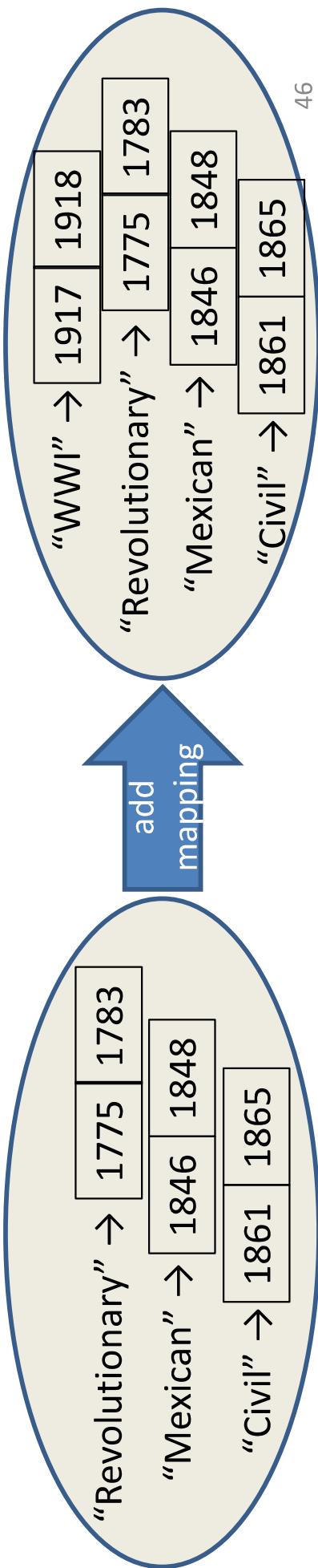
```
Print out the keys and the values
for (element_name, element_number) in atomicnumber.items():
 print("name:", element_name, "number:", element_number)
```

name: H number: 1  
name: Fe number: 26  
name: Au number: 79

# Modifying a Dictionary

```
us_wars1 = {
 "Revolutionary" : [1775, 1783],
 "Mexican" : [1846, 1848],
 "Civil" : [1861, 1865]}

us_wars1["WWI"] = [1917, 1918] # add mapping
us_wars1.pop("Mexican") # remove mapping
```



# Dictionary Exercises

- Convert a list to a dictionary:
  - Given [5, 6, 7], produce {5:25, 6:36, 7:49}
- Reverse key with value in a dictionary:
  - Given {5:25, 6:36, 7:49}, produce {25:5, 36:6, 49:7}
- What does this do?

```
squares = { 1:1, 2:4, 3:9, 4:16 }
squares[3] + squares[3] => 18
squares[3 + 3]
squares[2 + 2] => 16
```

KeyError: 6

# Dictionary Exercise Solutions

- Convert a list to a dictionary:
  - E.g. Given [5, 6, 7], produce {5:25, 6:36, 7:49}

```
d = []
for i in [5, 6, 7] : # or range(5, 8)
 d[i] = i * i
```
- Reverse key with value in a dictionary:
  - E.g. Given {5:25, 6:36, 7:49}, produce {25:5, 36:6, 49:7}

```
k = {}
for i in d.keys() :
 k[d[i]] = i
```

# A list is like a dictionary

- A list maps an integer to a value
  - The integers must be a continuous range  $0..i$

```
mylist = ['a', 'b', 'c']
mylist[1] => 'b'
mylist[3] = 'c' # index out of range error
```

- In what ways is a list **more** convenient than a dictionary?
- In what ways is a list **less** convenient than a dictionary?

# Not Every Value is Allowed to be a Key - 1

- Keys must be immutable values
    - int, float, bool, string, tuple
    - *not*: list, set, dictionary
  - Goal: only dictionary operations change the keyset
    - after “`mydict[x] = y`”, `mydict[x]` ⇒ `y`
    - if `a == b`, then `mydict[a] == mydict[b]`
- These conditions should hold until `mydict` itself is changed

# Not Every Value is Allowed to be a Key - 2

- Mutable keys can violate these goals

```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
mydict = {}

mydict[list1] = "z" ← Hypothetical; actually illegal in Python
mydict[list3] ⇒ "z"

list2.append("c")
mydict[list1] ⇒ ???
mydict[list3] ⇒ ???
```

# Lecture Overview

- Collections
  - Lists
  - Sets
  - Tuples
  - Dictionaries
- File I/O

**Disclaimer:** Much of the material and slides for this lecture were borrowed from

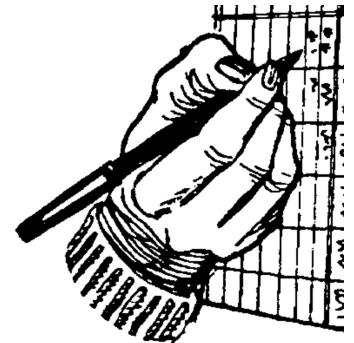
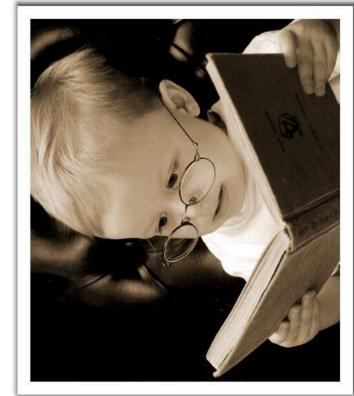
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

# File Input and Output

- As a programmer, when would one use a file?
- As a programmer, what does one do with a file?

Important operations:

- open a file 
- close a file 



# Files and Filenames

- A **file** object represents data on your disk drive
  - Can read from it and write to it
- A **filename** (usually a string) states where to find the data on your disk drive
  - Can be used to find/create a file
- Each operating system comes with its own file system for creating and accessing files:
  - Linux/Mac: "/home/reia/lectures/file\_io.pptx"
  - Windows: "C:\Users\reia\MyDocuments\cute\_dog.jpg"

# Two Types of Filenames

- An **Absolute** filename gives a specific location on disk:  
`" /home/reia/bbm101/14wi/lectures/file_io.pptx"` or  
`"C:\Users\rea\MyDocuments\homework3\images\Husky.png"`
    - Starts with “/” (Unix) or “C:\” (Windows)
    - Warning: code will fail to find the file if you move/ rename files or run your program on a different computer
  - A **Relative** filename gives a location relative to the *current working directory*:  
`"lectures/file_io.pptx"` or `" images\Husky.png"`
    - Warning: code will fail to find the file unless you run your program from a directory that contains the given contents
- **A relative filename is usually a better choice**

# Examples

Linux/Mac: These could all refer to the same file:

```
"~/rea/class/140/homework3/images/Husky.png"
"homework3/images/Husky.png"
"images/Husky.png"
"Husky.png"
```

Windows: These could all refer to the same file:

```
"C:\rea\My Documents\class\140\homework3\images\Husky.png"
"homework3\images\Husky.png"
"images\Husky.png"
"Husky.png"
```

Depending on what your current working directory is  
\$ pwd -> print working directory

# “Current Working Directory” in Python

The directory from which you ran Python

To determine it from a Python program:

```
>>> import os # "os" stands for "operating system"
>>> os.getcwd()
'/users/john Doe/Documents'
```

*Can be the source of confusion: where are my files?*

# Reading a File in Python

```
Open takes a filename and returns a file.
This fails if the file cannot be found & opened.
myfile = open("datafile.dat")

Approach 1:
for line_of_text in myfile:
... process line_of_text

Approach 2:
all_data_as_a_big_string = myfile.read()

myfile.close() # close the file when done reading
```

*Assumption: file is a sequence of lines*  
*Where does Python expect to find this file (note the relative pathname)?*

# Reading a File Example

```
Count the number of words in a text file
in_file = "thesis.txt"
myfile = open(in_file)
num_words = 0
for line_of_text in myfile:
 word_list = line_of_text.split()
 num_words += len(word_list)
myfile.close()
print("Total words in file: ", num_words)
```

# Reading a File Multiple Times

You can iterate over a list as many times as you like:

```
mylist = [3, 1, 4, 1, 5, 9]
for elt in mylist:
 ... process elt
 for elt in mylist:
 ... process elt
```

## How to read a file multiple times?

Solution 1: Read into a list, then iterate over it

```
myfile = open("datafile.dat")
mylines = []
for line_of_text in myfile:
 mylines.append(line_of_text)
... use mylines
```

Solution 2: Re-create the file object (slower, but a better choice if the file does not fit in memory)

```
myfile = open("datafile.dat")
for line_of_text in myfile:
 ... process line_of_text
myfile = open("datafile.dat")
for line_of_text in myfile:
 ... process line_of_text
```

This loop body will never be executed!

# Writing to a File in Python

```
Replaces any existing file of this name
myfile = open ("output.dat", "w")
```

open for Writing  
(no argument, or  
"r", for Reading)

```
Just like printing output
myfile.write("a bunch of data")
myfile.write("a line of text\n")
```

"\n" means  
end of line  
(Newline)

Wrong; results in:  
**TypeError: expected a character buffer object**

```
myfile.write(4)
myfile.write(str(4))
```

Right. Argument  
must be a string

```
myfile.close()
```

close when done  
with all writing

# More Examples - 1

```
nameHandle = open('characters.txt', 'w')

for i in range(2):
 name = input('Enter name: ')
 nameHandle.write(name + '\n')
 nameHandle.close()

nameHandle = open('characters.txt', 'r')

for line in nameHandle:
 print(line)
 nameHandle.close()
```

- If we had typed in the names Rick and Morty, this will print  
**Rick**  
**Morty**

- The extra line between Rick and Morty is there because print starts a new line each time it encounters the '\n' at the end of each line in the file.

# More Examples - 2

```
nameHandle = open('characters.txt', 'w')
nameHandle.write('Jerry\n')
nameHandle.write('Beth\n')
nameHandle.close()
```

```
nameHandle = open('characters.txt', 'r')
for line in nameHandle:
 print line[:-1]
nameHandle.close()
```

- It will print  
**Jerry**  
**Beth**
- Notice that
  - we have overwritten the previous contents of the file.
  - **print line[:-1]** avoids extra newline in the output

# More Examples - 3

```
nameHandle = open('characters.txt', 'a')
nameHandle.write('Rick\n')
nameHandle.write('Morty\n')
nameHandle.close()
```

```
nameHandle = open('kids', 'r')
for line in nameHandle:
 print line[:-1]
nameHandle.close()
```

- It will print

**Jerry**

**Beth**

**Rick**

**Morty**

- Notice that we can open the file for appending (instead of writing) by using the argument 'a'.

# Common functions for accessing files

- **open (fn, 'w')** fn is a string representing a file name.  
Creates a file for writing and returns a file handle.
- **open (fn, 'r')** fn is a string representing a file name. Opens an existing file for reading and returns a file handle.
- **open (fn, 'a')** fn is a string representing a file name. Opens an existing file for appending and returns a file handle.
- **fh.close ()** closes the file associated with the file handle fh.

# Common functions for accessing files

- **`fh.read()`** returns a string containing the contents of the file associated with the file handle `fh`.
- **`fh.readline()`** returns the next line in the file associated with the file handle `fh`.
- **`fh.readlines()`** returns a list each element of which is one line of the file associated with the file handle `fh`.
- **`fh.write(s)`** write the string `s` to the end of the file associated with the file handle `fh`.
- **`fh.writelines(S)`** `S` is a sequence of strings. Writes each element of `S` to the file associated with the file handle `fh`.