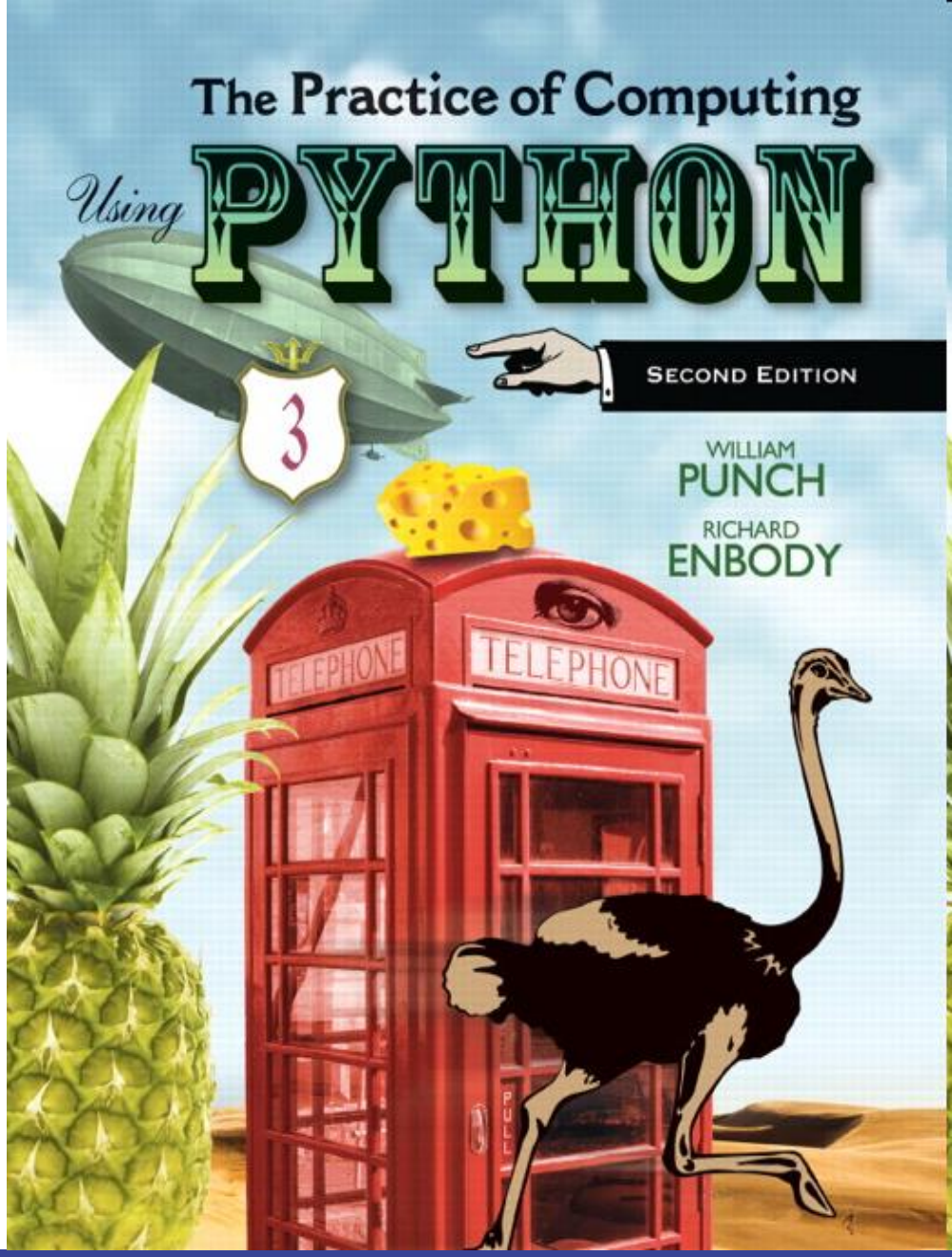
Sets and Dictionaries



More Data Structures

- We have seen the list data structure and what it can be used for
- We will now examine two more data structures: **Set** and **Dictionary**

Sets

Remember our first set-based code

```
dompet = {1000, 10000, 20000, 20000, 20000, 20000, 20000}

print("Uang di dompet:", dompet)
```

Suppose you are building a virtual money application. What is the problem with the above code?

Remember our first set-based code

```
dompet = {1000, 10000, 20000, 20000, 20000, 20000, 20000}

print("Uang di dompet:", dompet)
```

Suppose you are building a virtual money application. What is the problem with the above code?

```
OUTPUT:
Uang di dompet: {1000, 10000, 20000}
```

Now you see it :)

Sets, as in Mathematical Sets

- In maths, 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**
- There is **no order** to the elements of a set

Note:
A set with no elements is the empty set

Students' Music Genres

Tell me your favorite music genres?

PS: Don't care about the counts, just want to know what are the music genres of students.

Students' Music Genres

Tell me your favorite music genres?

PS: Don't care about the counts, just want to know what are the music genres of students.
PSS: The music genres will be stored as a set.

Creating a set

Set can be created in one of 2 ways:

- Constructor: `set(iterable)`

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

- Shortcut: `{}`

```
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}
```

No duplicates

- Duplicates are automatically removed

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

→ `{'a', 'c', 'b', 'd'}`

Live coding: Set of integers

```
my_set = set([6,1,4,6,1,1,2])

print(my_set)
```

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`
  - indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`

Live coding: Print unique words (*Oops!*)

```
a_str = "rock pop rock jazz rock rock dangdut"

a_set = set(a_str)

print(a_set)
```

Live coding: Print unique words

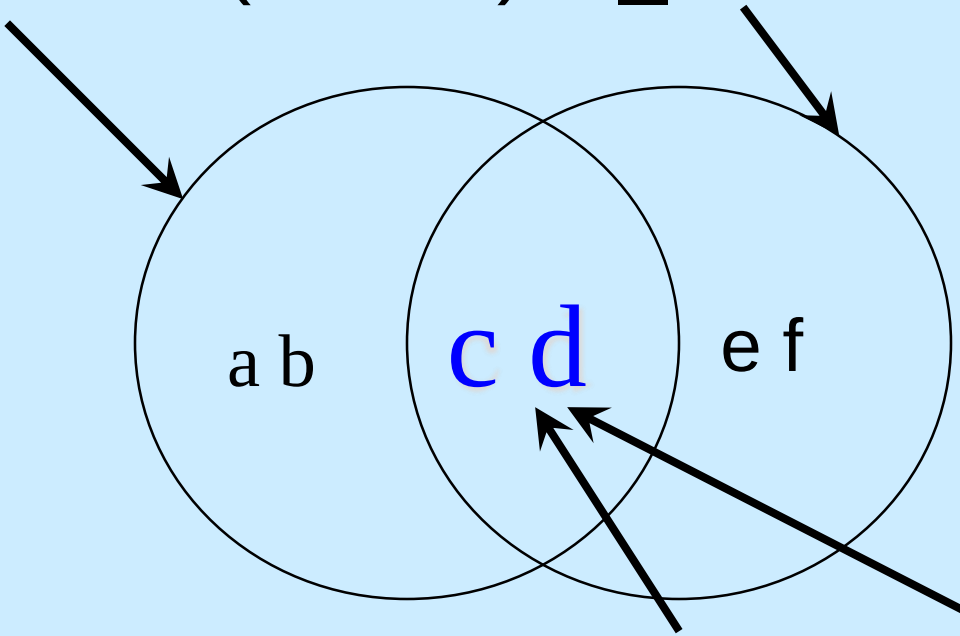
```
a_str = "rock pop rock jazz rock rock dangdut"

a_set = set(a_str.split())

print(a_set)
```

Set operators

- The set data structure provides some special operators
- These are various combinations of set contents
- These operations have both a method name and a shortcut binary operator

method: intersection, op: &

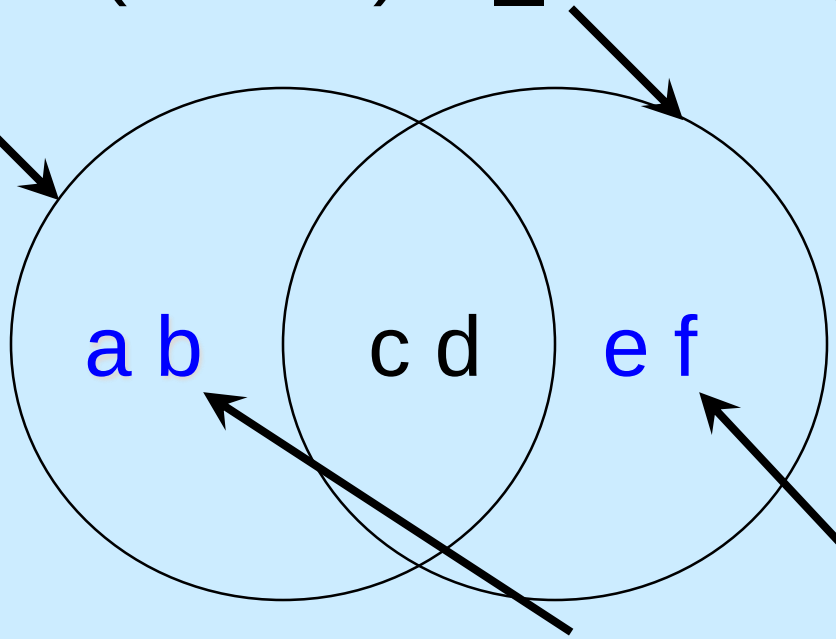
a_set=set("abcd") b_set=set("cdef")




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

method:difference op: -

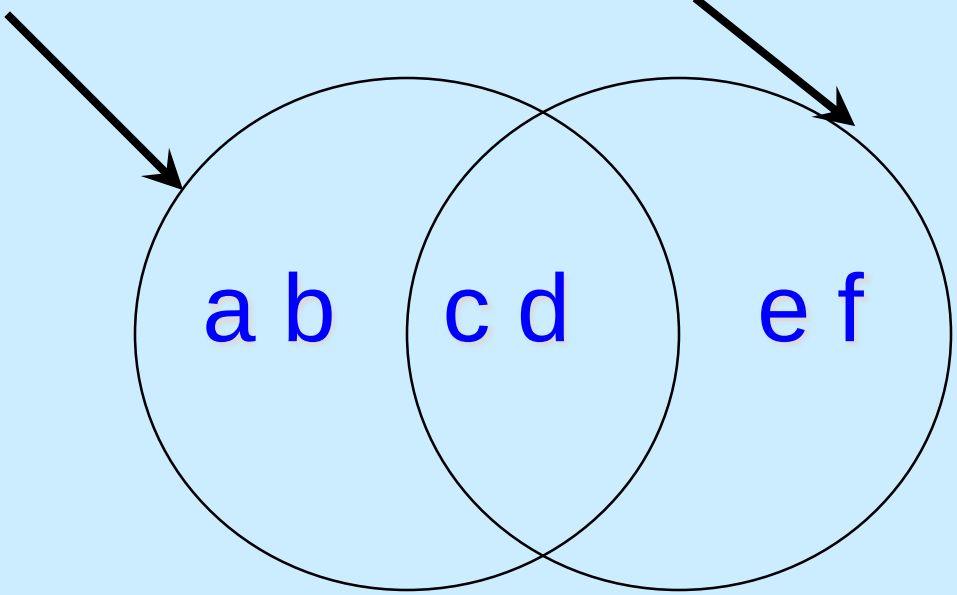
a_set=set("abcd") b_set=set("cdef")




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

method: union, op: |

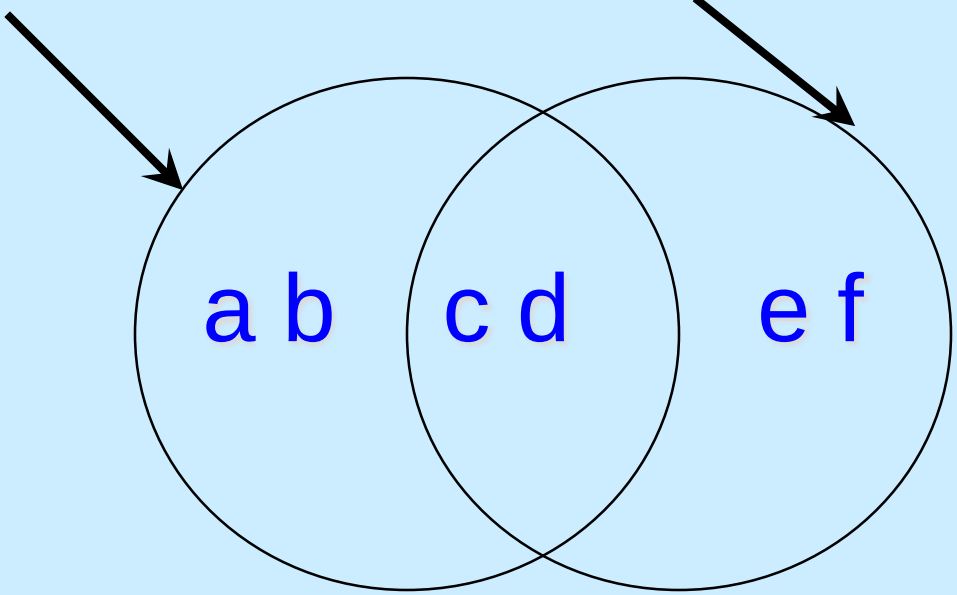
a_set=set("abcd") b_set=set("cdef")




```
a_set | b_set → {'a', 'b', 'c', 'd', 'c', 'd',
'e', 'f'}
```

method: union, op: |

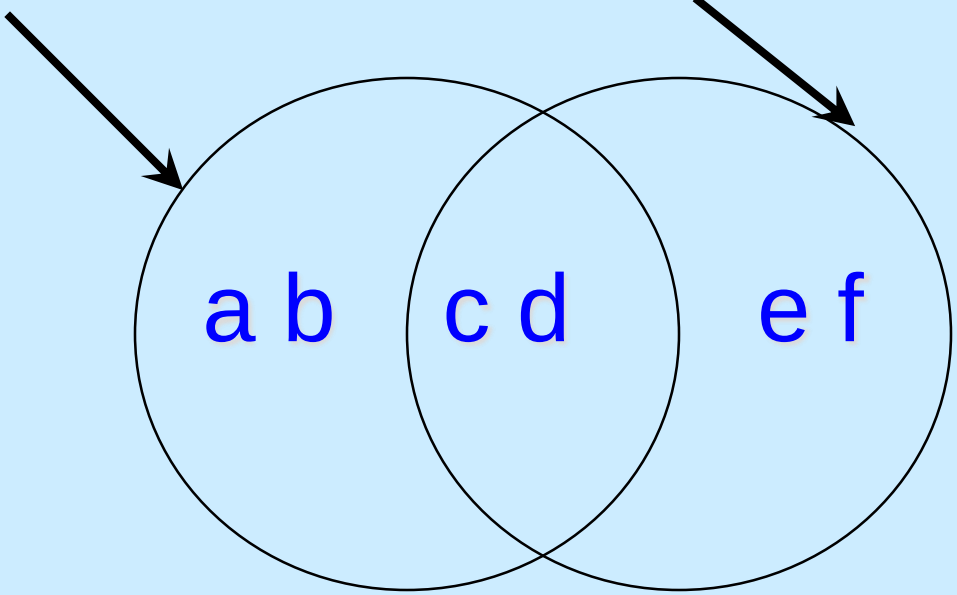
a_set=set("abcd") b_set=set("cdef")




```
a_set | b_set → {'a', 'b', 'c', 'd', 'c', 'd', 'e', 'f'}
```

method: union, op: |

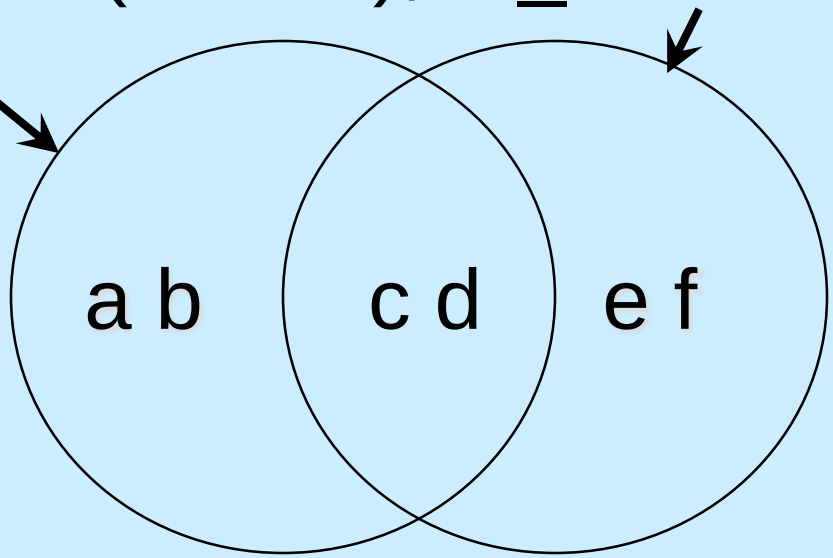
a_set=set("abcd") b_set=set("cdef")




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

method:symmetric_difference*, op: ^

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'}
```

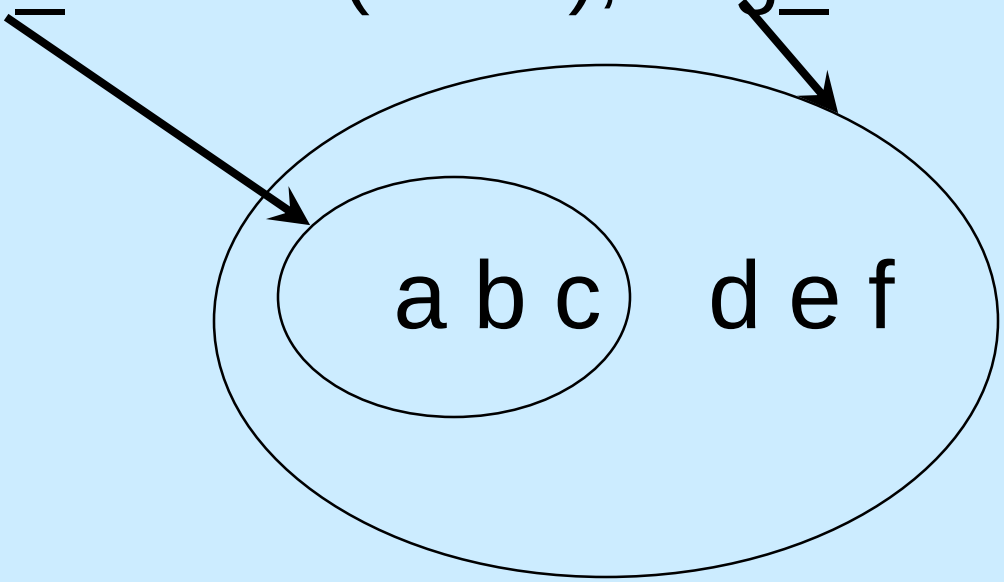
* in a or b but **not both**

Live coding: Set operators

```
a = set('abracadabra')
b = set('alacazam')
print(a - b)
print(a | b)
print(a & b)
print(a ^ b)
```

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

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




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

Live coding: The empty set

```
a = set()
b = set('alacazam')
print(a <= a)
print(a <= b)
```

Other Set Ops

- `my_set.add("g")`
  - adds to the set, no effect if item is in set already
- `my_set.update((1,1,2))`
  - adds each element of the argument
- `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`

What about indexing and slicing?

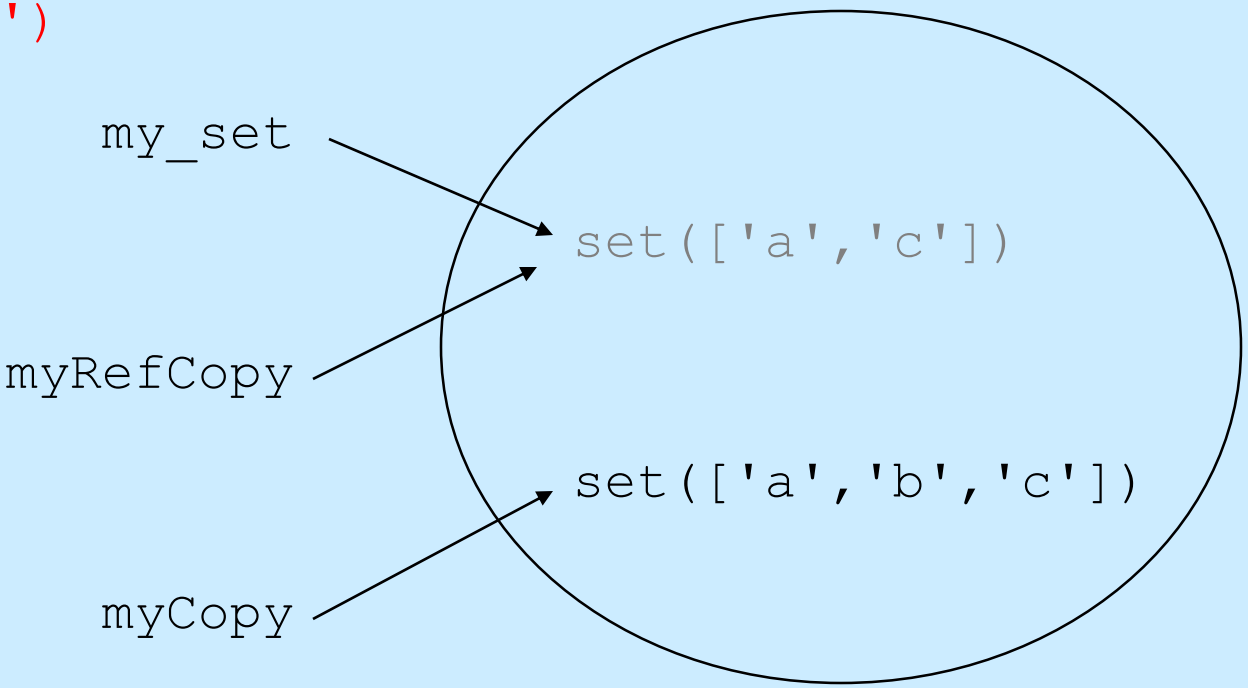
- `Try this: {1,2,3}[0]`
- `And this: {1,2,3}[:]`

Live coding: Other set ops

```
my_set = {1,3}
my_set.update((1,1,2))
print(my_set)

my_set = {1,3}
my_set.add((1,1,2))
print(my_set)
```

Copy vs. assignment

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



Set Comprehension

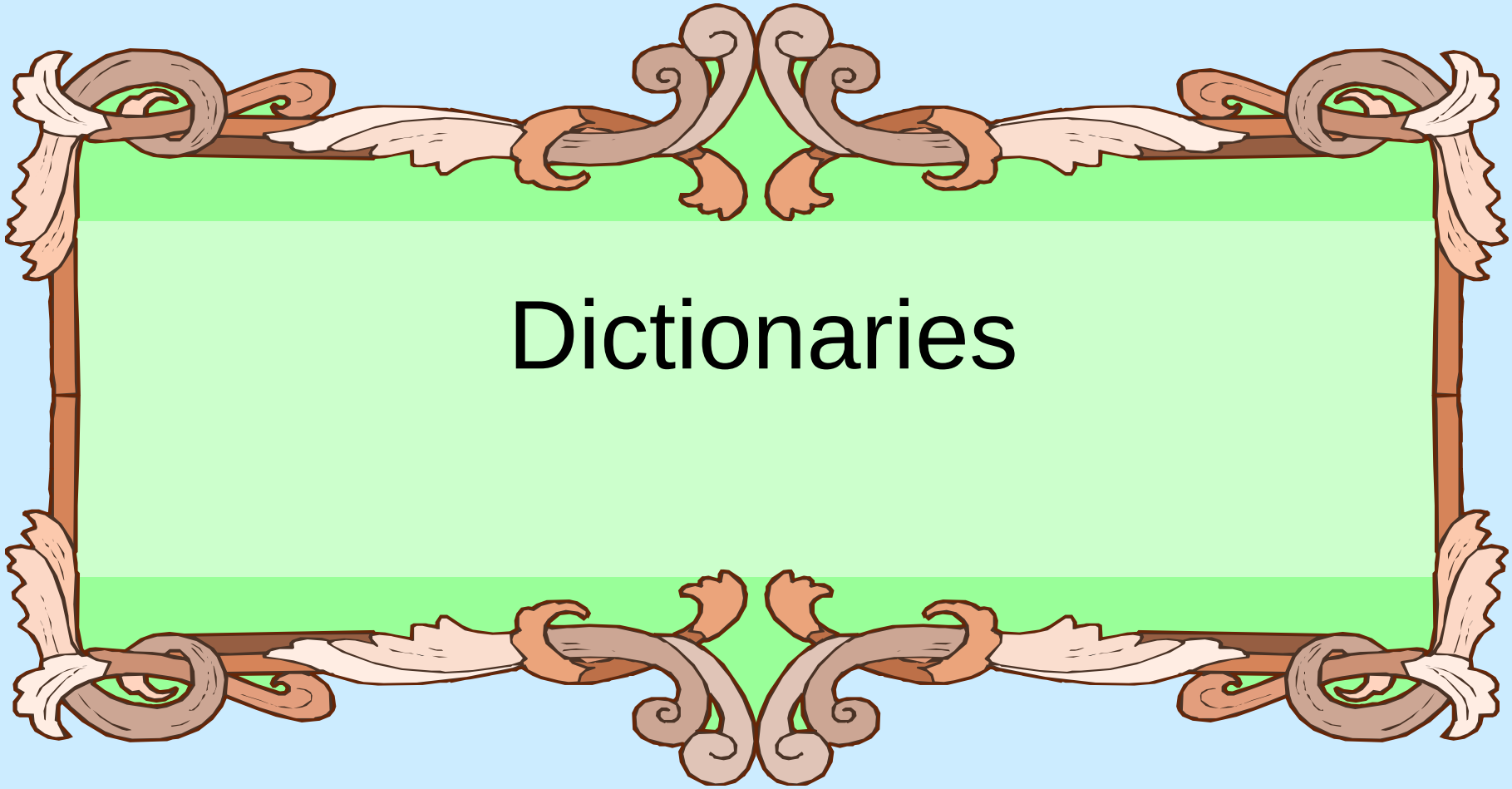
```
>>> a_set = {ch for ch in 'to be or not to be'}
>>> a_set
{' ', 'b', 'e', 'o', 'n', 'r', 't'}  # set of unique characters
>>> sorted(a_set)
[' ', 'b', 'e', 'n', 'o', 'r', 't']
```

Live coding: Number of distinct numbers

```
def num_distinct(list_integers):
```

Live coding: Number of distinct numbers

```
def num_distinct(list_integers):
        return(len(set(list_integers)))
```



Students' favorite food

{

}

What is a dictionary?

- Other names: *associative array, associative list* or a *map*.
- It is a list of **key:value** pairs.
- Yes, we ***map a key to a value***
- Note here the key is used as an index (compare it to list, what is used as an index in a list?)

Python Dictionary

- Use the { } marker to create a dictionary (similar to set but here we use colons!)
- 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': '369-1234'}
```

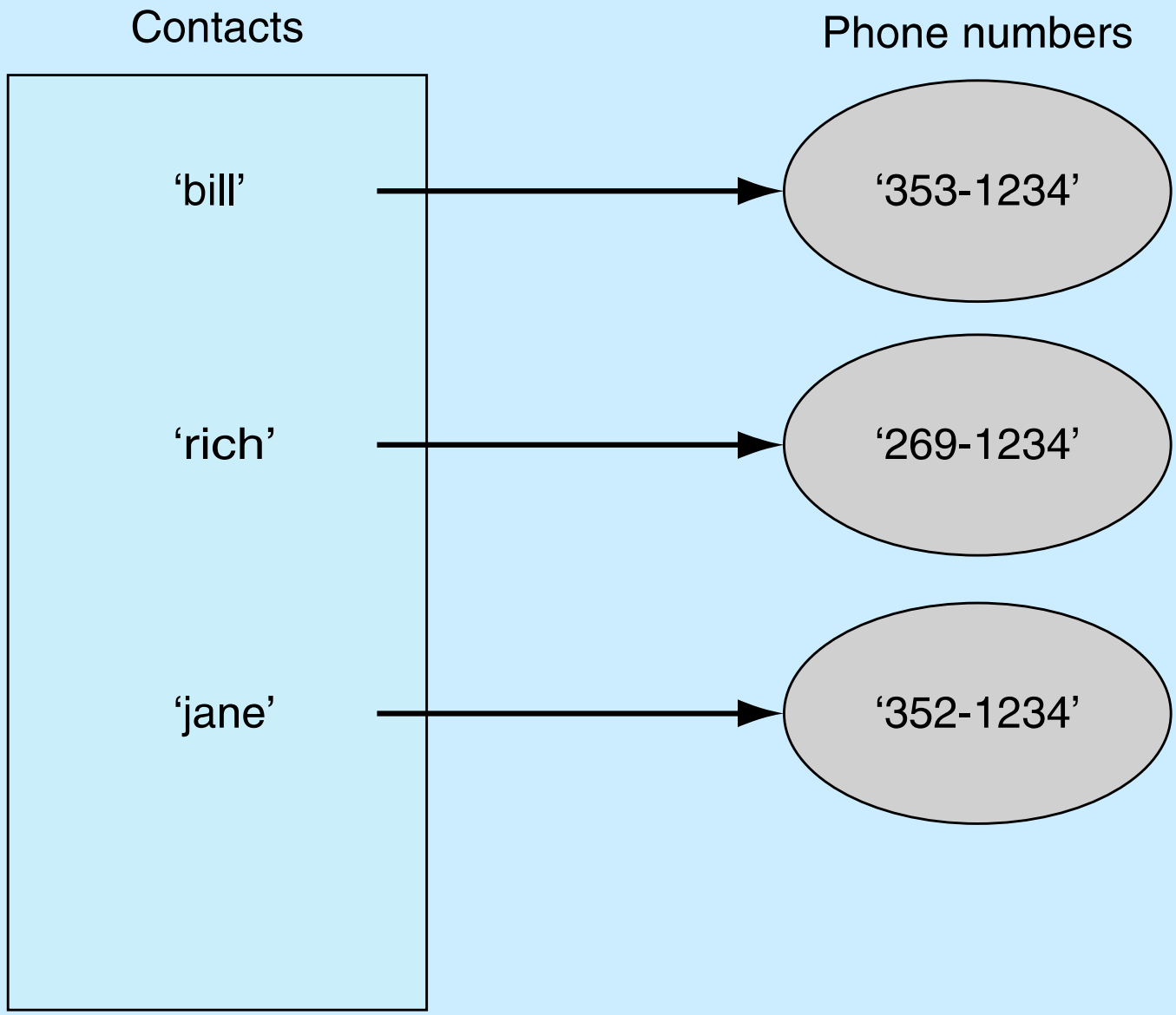
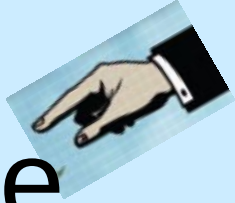


FIGURE 9.1 Phone contact list: names and phone numbers.

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['rich'])
my_dict['bill'] = 100
print(my_dict['bill'])
```

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]


demo[(2,4)]


demo ['x']


demo['x'][1]
```

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

All returns objects of class View!

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
>>>
```

3 ways of computing frequency of words in list using dictionary

Do it yourself first

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
```

Passing mutables

- When 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.

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 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)]
```

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 (we'll talk more about this one later in the classes section)

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

`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

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

```
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

```
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 builtins
print(dir(builtins))
```

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!

```
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
```