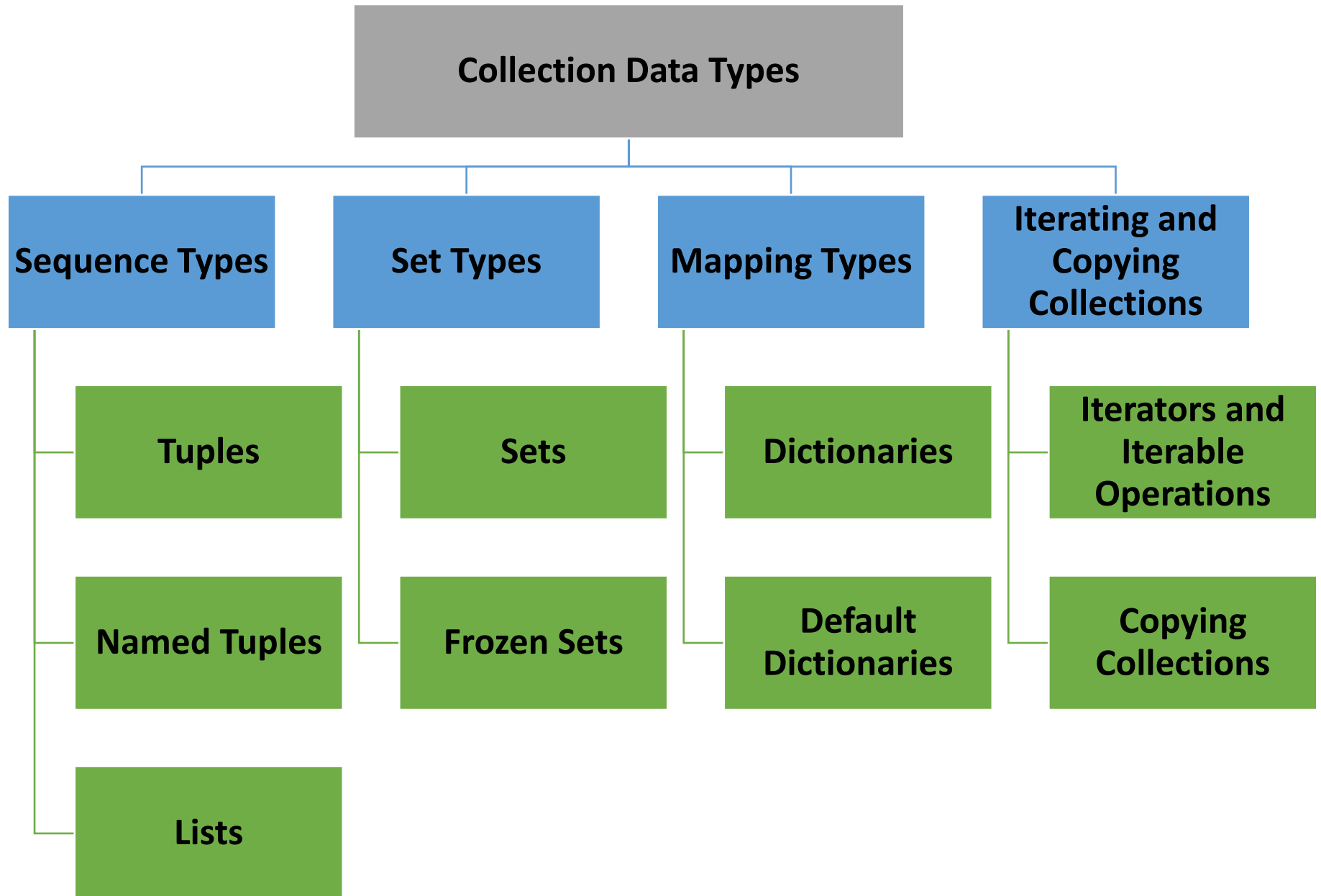


# Dictionaries and Sets

## Collection Data Types

Dr. M. Prabu, Dr. M. Umme Salma



Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	<code>[ ]</code> or <code>list()</code>	<code>[5.7, 4, 'yes', 5.7]</code>
Tuple	Yes	No	<code>( )</code> or <code>tuple()</code>	<code>(5.7, 4, 'yes', 5.7)</code>
Set	No	Yes	<code>{ }*</code> or <code>set()</code>	<code>{5.7, 4, 'yes'}</code>
Dictionary	No	Yes**	<code>{ : }</code> or <code>dict()</code>	<code>{ 'Jun': 75, 'Jul': 89 }</code>

# 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 of 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***

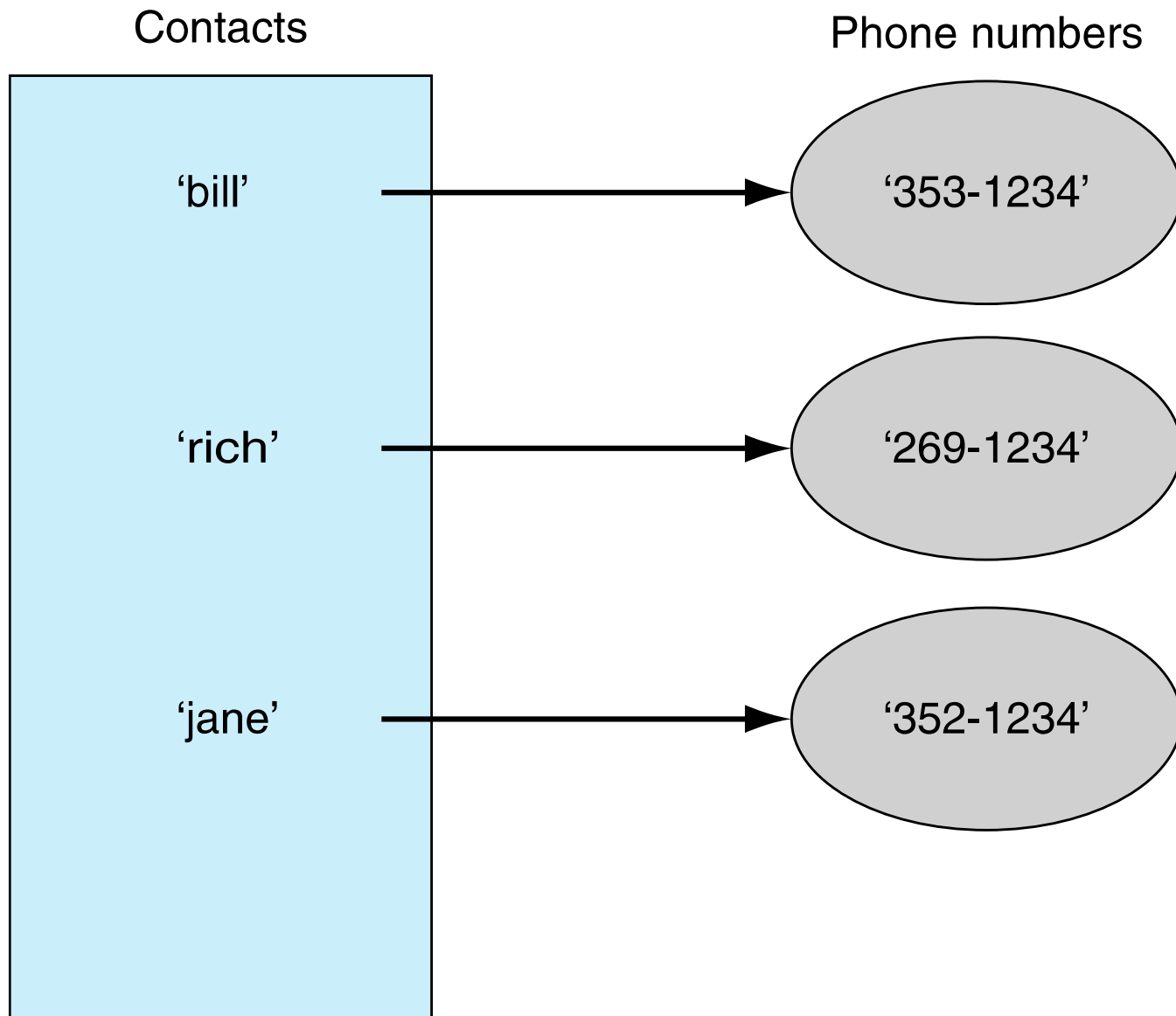
# Key Value pairs

- 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': '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['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
```

# 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.



```
1 def add_word(word, word_count_dict):  
2     '''Update the word frequency: word is the key, frequency is the value.'''  
3     if word in word_count_dict:  
4         word_count_dict[word] += 1  
5     else:  
6         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)
```

# 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()
>>> null_set
set()
>>> a_set = {1,2,3,4}
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)}
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")
>>> a_set
{'a', 'c', 'b', 'd'}
```

*# set() creates the empty set*

*# no colons means set*

*# duplicates are ignored*

*# different types is OK*

*# set constructed from iterable*

*# order not maintained!*

# common operators

Most data structures respond to these:

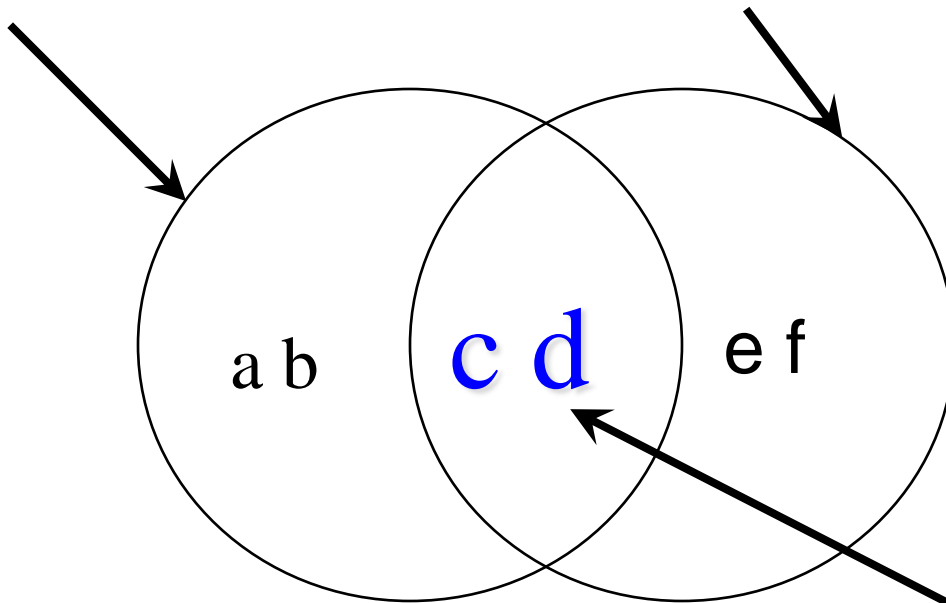
- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - boolean indicating whether element is in the set
- `for element in my_set:`
  - iterate through the elements in `my_set`

# 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, op: &

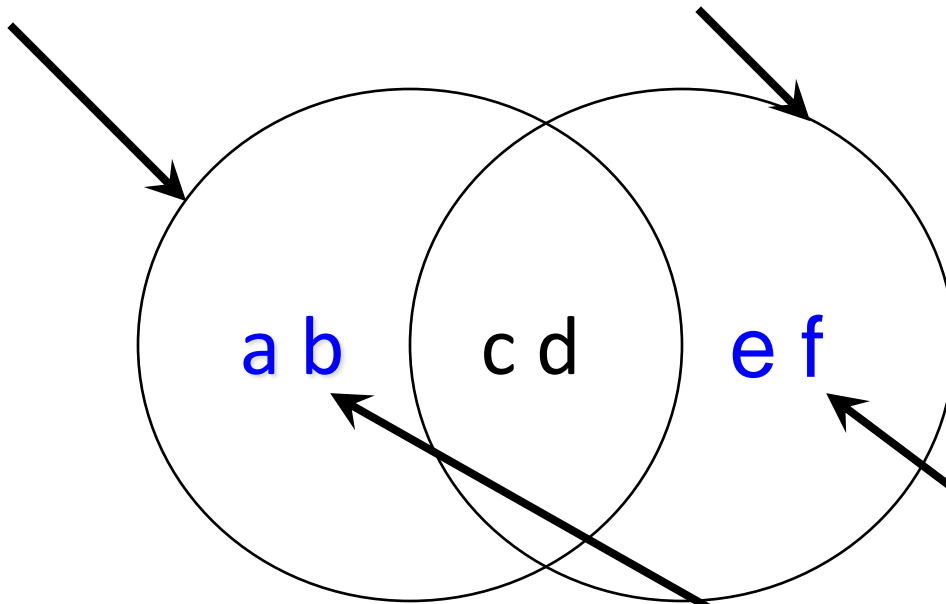
```
a_set=set("abcd") b_set=set("cdef")
```



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

method: difference op: -

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

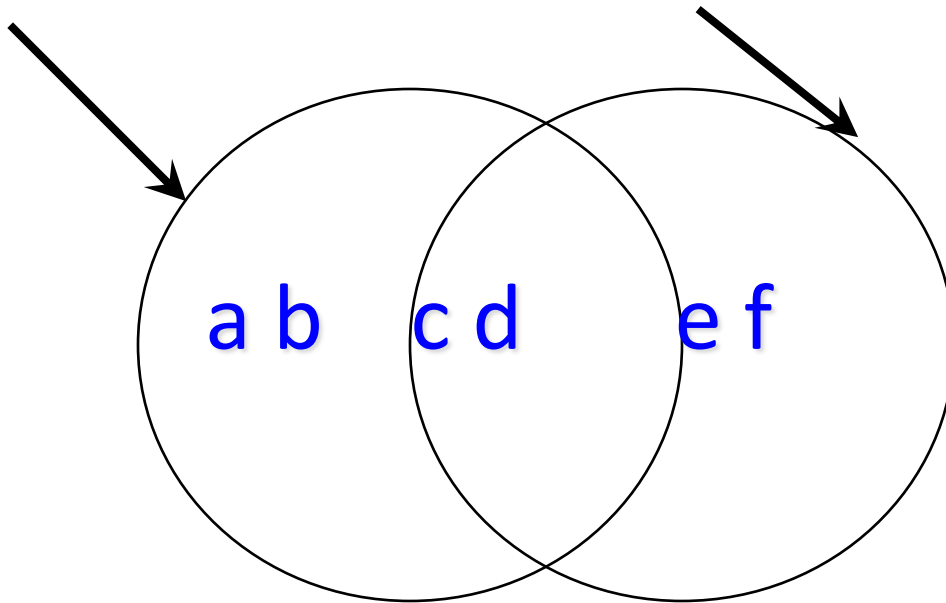


```
a_set - b_set → {'a', 'b'}
```

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

method: union, op: |

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

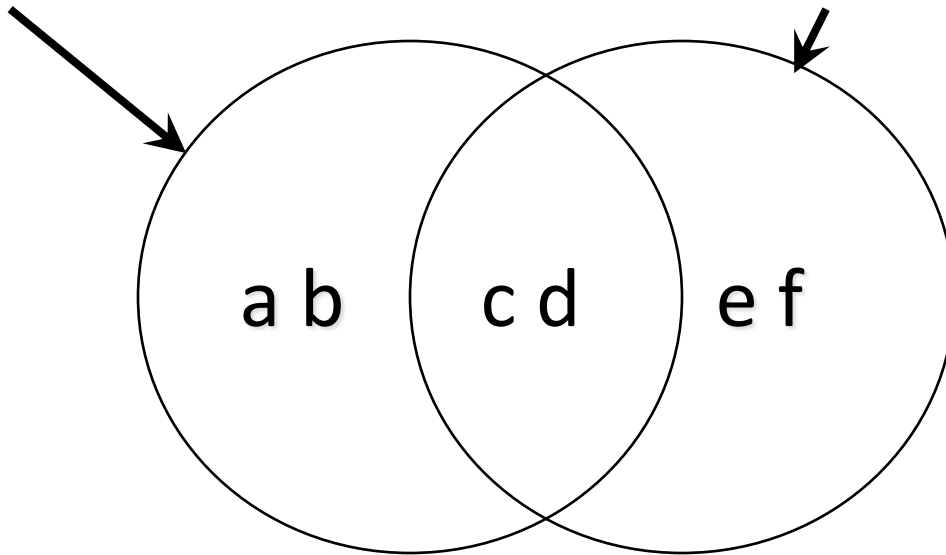


```
a_set | b_set → {'a', 'b', 'c', 'd', 'e', 'f'}  
b_set.union(a_set) → {'a', 'b', 'c', 'd', 'e', 'f'}
```



method:symmetric\_difference, op: ^

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

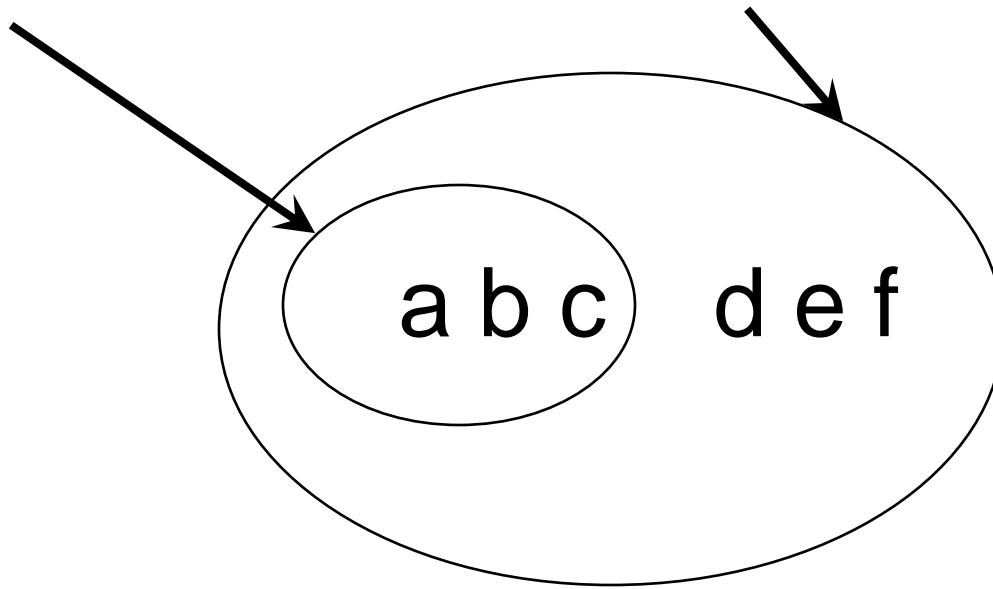


```
a_set ^ b_set → {'a', 'b', 'e', 'f'}
```

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

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

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



```
small_set <= big_set → True
```

```
big_set >= small_set → True
```

# Other Set Ops

- `my_set.add("g")`
  - adds to the set, no effect if item is in 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`

# Frozen Sets

Though sets can't contain mutable objects, sets are mutable:

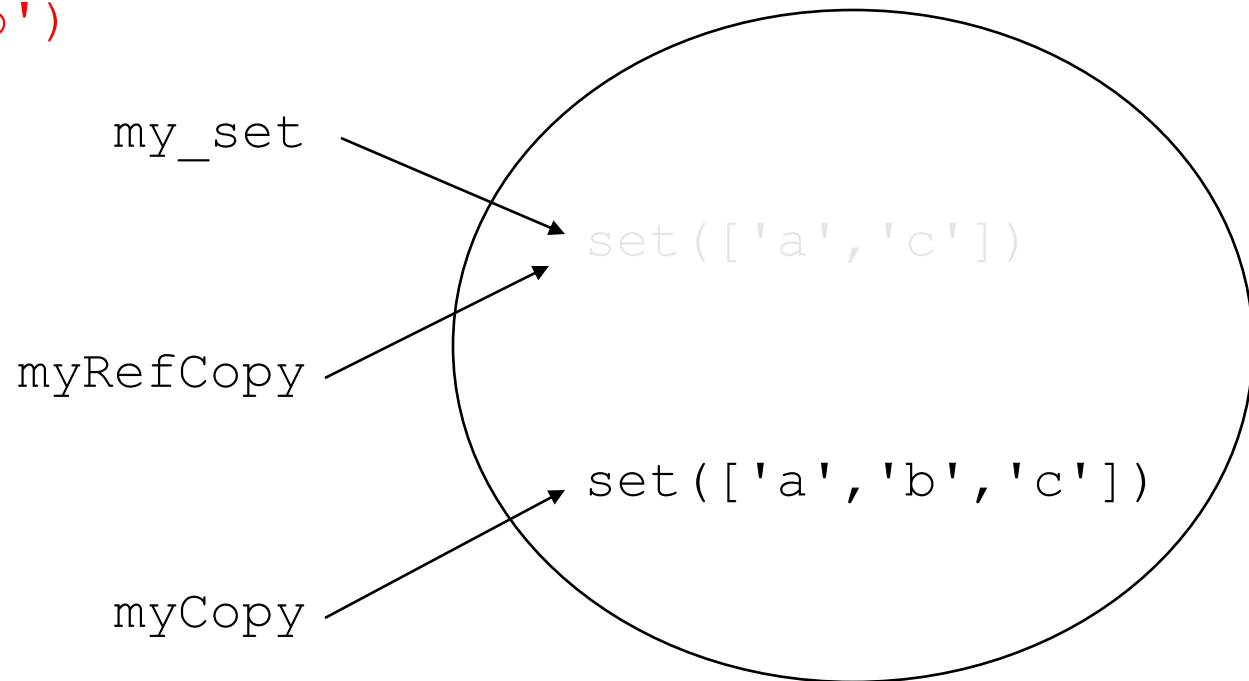
```
>>> cities = set(["Frankfurt",  
"Basel", "Freiburg"])  
>>> cities.add("Strasbourg")  
>>> cities  
set(['Freiburg', 'Basel', 'Frankfurt',  
'Strasbourg'])  
>>>
```

Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

```
>>> cities = frozenset(["Frankfurt",  
"Basel", "Freiburg"])  
>>> cities.add("Strasbourg")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'frozenset' object has no  
attribute 'add'  
>>>
```

# Copy vs. assignment

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



## **What is For Loop?**

For loop is used to iterate over elements of a sequence. It is often used when you have a piece of code which you want to repeat "n" number of time.

## **What is While Loop?**

While Loop is used to repeat a block of code. Instead of running the code block once, It executes the code block multiple times until a certain condition is met.