INT3075 Programming and Problem Solving for Mathematics

Dictionaries and Sets

More Data Structures

- We have seen the list data structure and what it can be used for
- We will now examine two more advanced data structures, the Set and the Dictionary
- In particular, the dictionary is an important, very useful part of python, as well as generally useful to solve many problems.

Dictionaries

What is a dictionary?

- In data structure terms, a dictionary is better termed an *associative array, associative list* or a *map*.
- You can think if it as a list of pairs, where the first element of the pair, the ***key***, is used to retrieve the second element, the ***value***.
- Thus we ***map a key to a value***

Key Value pairs

- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key

Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate key:value pairs

```
contacts= {'bill': '353-1234',
 'rich': '269-1234', 'jane': '352-1234'}
print (contacts)
{'jane': '352-1234',
 'bill': '353-1234',
 'rich': '269-1234'}
```

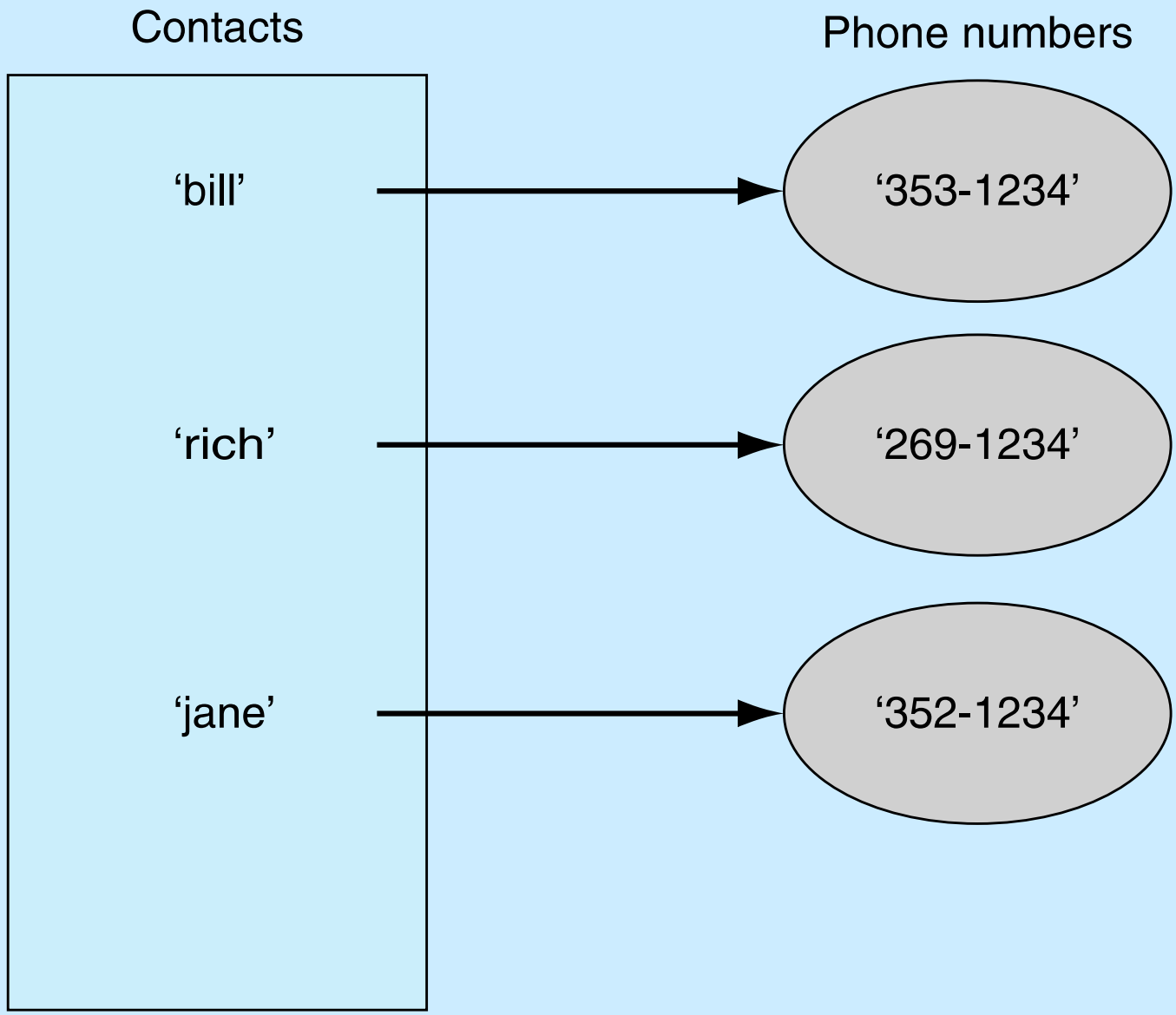


FIGURE 9.1 Phone contact list: names and phone numbers.

keys and values

- Key must be immutable
  - strings, integers, tuples are fine
  - lists are NOT
- Value can be anything

collections but not a sequence

- dictionaries are collections but they are not sequences such as lists, strings or tuples
  - there is no order to the elements of a dictionary
  - in fact, the order (for example, when printed) might change as elements are added or deleted.
- So how to access dictionary elements?

Access dictionary elements

Access requires `[ ]`, but the *key* is the index!

```
my_dict={}
```

– an empty dictionary

```
my_dict['bill']=25
```

– added the pair 'bill':25

```
print(my_dict['bill'])
```

– prints `25`

Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3, 'rich':10}
print(my_dict['bill'])    # prints 3
my_dict['bill'] = 100
print(my_dict['bill'])    # prints 100
```

Dictionary keys can be any immutable object

```
demo = {2: ['a','b','c'], (2,4): 27, 'x': {1:2.5, 'a':3}}
demo
    {'x': {'a':3, 1:2.5}, 2: ['a','b','c'], (2,4): 27}
demo[2]
    ['a', 'b', 'c']
demo[(2,4)]
    27
demo ['x']
    {'a':3, 1: 2.5}
demo['x'][1]
    2.5
```

again, common operators

Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

fewer methods

Only 9 methods in total. Here are some:

- `key in my_dict`

  does the key exist in the dictionary
- `my_dict.clear()` – empty the dictionary
- `my_dict.update(yourDict)` – for each key in `yourDict`, updates `my_dict` with that key/value pair
- `my_dict.copy` - shallow copy
- `my_dict.pop(key)` – remove key, return value

Dictionary content methods

- `my_dict.items()` – all the key/value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values

They return what is called a *dictionary view*.

- the order of the views correspond
- are dynamically updated with changes
- are iterable

Views are iterable

```
for key in my_dict:
    print(key)
```

- prints all the keys

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

- prints all the key/value pairs

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

- prints all the values

```
my_dict = {'a':2, 3:['x', 'y'], 'joe':'smith'}
```

```
>>> dict_value_view = my_dict.values()
>>> dict_value_view                                    # a view
dict_values([2, ['x', 'y'], 'smith'])
>>> type(dict_value_view)                              # view type
<class 'dict_values'>
>>> for val in dict_value_view:                        # view iteration
        print(val)

2
['x', 'y']
smith
>>> my_dict['new_key'] = 'new_value'
>>> dict_value_view                                    # view updated
dict_values([2, 'new_value', ['x', 'y'], 'smith'])
>>> dict_key_view = my_dict.keys()
dict_keys(['a', 'new_key', 3, 'joe'])
>>> dict_value_view
dict_values([2, 'new_value', ['x', 'y'], 'smith']) # same order
>>>
```

Frequency of words in list
3 ways

membership test

```
count_dict = {}
for word in word_list:
    if word in count_dict:
        count_dict[word] += 1
    else:
        count_dict[word] = 1
```

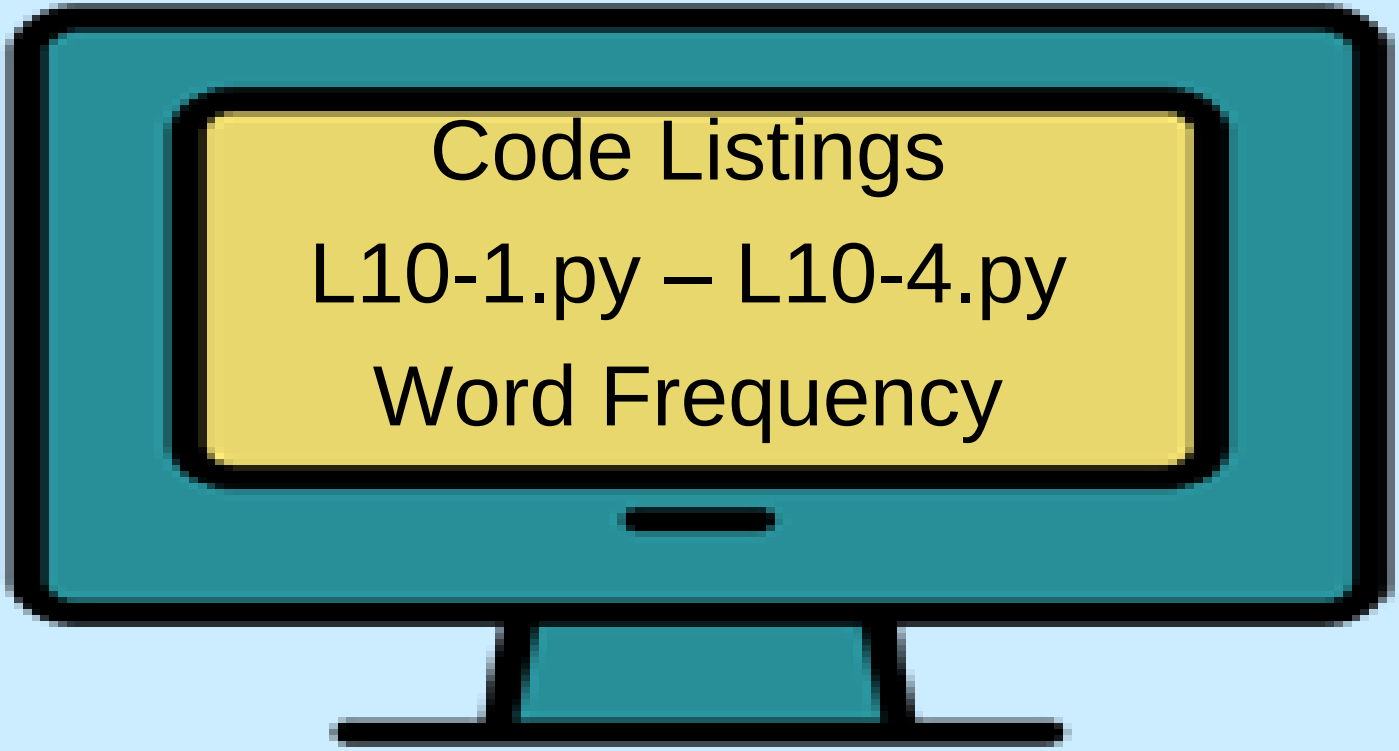
exceptions

```
count_dict = {}
for word in word_list:
    try:
        count_dict[word] += 1
    except KeyError:
        count_dict[word] = 1
```

get method

`get` method returns the value associated with a dict key or a default value provided as second argument. Below, the default is 0

```
count_dict = {}
for word in word_list:
    count_dict[word] = count_dict.get(word,0) + 1
```


Code Listings
L10-1.py – L10-4.py
Word Frequency

4 functions

- `add_word(word, word_dict)`. Add word to the dictionary. No return
- `process_line(line, word_dict)`. Process line and identify words. Calls `add_word`. No return.
- `pretty_print(word_dict)`. Nice printing of the dictionary contents. No return
- `main()`. Function to start the program.

Passing mutables

- Because we are passing a mutable data structure, a dictionary, we do not have to return the dictionary when the function ends
- If all we do is update the dictionary (change the object) then the argument will be associated with the changed object.

```
def add_word(word, word_count_dict):
    '''Update the word frequency: word is the key, frequency is the value.'''
    if word in word_count_dict:
        word_count_dict[word] += 1
    else:
        word_count_dict[word] = 1
```

```
1  import string
2  def process_line(line, word_count_dict):
3      '''Process the line to get lowercase words to add to the dictionary.'''
4      line = line.strip()
5      word_list = line.split()
6      for word in word_list:
7          # ignore the '--' that is in the file
8          if word != '--':
9              word = word.lower()
10             word = word.strip()
11             # get commas, periods and other punctuation out as well
12             word = word.strip(string.punctuation)
13             add_word(word, word_count_dict)
```

sorting in pretty_print

- the `sort` method works on lists, so if we sort we must sort a list
- for complex elements (like a tuple), the sort compares the first element of each complex element:

```
(1, 3) < (2, 1)        # True
(3,0) < (1,2,3)        # False
```

- a list comprehension (commented out) is the equivalent of the code below it

```
1  def pretty_print(word_count_dict):
2      '''Print nicely from highest to lowest frequency.'''
3      # create a list of tuples, (value, key)
4      # value_key_list = [(val,key) for key,val in d.items()]
5      value_key_list=[]
6      for key,val in word_count_dict.items():
7          value_key_list.append((val,key))
8      # sort method sorts on list's first element, the frequency.
9      # Reverse to get biggest first
10     value_key_list.sort(reverse=True)
11     # value_key_list = sorted([(v,k) for k,v in value_key_list.items()],
   reverse=True)
12     print('{:11s}{:11s}'.format('Word', 'Count'))
13     print('_'*21)
14     for val,key in value_key_list:
15         print('{:12s}  {:<3d}'.format(key,val))
```

```
1 def main ():
2     word_count_dict={}
3     gba_file = open('gettysburg.txt','r')
4     for line in gba_file:
5         process_line(line, word_count_dict)
6     print('Length of the dictionary:',len(word_count_dict))
7     pretty_print(word_count_dict)
```

Periodic Table example

comma separated values (csv)

- ***csv*** files are a text format that are used by many applications (especially spreadsheets) to exchange data as text
- row oriented representation where each line is a row, and elements of the row (columns) are separated by a comma
- despite the simplicity, there are variations and we'd like Python to help

csv module

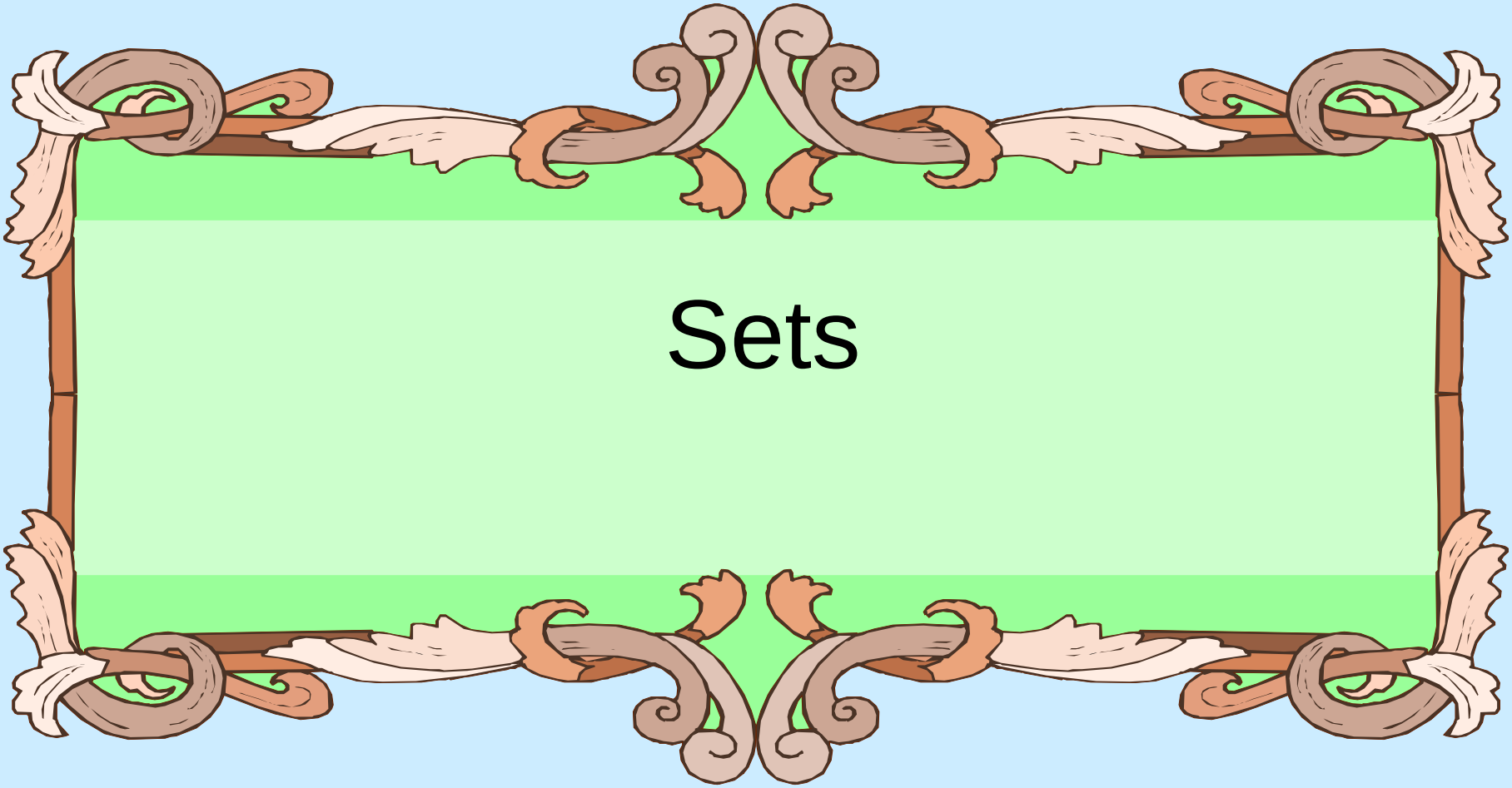
- `csv.reader` takes an opened file object as an argument and reads one line at a time from that file
- Each line is formatted as a list with the elements (the columns, the comma separated elements) found in the file

encodings other than UTF-8

- this example uses a csv file encoded with characters other than UTF-8 (our default)
  - in particular, the symbol ± occurs
- can solve by opening the file with the correct encoding, in this case `windows-1252`

example

```
>>> import csv
>>> periodic_file = open("Periodic-Table.csv", "r",encoding="windows-1252")
>>> reader = csv.reader(periodic_file)
>>> for row in reader:
        print(row)

# some of the output data
['2', 'He', '18', 'VIII A', '1', 'helium', '4.003', '0', '', '', ...]
['3', 'Li', '1', 'I A', '2', 'lithium', '6.941', '+1', '', '', ... ]
['4', 'Be', '2', 'II A', '2', 'beryllium', '9.012', '+2', '', '', ...]
['5', 'B', '13', 'III A', '2', 'boron', '10.81', '+3', '', '', ...]
# etc. etc.
```


Sets

Sets, as in Mathematical Sets

- in mathematics, a set is a collection of objects, potentially of many different types
- in a set, no two elements are identical. That is, a set consists of elements each of which is unique compared to the other elements
- there is no order to the elements of a set
- a set with no elements is the empty set

Creating a set

Set can be created in one of two ways:

- constructor: `set(iterable)` where the argument is iterable

```
my_set = set('abc')
my_set → {'a', 'b', 'c'}
```

- shortcut: `{}`, braces where the elements have no colons (to distinguish them from dicts)

```
my_set = {'a', 'b','c'}
```

Diverse elements

- A set can consist of a mixture of different types of elements

```
my_set = {'a',1,3.14159,True}
```

- as long as the single argument can be iterated through, you can make a set of it

no duplicates

- duplicates are automatically removed

```
my_set = set("aabbccdd")
print(my_set)
```

→ `{'a', 'c', 'b', 'd'}`

example

```
>>> null_set = set()          # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4}         # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}       # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")       # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'}          # order not maintained!
```

common operators

Most data structures respond to these:

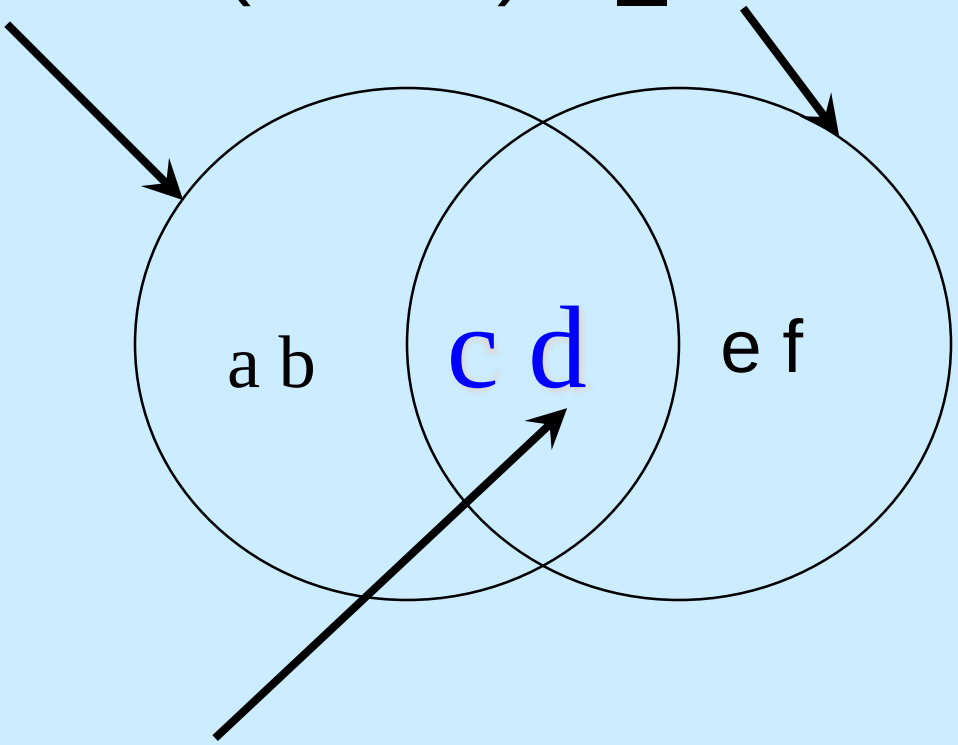
- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - boolean indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`

Set operators

- The set data structure provides some special operators that correspond to the operators you learned in middle school.
- These are various combinations of set contents
- These operations have both a method name and a shortcut binary operator

method: intersection, &

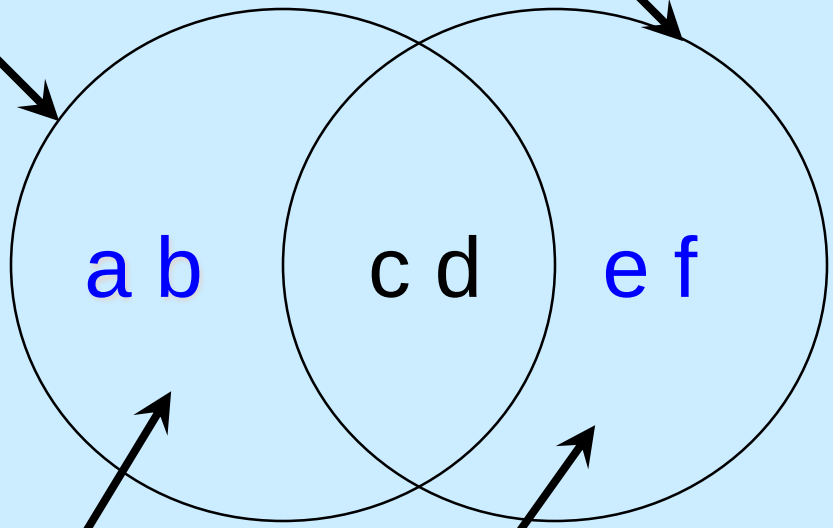
a_set=set("abcd") b_set=set("cdef")




```
a_set & b_set → {'c', 'd'}
b_set.intersection(a_set) → {'c', 'd'}
```

method: difference, -

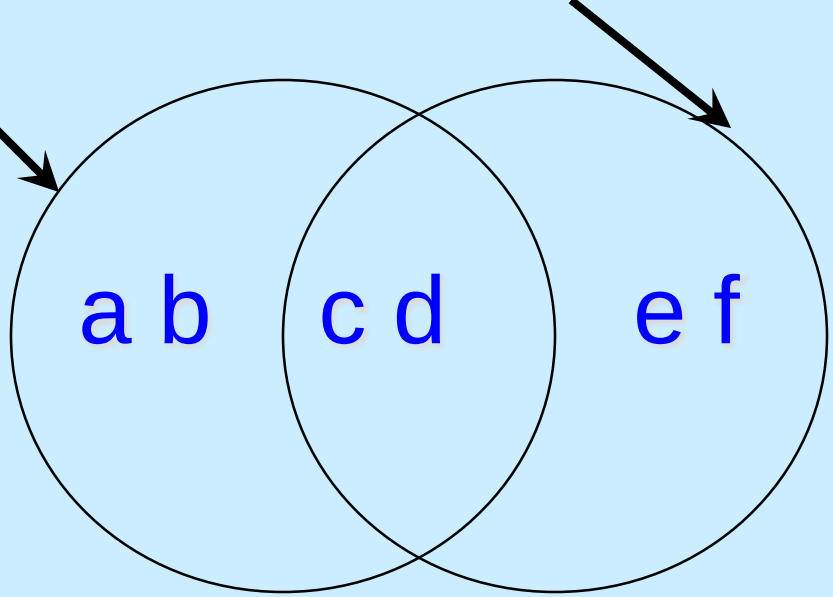
a_set=set("abcd") b_set=set("cdef")




```
a_set - b_set → {'a', 'b'}
b_set.difference(a_set) → {'e', 'f'}
```

method: union, |

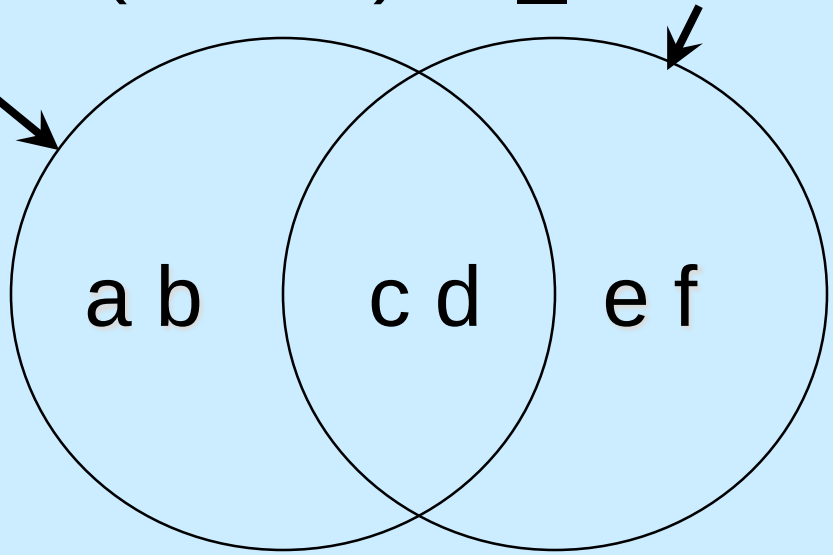
a_set=set("abcd") b_set=set("cdef")




```
a_set | b_set → {'a', 'b', 'c', 'd', 'e', 'f'}
b_set.union(a_set)→{'a', 'b', 'c', 'd', 'e', 'f'}
```

method: symmetric_difference, ^

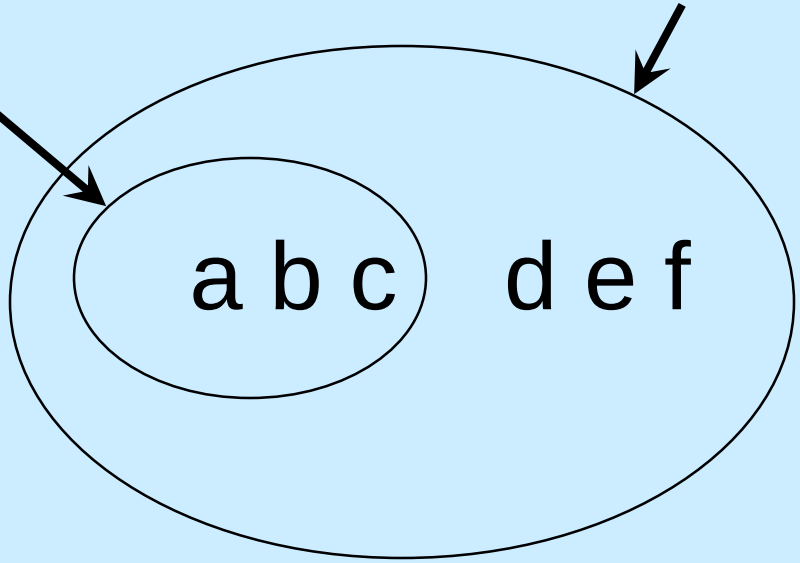
a_set=set("abcd"); b_set=set("cdef")




```
a_set ^ b_set → {'a', 'b', 'e', 'f'}
b_set.symmetric_difference(a_set) → {'a', 'b', 'e', 'f'}
```

method: issubset, <=
method: issuperset, >=

small_set=set("abc"); big_set=set("abcdef")



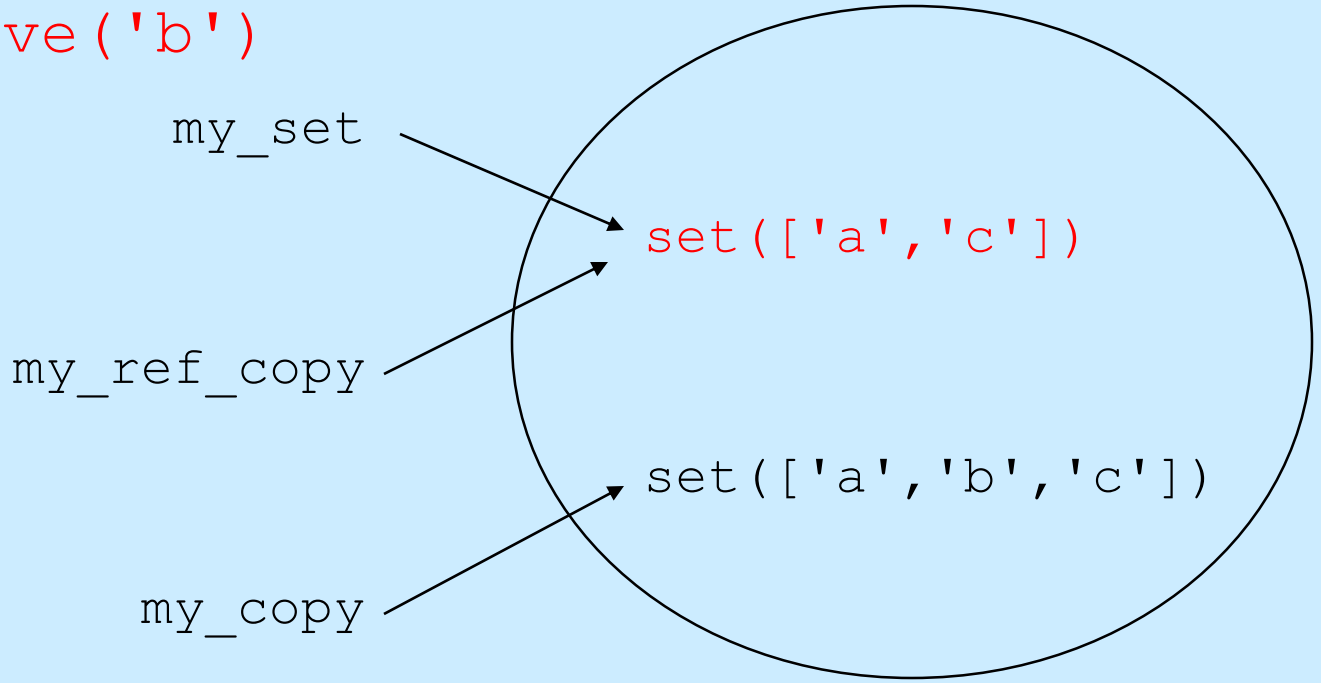

```
small_set <= big_set → True
big_set >= small_set → True
```

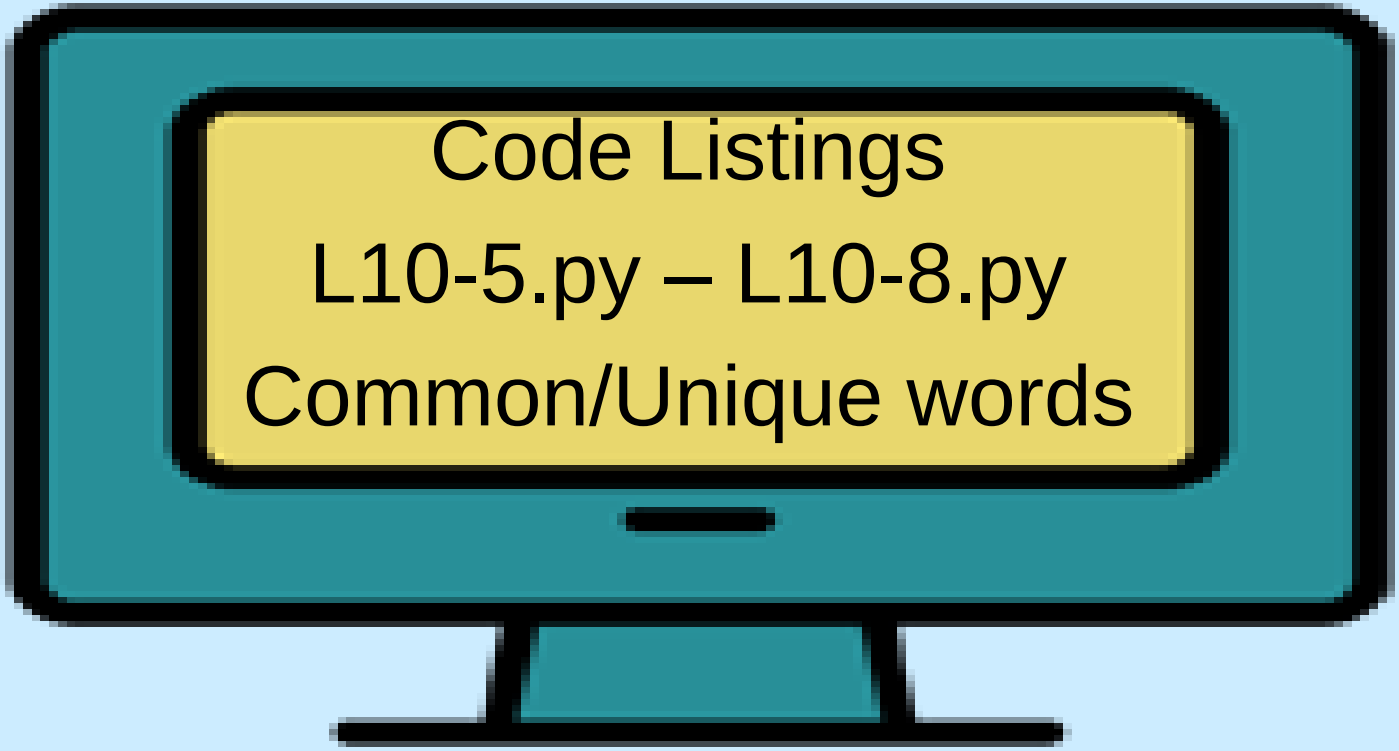
Other Set Operations

- `my_set.add("g")`
  - adds to the set, no effect if item is in the set already
- `my_set.clear()`
  - empties the set
- `my_set.remove("g")` versus `my_set.discard("g")`
  - `remove` throws an error if "`g`" isn't there. `discard` doesn't care. Both remove "`g`" from the set
- `my_set.copy()`
  - returns a shallow copy of `my_set`

Copy vs. assignment

```
my_set=set {'a', 'b', 'c'}
my_copy=my_set.copy()
my_ref_copy=my_set
my_set.remove('b')
```




Code Listings
L10-5.py – L10-8.py
Common/Unique words

4 functions

- `add_word(word, word_set)`. Add word to the set (instead of dict). No return.
- `process_line(line, word_set)`. Process line and identify words. Calls `add_word`. No return. (no change except for parameters)
- `pretty_print(word_set)`. Nice printing of the various set operations. No return
- `main()`. Function to start the program.

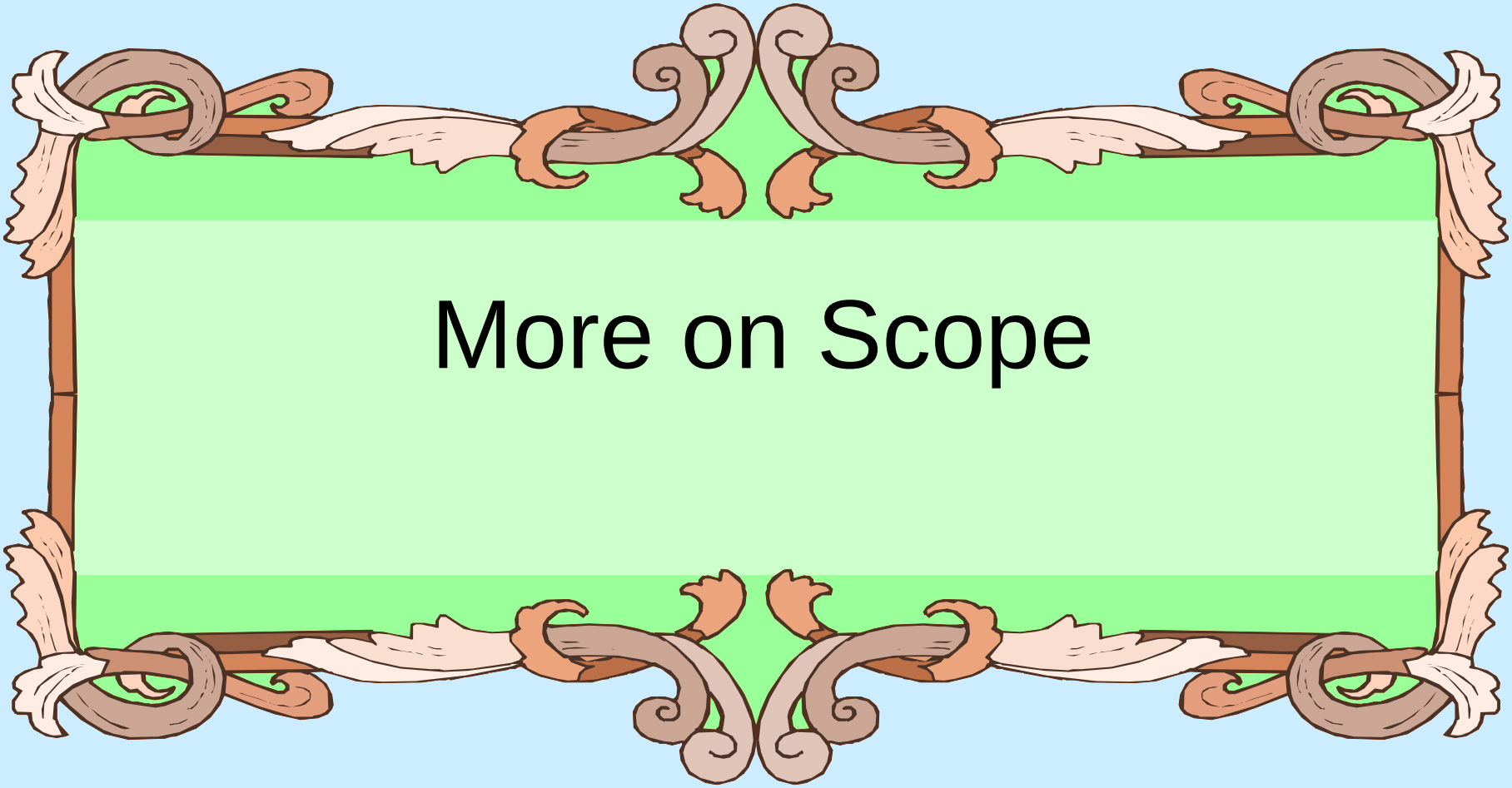
```
1 def add_word(word, word_set):
2     '''Add the word to the set. No word smaller than length 3.'''
3     if len(word) > 3:
4         word_set.add(word)
```

```
import string
def process_line(line, word_set):
    '''Process the line to get lowercase words to be added to the set.'''
    line = line.strip()
    word_list = line.split()
    for word in word_list:
        # ignore the '--' that is in the file
        if word != '--':
            word = word.strip()
            # get commas, periods and other punctuation out as well
            word = word.strip(string.punctuation)
            word = word.lower()
            add_word(word, word_set)
```

more complicated pretty print

- the `pretty_print` function applies the various set operators to the two resulting sets
- prints, in particular, the intersection in a nice format
- should this have been broken up into two functions??

```
def pretty_print(ga_set, doi_set):
    # print some stats about the two sets
    print('Count of unique words of length 4 or greater')
    print('Gettysburg Addr: {}, Decl of Ind: {}\n'.format(len(ga_set),len(
doi_set)))
    print('{:15s} {:15s}'.format('Operation', 'Count'))
    print('-'*35)
    print('{:15s} {:15d}'.format('Union', len(ga_set.union(doi_set))))
    print('{:15s} {:15d}'.format('Intersection', len(ga_set.intersection(
doi_set))))
    print('{:15s} {:15d}'.format('Sym Diff', len(ga_set.symmetric_difference(
doi_set))))
    print('{:15s} {:15d}'.format('GA-DoI', len(ga_set.difference(doi_set))))
    print('{:15s} {:15d}'.format('DoI-GA', len(doi_set.difference(ga_set))))

    # list the intersection words, 5 to a line, alphabetical order
    intersection_set = ga_set.intersection(doi_set)
    word_list = list(intersection_set)
    word_list.sort()
    print('\n Common words to both')
    print('-'*20)
    count = 0
    for w in word_list:
        if count % 5 == 0:
            print()
        print('{:13s}'.format(w), end=' ')
        count += 1
```


More on Scope

OK, what is a namespace

- We've had this discussion, but lets' review
- A namespace is an association of a name and a value
- It looks like a dictionary, and for the most part it is (at least for modules and classes)

Scope

- What namespace you might be using is part of identifying the scope of the variables and function you are using
- by "scope", we mean the context, the part of the code, where we can make a reference to a variable or function

Multiple scopes

- Often, there can be multiple scopes that are candidates for determining a reference.
- Knowing which one is the right one (or more importantly, knowing the order of scope) is important

Two kinds

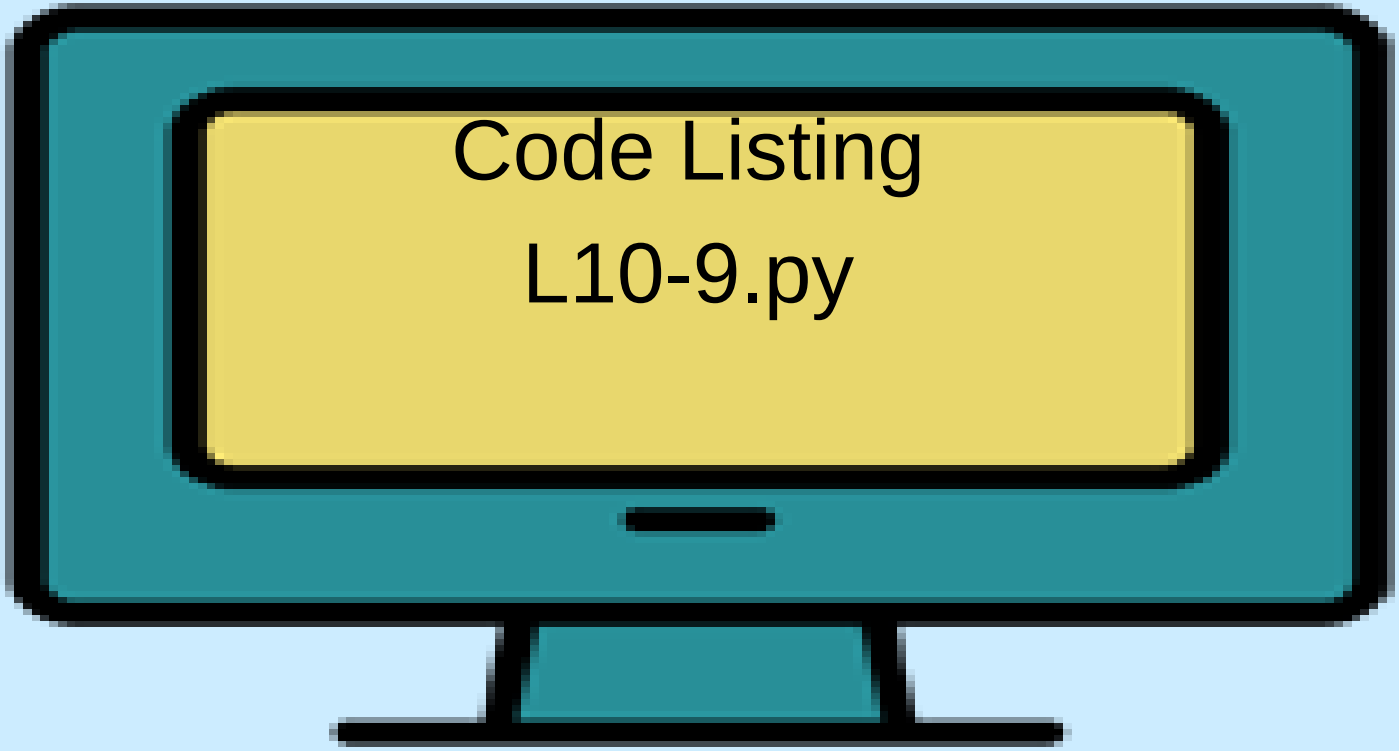
- ***Unqualified namespaces***. This is what we have pretty much seen so far. Functions, assignments etc.
- ***Qualified namespaces***. This is modules and classes

Unqualified

- this is the standard assignment and def we have seen so far
- Determining the scope of a reference identifies what its true 'value' is

unqualified follow the LEGB rule

- ***local***, inside the function in which it was defined
- if not there, ***enclosing/encomposing***. Is it defined in an enclosing function
- if not there, is it defined in the ***global*** namespace
- finally, check the ***built-in***, defined as part of the special builtin scope
- else ERROR


Code Listing
L10-9.py

`locals()` function

Returns a dictionary of the current (presently in play) local namespace. Useful for looking at what is defined where.

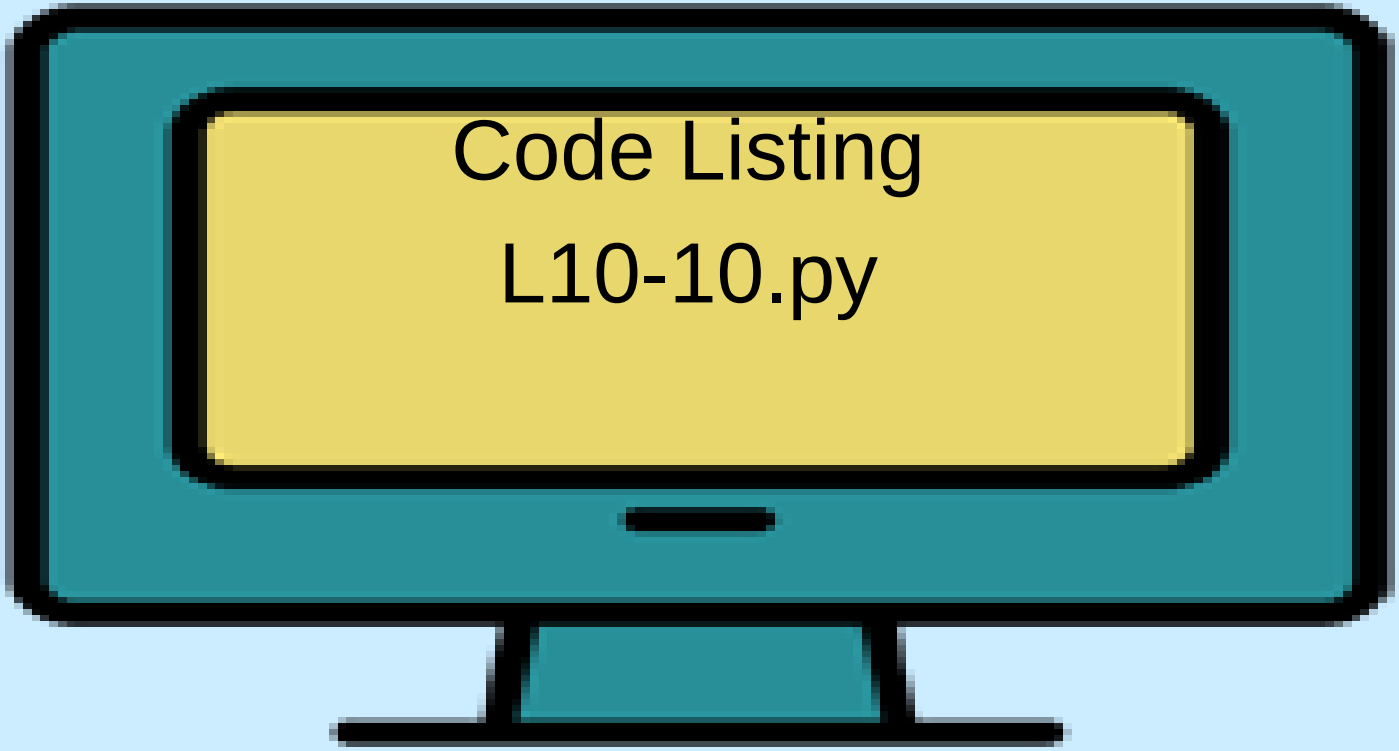
function local values

- if a reference is assigned in a function, then that reference is only available within that function
- if a reference with the same name is provided outside the function, the reference is reassigned

```
global_X = 27

def my_function(param1=123, param2='hi mom'):
    local_X = 654.321
    print('\n=== local namespace ===')
    for key,val in locals().items():
        print('key:{}, object:{}'.format(key, str(val)))
    print('local_X:',local_X)
    print('global_X:',global_X)

my_function()
```

```
=== local namespace ===
key:local_X, object:654.321
key:param1, object:123
key:param2, object:hi mom
local_X: 654.321
global_X: 27
```

global is still found because of the sequence of namespace search


Code Listing
L10-10.py

globals() function

Like the `locals()` function, the `globals()` function will return as a dictionary the values in the global namespace

```
import math
global_X = 27

def my_function(param1=123, param2='hi mom'):
    local_X = 654.321
    print('\n=== local namespace ===')
    for key,val in locals().items():
        print('key: {}, object: {}'.format(key, str(val)))
    print('local_X:',local_X)
    print('global_X:',global_X)

my_function()

key,val = 0,0  # add to the global namespace. Used below
print('\n--- global namespace ---')

for key,val in globals().items():
    print('key: {:15s} object: {}'.format(key, str(val)))

print('\n-----------------------')
#print 'Local_X:', local_X
print('Global_X:', global_X)
print('Math.pi:',math.pi)
print('Pi:',pi)
```

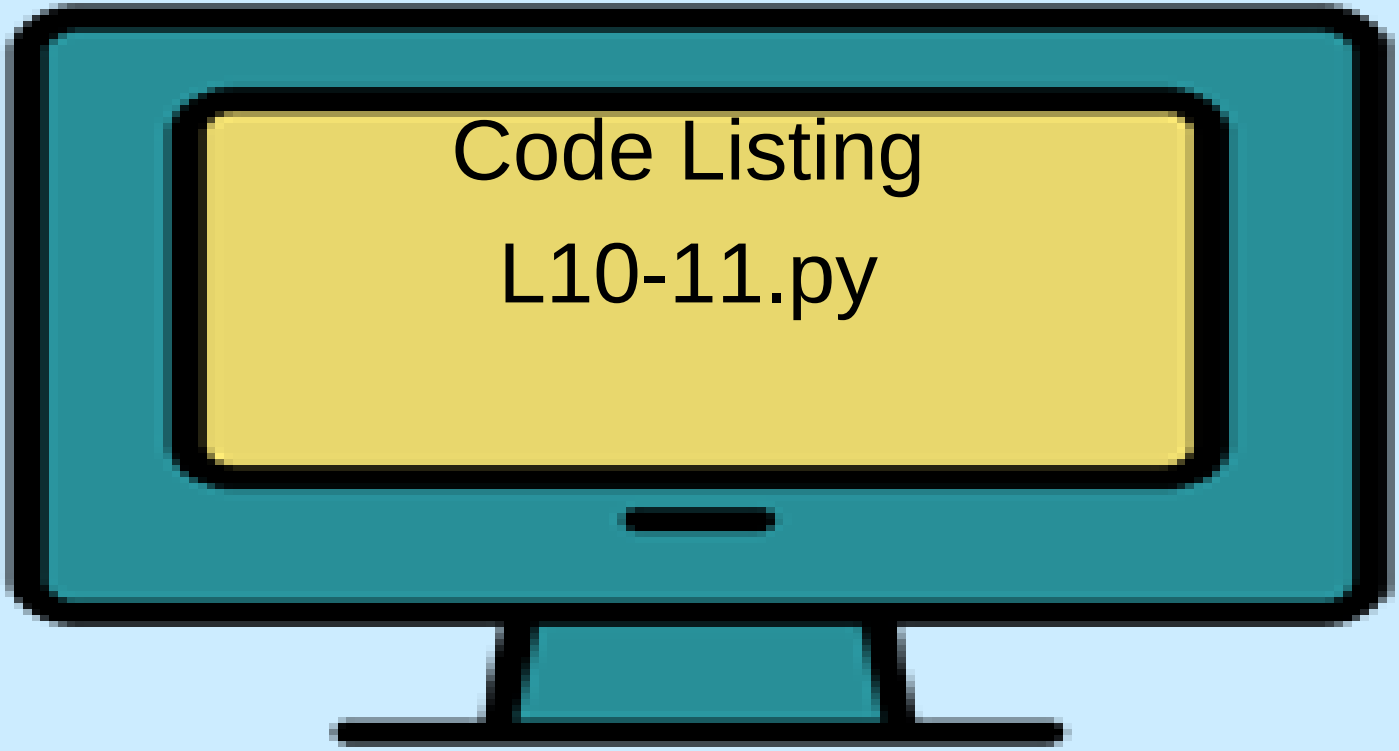
```
=== local namespace ===
key: local_X, object: 654.321
key: param1, object: 123
key: param2, object: hi mom
local_X: 654.321
global_X: 27

--- global namespace ---
key: my_function     object: <function my_function at 0xe15a30>
key: __builtins__    object: <module '__builtin__' (built-in)>
key: __package__     object: None
key: global_X        object: 27
key: __name__        object: __main__
key: __doc__         object: None
key: math            object: <module 'math' from '/Library/Frameworks/Python.
framework/Versions/3.2/lib/python3.2/lib-dynload/math.so'>

----------------------
Global_X: 27
Math.pi: 3.14159265359
Pi:

Traceback (most recent call last):
  File "/Volumes/Admin/Book/chapterDictionaries/localsAndGlobals.py", line 22, in
<module>
    print('Pi:',pi)
NameError: name 'pi' is not defined
```

Global Assignment Rule

A quirk of Python.

If an assignment occurs ***anywhere*** in the suite of a function, Python adds that variable to the local namespace

- means that, even if the variable is assigned later in the suite, the variable is still local


Code Listing
L10-11.py

```
my_var = 27

def my_function(param1=123, param2='Python'):
    for key,val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1      # causes an error!

my_function(123456, 765432.0)
```

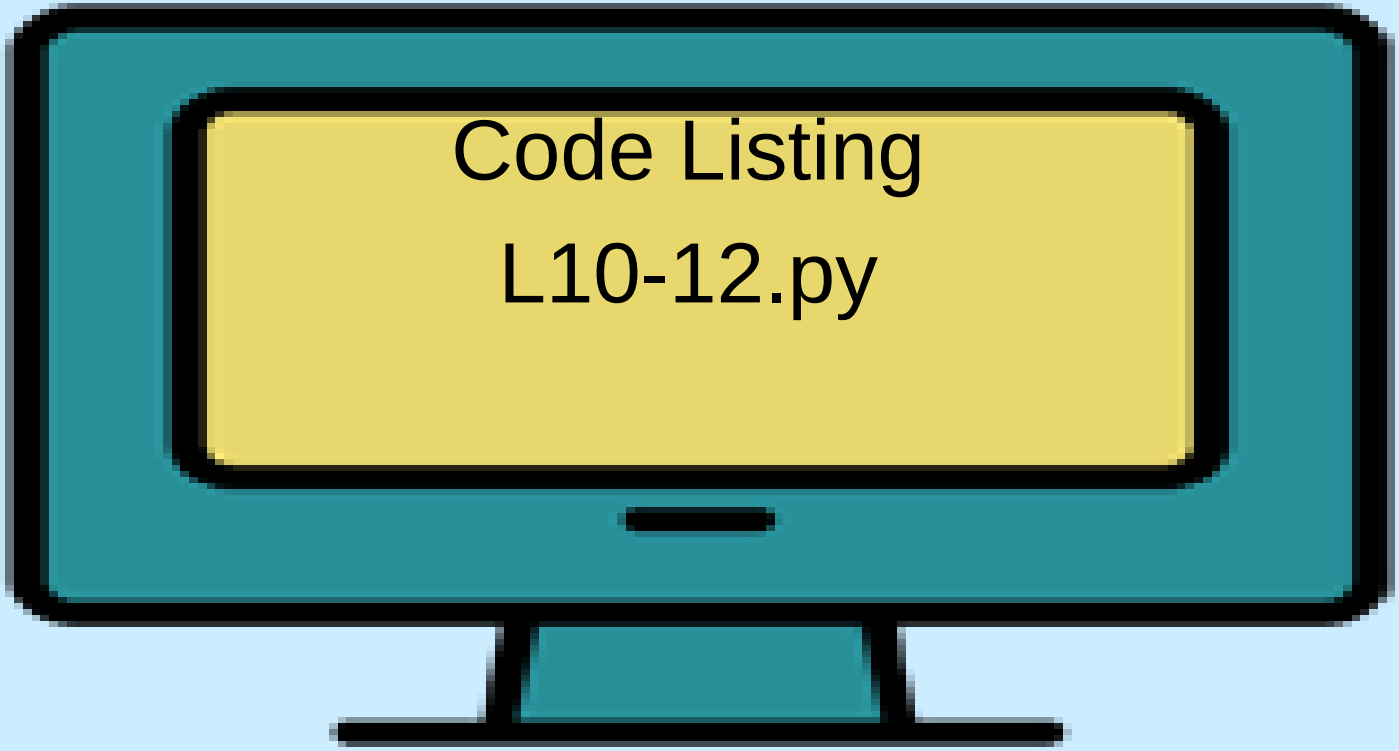
```
key param2: 765432.0
key param1: 123456
Traceback (most recent call last):
  File "localAssignment1.py", line 9, in <module>
    my_function(123456, 765432.0)
  File "localAssignment1.py", line 7, in my_function
    my_var = my_var + 1      # causes an error!
UnboundLocalError: local variable 'my_var' referenced before assignment
```

`my_var` is local (is in the local namespace) because it is assigned in the suite

the global statement

You can tell Python that you want the object associated with the global, not local namespace, using the `global` statement

- avoids the local assignment rule
- should be used carefully as it is an over-ride of normal behavior


Code Listing
L10-12.py

```
my_var = 27

def my_function(param1=123, param2='Python'):
    for key,val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1      # causes an error!

def better_function(param1=123, param2='Python'):
    global my_var
    for key,val in locals().items():
        print('key {}: {}'.format(key, str(val)))
    my_var = my_var + 1
    print('my_var:',my_var)



# my_function(123456, 765432.0)
better_function()
```

```
key param2: Python
key param1: 123
my_var: 28
```

`my_var` is not in the local namespace

Builtin

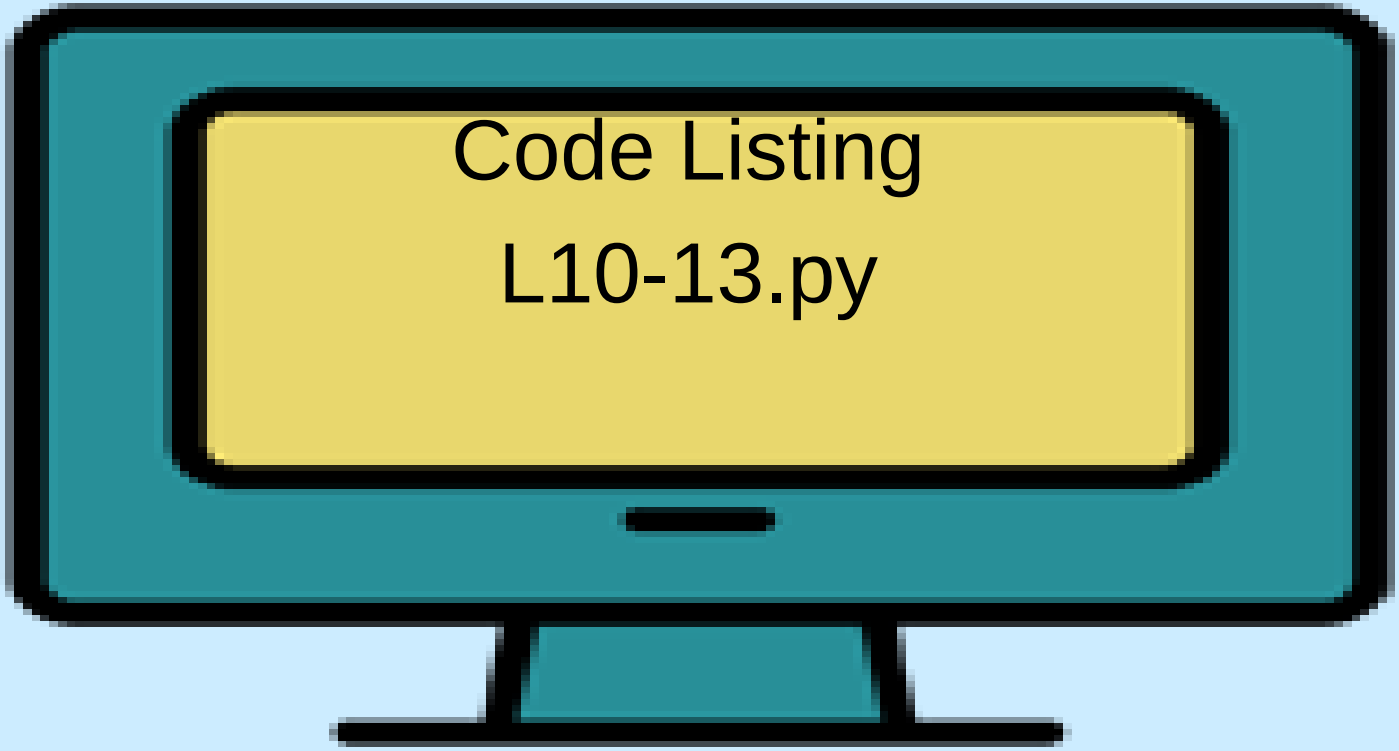
- This is just the standard library of Python.
- To see what is there, look at

```
import __builtin__
dir(__builtin__)
```

Enclosed

Functions which define other functions in a function suite are ***enclosed***, defined only in the enclosing function

- the inner/enclosed function is then part of the local namespace of the outer/enclosing function
- remember, a function is an object too!


Code Listing
L10-13.py

```
global_var = 27

def outer_function(param_outer = 123):
    outer_var = global_var + param_outer

    def inner_function(param_inner = 0):
        # get inner, enclosed and global
        inner_var = param_inner + outer_var + global_var

        # print inner namespace
        print('\n--- inner local namespace ---')
        for key,val in locals().items():
            print('{}:{}'.format(key,str(val)))
        return inner_var

    result = inner_function(outer_var)
    # print outer namespace
    print('\n--- outer local namespace ---')
    for key,val in locals().items():
        print('{}:{}'.format(key,str(val)))
    return result

result  = outer_function(7)
print('\n--- result ---')
print('Result:',result)
```

```
--- inner local namespace ---
outer_var:34
inner_var:95
param_Inner:34

--- outer local namespace ---
outer_var:34
param_Outer:7
result:95
inner_function:<function inner_function at 0xe2ba30>

--- result ---
Result: 95
```

Building dictionaries faster

- `zip` creates pairs from two parallel lists
  - `zip("abc",[1,2,3])` yields

    `[('a',1),('b',2),('c',3)]`
- That's good for building dictionaries. We call the `dict` function which takes a list of pairs to make a dictionary
  - `dict(zip("abc",[1,2,3]))` yields
  - `{'a': 1, 'c': 3, 'b': 2}`

dict and set comprehensions

Like list comprehensions, you can write shortcuts that generate either a dictionary or a set, with the same control you had with list comprehensions

- both are enclosed with `{}` (remember, list comprehensions were in `[]`)
- difference is if the collected item is a : separated pair or not

dict comprehension

```
>>> a_dict = {k:v for k,v in enumerate('abcdefg')}
>>> a_dict
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g'}
>>> b_dict = {v:k for k,v in a_dict.items()} # reverse key—value pairs
>>> b_dict
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'g': 6, 'f': 5}
>>> sorted(b_dict)     # only sorts keys
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> b_list = [(v,k) for v,k in b_dict.items()]   # create list
>>> sorted(b_list)                               # then sort
[('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4), ('f', 5), ('g', 6)]
```

set comprehension

```
>>> a_set = {ch for ch in 'to be or not to be'}
>>> a_set
{' ', 'b', 'e', 'o', 'n', 'r', 't'}  # set of unique characters
>>> sorted(a_set)
[' ', 'b', 'e', 'n', 'o', 'r', 't']
```