Strings

Strings

- **STRING** --- an **ordered** collection of characters to *store* and *represent* text-based information

- Python strings are categorized as *immutable sequences* --- meaning they have a **left-to-right order** (sequence) and *cannot be changed in place* (immutable)

Single- and Double-Quoted

- Single- and Double-Quoted strings **are the same**

>>> ‘Hello World’ , “Hello World”

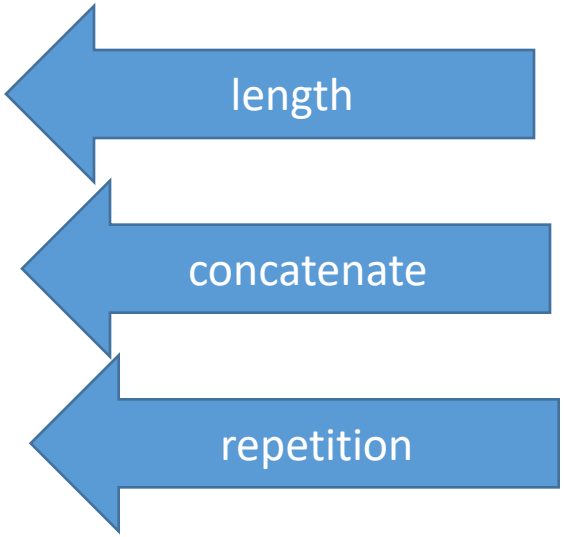
- The reason for including both is that it allows you to embed a quote character of the other inside a string

>>> “knight’s” , ‘knight”s’

Strings in Action

- **Basic operations:**

1. len()
2. +
3. *



len()

- The len build-in function returns the length of strings

```
>>> len('abc')
```


```
>>> a='abc'
>>> len(a)
```

+

- Adding two string objects creates a new string object

```
>>> ‘abc’ + ‘def’
```

```
>>> a=‘Hello’
>>> b=‘World’
>>> a + b
>>> a+ ‘ ’ +b
```

Concatenation of strings

- Repetition may seem a bit obscure at first, but it comes in handy in a surprising number of contexts
- For example, to print a line of 80 dashes

>>> print ‘-’ * 80

Check String using ‘in’ and ‘not in’

```
>>> strng="There is no substitute for hard work"
>>> print('hard' in strng)
True
>>> if 'of' not in strng:
  print("The word 'of' is not present in the string")

The word 'of' is not present in the string
```

Slices and Indexes

Indexes in slices

- Characters in a string are numbered with *indexes* starting at 0:
  - Example:

```
name = "ABCDEFGH"
```

index	0	1	2	3	4	5	6	7
character	A	B	C	D	E	F	G	H

- Accessing an individual character of a string:

  variableName `[` ***index*** `]`

  - Example:

```
print name, "starts with", name[0]
```

  Output:

```
ABCDEFGH starts with A
```

More examples on Index and slice

```
>>> a="COLLEGE"
>>> a[0], a[-2]
        ('C','G')
>>> A[2:4],A[2:],A[:4],A[:-2],A[1:5:2]
('LL', 'LLEGE', 'COLL', 'COLLE', 'OL')
```

String Methods: modifying and checking strings assigned to variables

- Assigning a string to a variable

  strng="aurangabad"

- strng.title() 'Aurangabad'
- strng.upper() 'AURANGABAD'
- strng.lower() 'áurangabad'
- strng.isdigit() False
- strng.islower() True
- Len() len(strng)→ 10
- Str() str(100)→ '100'

String Methods: modifying and checking strings assigned to variables

```
                    strng="   Good Morning  "
strng.strip()          'Good Morning'
strng.replace()
       strng.replace("Morning","Evening")
       'Good Evening'
Strng.split()
       strng="Good Morning"
       >>> strng.split(" ")
       ['Good', 'Morning']
```

Containers

Python Objects: Lists, Tuples, Dictionaries

- Lists (mutable sets of strings)
  - `var = [] # create list`
  - `var = ['one', 2, 'three', 4]`
- Tuples (immutable sets)
  - `var = ('one', 2, 'three', 4)`
- Dictionaries (associative arrays or 'hashes')
  - `var = {} # create dictionary`
  - `var = {'one': 1, 'two': 2}`
  - `var['two'] = 2`

- Each has ***its own set of methods***

Tuples, Lists, and Strings:

Similarities

Similar Syntax of tuples and lists

- **Containers** are any object that holds an arbitrary number of other objects. Generally, **containers** provide a way to access the contained objects and to iterate over them.
- Tuples and lists are sequential containers that share much of the same syntax and functionality.

How Tuples, Lists, and Strings are defined

Defining Tuples

- **Tuples** are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

Defining Lists

- **Lists** are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

Defining Strings

- **Strings** are defined using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

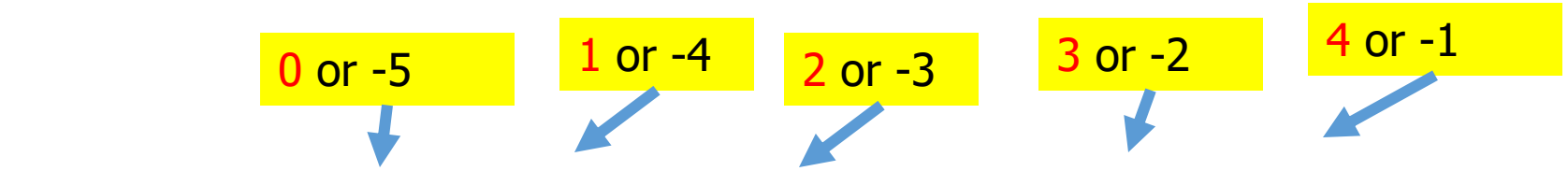
Individual access in Tuples, Lists, and Strings

- We can **access individual members** of a tuple, list, or string using square bracket "array" notation.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34
>>> st = "Hello World"
>>> st[1]   # Second character in string.
 'e'
```

This is similar to strings that we discussed last time

Looking up an Item in a tuple from start and end




```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset, part 1

Value of variable t is a tuple:

```
>>> t=(23,'abc',4.56,(2,3),'def')
```

Return a *copy of the container* with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset, part 2

```
>>> t=(23,'abc',4.56,(2,3),'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

The 'in' Operator in containers

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, (2,3),4, 5]
>>> 3 in t
False
>>> (2,3) in t
True
>>> 4 not in t
 False
```

In in strings and containers

- The **in** keyword checks if the given object is contained within the aggregate.
- >>> strng="College"
- >>> 'e' in strng
- True
- >>> 'x' in strng
- False
- >>> 'eg' in strng
- True
- >>> strng=['strength','string']
- >>> 'string' in strng
- True
- >>> 'strength'in strng[0:]
- True

Range function

- Python has a range function to easily *form lists of integers.*

```
>>> range(5)
[0, 1, 2, 3, 4]
```

[0, 1, 2, 3, 4]

```
>>> range(2,5)
[2, 3, 4]
```

[0, 1, 2, 3, 4]

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
>>> range(5, 0, -1)
[5, 4, 3, 2, 1]
```

Count down

The + Operator

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)
```

( ) for tuples

```
>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]
```

[ ] for lists

```
>>> "Hello" + " " + "World"
 'Hello World'
```

" " for strings

The * Operator

- The * operator produces a **new tuple, list, or string** that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Lists

Lists are the most flexible containers

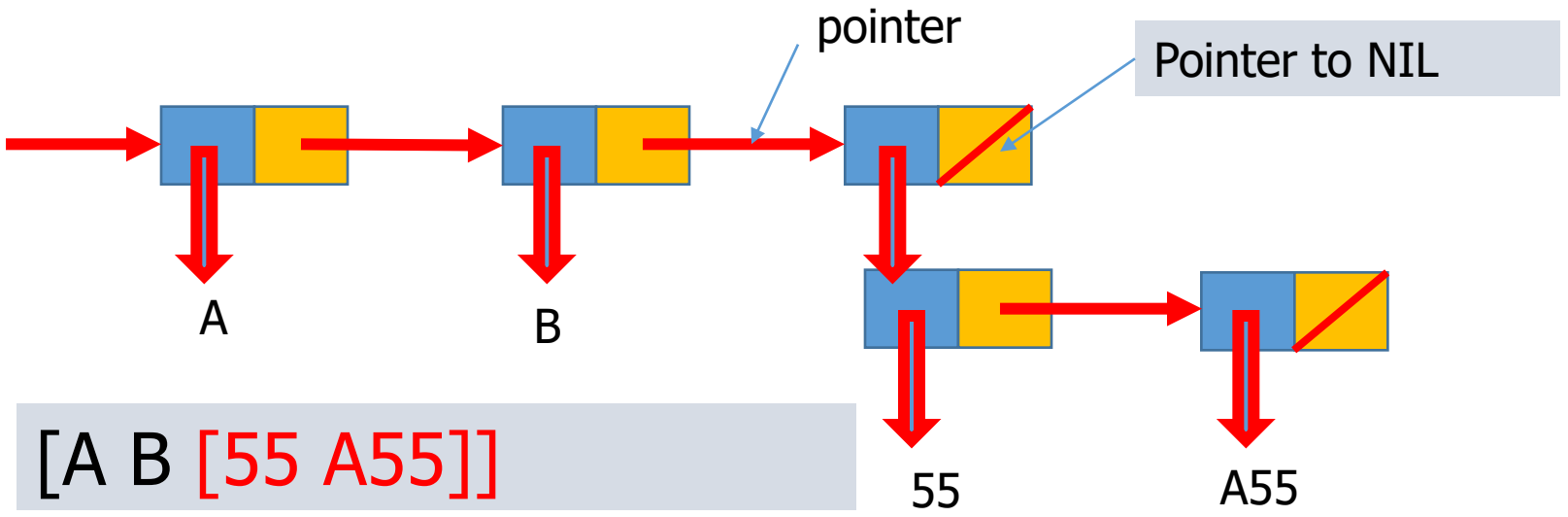
- Lists are Python's most flexible **ordered** collection object type

- Lists can contain any sort of object:
  - numbers,
  - strings
  - and even other lists

Lists are ordered places for objects

- **Ordered collections of arbitrary objects**

1. from the functional view, lists are just a place to collect other objects
2. Lists also define a left to right positional ordering of the items in the list



Schemata like this will help us to understand the concepts of mutable and immutable objects

Lists are mutable

- Ordered collection of data
- Data can be of different types
- Lists are ***mutable***
- Lists have the same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]

>>> x
[1, 'hello', (3+2j)]

>>> x[2]
(3+2j)

>>> x[0:2]
[1, 'hello']
```

Using lists

- **Accessed by offset**

--- you can fetch a component object out of a list by indexing the list on the object's offset

--- you can also use lists for such tasks as slicing and concatenation

Declaring Lists

- >>> aa=[] #An empty list

- >>> aa=[1,2,3,4] #4 items, index 0-3
- >>> aa=list()

  >>> aa

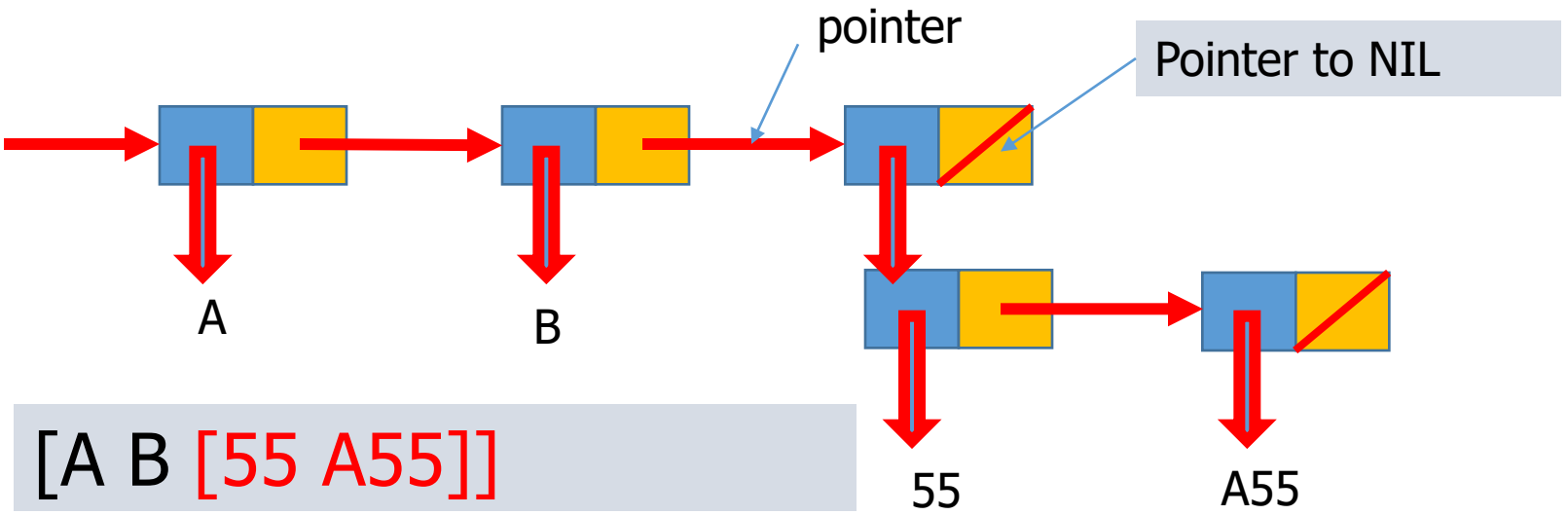
  []

Lists can grow and shrink, have any objects

- **Variable length, heterogeneous, arbitrary nestable**

--- Unlike strings, list can grow and shrink in place

--- lists may contain any sort of object, not just one-character strings (they are heterogeneous)



List in action

- List respond to the + and * operations much like strings

```
>>> aa=[1,2,3]
>>> bb=[4,5,6]
>>> aa+bb


>>> aa*3
```

List in action: len and in

- Lists also have the function len() to tell the size of lists and “in” function

```
>>> aa=[1,2,3]
>>> len(aa)                # test of len

>>> 3 in aa                # test of in
```

Append

List method calls

The list append method simply appends a single item onto the end of the list

```
>>> aa=[]
>>> aa.append(1)
>>> aa.append(2)
>>> aa.append(3)
>>> aa.append('4')
>>> aa.append(5.0)
```

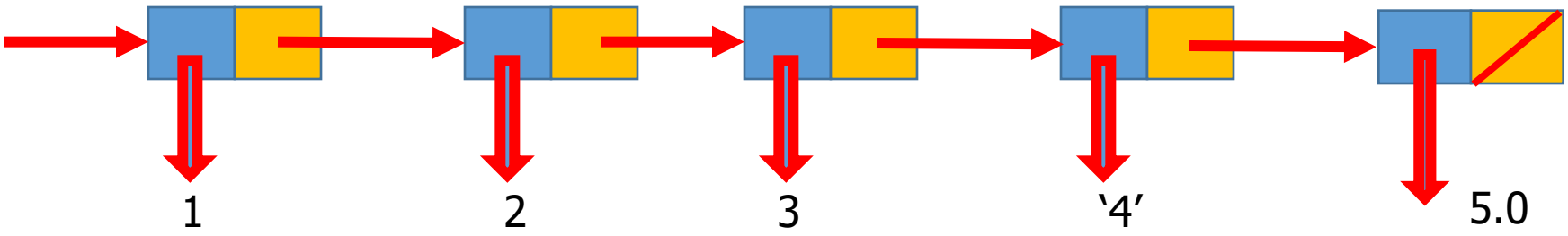
[1]

[1, 2]

[1, 2, 3]

[1, 2, 3, '4']

[1, 2, 3, '4', 5.0]



Summary of important operations on lists.

- **l=[23,56,3,67,89]**
- **Removing from the List**
  - var[*n*] = [] *empties contents of card, but preserves order*
    - l[2]=[]
    - >>> l
    - [23, 56, [], 67, 89]
  - var.remove(*n*)
    - >>>l.remove(56)
    - >>> l
    - [23, [], 67, 89]
    - removes card at *n*
  - var.pop(*n*) removes *n* and returns its value
    - >>> l.pop(1)
    - []
    - >>> l
    - [23, 67, 89]

More examples of operations on lists

- List:
  - A container that holds a number of other objects, in a given order
  - Defined in square brackets

```
a = [1, 2, 3, 4, 5]

print a[1]   # number 2
some_list = []
some_list.append("two")
some_list.insert(1,12)
>>> some_list.index(12)
1
```

```
[]
```

```
["two"]
```

```
["two". 12]
```

More operators on Lists: del and slices

- a = [98, "bottles of buttermilk", ["on", "the", "wall"]]

Nested list

- **Same** operators as for strings
  - a+b, a*3, a[0], a[-1], a[1:], len(a)

- Item and slice assignment
  - a[0] = 98
  - a[1:2] = ["bottles", "of", “buttemilk"]
    -> [98, "bottles", "of", “buttermilk", ["on", "the", "wall"]]

A list that includes three strings

```
del a[-1]        # -> [98, "bottles", "of", “buttemilk"]
```

Last element removed

More list operations: range, append, pop, insert, reverse, sort, extend

```
>>> a = range(5)        # [0,1,2,3,4]
>>> a.append(5)         # [0,1,2,3,4,5]
>>> a.pop()             # [0,1,2,3,4]
5
>>> a.insert(0, 5.5)    # [5.5,0,1,2,3,4]
>>> a.pop(0)            # [0,1,2,3,4]
5.5
>>> a.reverse()         # [4,3,2,1,0]
>>> a.sort()            # [0,1,2,3,4]
```

>>> some_list.index(12)
1

More list operations: range, append, pop, insert, reverse, sort, extend, count, remove

```
>>> a=[1,2,3,4]
>>> a
[1, 2, 3, 4]

>>> a.extend([5,6,7])
>>> a
[1, 2, 3, 4, 5, 6, 7]

>>> a.append([5,6,7])
>>> a
[1, 2, 3, 4, [5, 6, 7]]
```

```
>>> a=[1,2,3,4,5,4,5]
>>> a.count(5)
2

>>> a.remove(4)
>>> a
[1, 2, 3, 5, 4, 5]
```

List method calls: SORT

- The sort function orders a list in place (in ascending fashion)

```
>>> aa=[4,2,6,8,1,3,4,10]
>>> aa.sort()
>>> aa
[1, 2, 3, 4, 4, 6, 8, 10]
```

Sorting in descending fashion

```
>>> aa.sort(reverse=True)
>>> aa
[10, 8, 6, 4, 4, 3, 2, 1]
```

List method calls: REVERSE and POP

- 'reverse' **reverses** the list **in-place**

```
>>> aa=[1,2,3,4]
>>> aa.reverse()
>>> aa
[4, 3, 2, 1]
```

- 'pop' deletes an item from the end

```
>>> aa.pop()
1
>>> aa
[4, 3, 2]
```

Summary of Operations in List

- append
- insert
- index
- count
- sort
- reverse
- remove
- pop
- extend

- Indexing e.g., L[i]
- Slicing e.g., L[1:5]
- Concatenation e.g., L + L
- Repetition e.g., L * 5
- Membership test e.g., ‘a’ in L
- Length e.g., len(L)

Nested List

- List in a list
- E.g.,

```
>>> s = [1,2,3]
>>> t = ['begin', s, 'end']
>>> t
['begin', [1, 2, 3], 'end']
```

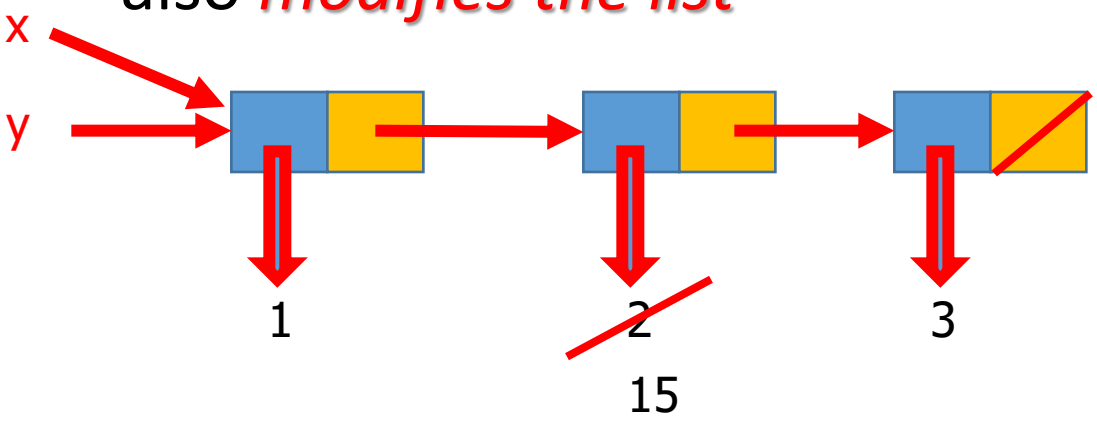
```
>>> t[1][1]
2
```

Second element from second element

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also *modifies the list*

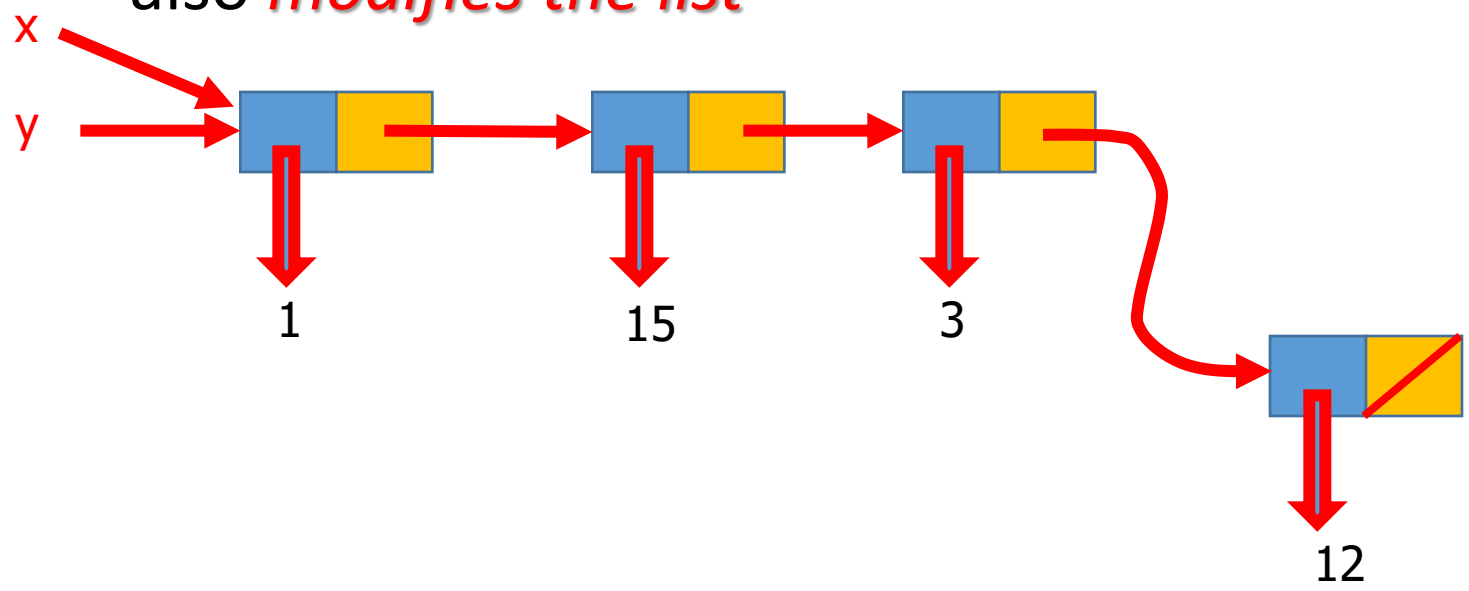
```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
```



Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also *modifies the list*

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```



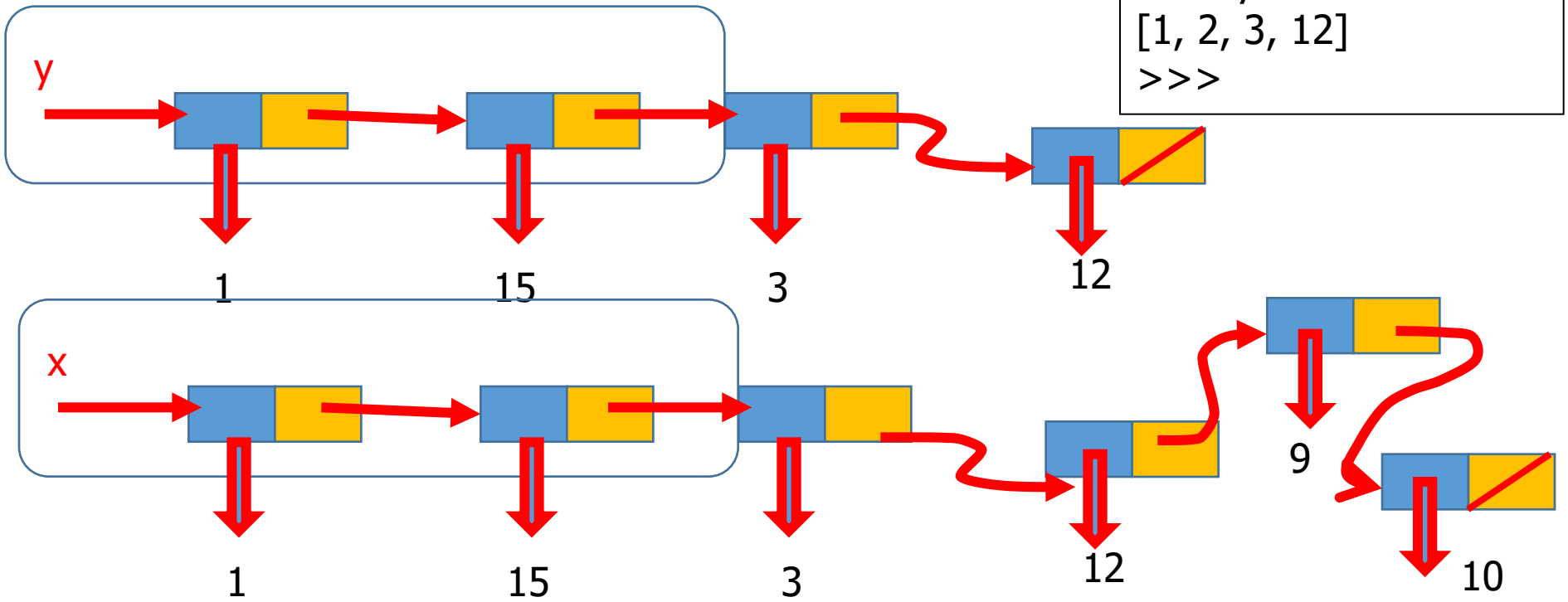
Lists: Modifying Contents

- The method **append** modifies the list *and returns* ***None***

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
```

- List addition (+) returns a new list

```
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```



List copy

```
import copy
Ori_List=[10,11,12,13,14,15]
Cpy_List=list(Ori_List)
print("Id of Original List : ",id(Ori_List))
print("Id of copied   List : ",id(Cpy_List))
print("Original List      : ",Ori_List)
print("Copied List        : ",Cpy_List)
Ori_List[1]=25
Ori_List[4]=34
print("Changed Original List      : ",Ori_List)
print("Changed Copied List        : ",Cpy_List)
```

```
Id of Original List :  1535369288576
Id of copied   List :  1535368686272
Original List      :  [10, 11, 12, 13, 14, 15]
Copied List        :  [10, 11, 12, 13, 14, 15]
Changed Original List      :  [10, 25, 12, 13, 34, 15]
Changed Copied List        :  [10, 11, 12, 13, 14, 15]
```

List copy using = operator

```
import copy
Ori_List=[10,11,12,13,14,15]
Cpy_List=Ori_List
print("Id of Original List : ",id(Ori_List))
print("Id of copied List  : ",id(Cpy_List))
print("Original List      : ",Ori_List)
print("Copied List        : ",Cpy_List)
Ori_List[1]=25
Ori_List[4]=34
print("Changed Original List      : ",Ori_List)
print("Changed Copied List        : ",Cpy_List)
```

```
Id of Original List :  2098089040832
Id of copied  List :  2098089040832
Original List          :  [10, 11, 12, 13, 14, 15]
Copied List            :  [10, 11, 12, 13, 14, 15]
Changed Original List      :  [10, 25, 12, 13, 34, 15]
Changed Copied List        :  [10, 25, 12, 13, 34, 15]
```

List copy using = operator

```
import copy
Ori_List=[[10,11,12],[20,21,22],[30,31,32]]
Cpy_List=Ori_List
print("Id of Original List  : ",id(Ori_List))
print("Id of copied   List : ",id(Cpy_List))
print("Original List      : ",Ori_List)
print("Copied List        : ",Cpy_List)
Ori_List[1][2]=25
Ori_List[2][0]=34
print("Changed Original List          : ",Ori_List)
print("Changed Copied List            : ",Cpy_List)
```

Share the same id. Hence changes made in original list are also reflected in copied list

```
Id of Original List  :  2695298962816
Id of copied List    :  2695298962816
Original List      :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Copied List        :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Changed Original List          :  [[10, 11, 12], [20, 21, 25], [34, 31, 32]]
Changed Copied List            :  [[10, 11, 12], [20, 21, 25], [34, 31, 32]]
```

List copy- Shallowcopy

```
import copy
Ori_List=[[10,11,12],[20,21,22],[30,31,32]]
Cpy_List=copy.copy(Ori_List)
print("Original List      : ",Ori_List)
print("Shallow Copied List: ",Cpy_List)
Ori_List[1][2]=25
Ori_List[2][0]=34
print("Changed Original List          : ",Ori_List)
print("Changed Shallow Copied List: ",Cpy_List)
```

```
Original List      :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Shallow Copied List:  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Changed Original List          :  [[10, 11, 12], [20, 21, 25], [34, 31, 32]]
Changed Shallow Copied List:  [[10, 11, 12], [20, 21, 25], [34, 31, 32]]
```

List copy- Shallowcopy

```
import copy
Ori_List=[[10,11,12],[20,21,22],[30,31,32]]
Cpy_List=copy.copy(Ori_List)
print("Original List id          : ",id(Ori_List))
print("Shallow copied List id : ",id(Ori_List))
print("Original List                      : ",Ori_List)
print("Shallow Copied List             : ",Cpy_List)
Ori_List.append([40,41,42])
print("Changed Original List          : ",Ori_List)
print("Changed Shallow Copied List : ",Cpy_List)
```

```
Original List id          :  1387691202368
Shallow copied List id :  1387691202368
Original List                     :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Shallow Copied List            :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Changed Original List         :  [[10, 11, 12], [20, 21, 22], [30, 31, 32], [40, 41, 42]]
Changed Shallow Copied List :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
```

List copy- deepcopy

```
import copy
Ori_List=[[10,11,12],[20,21,22],[30,31,32]]
Cpy_List=copy.deepcopy(Ori_List)
print("Id of Original List : ",id(Ori_List))
print("Id of Deep copied List : ",id(Cpy_List))
print("Original List                    : ",Ori_List)
print("Deep Copied List              : ",Cpy_List)
Ori_List[1][2]=25
Ori_List[2][0]=34
print("Changed Original List          : ",Ori_List)
print("Changed Deep Copied List: ",Cpy_List)
```

```
Id of Original List :  2825547044352
Id of Deep copied List :  2825547044608
Original List                    :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Deep Copied List              :  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
Changed Original List          :  [[10, 11, 12], [20, 21, 25], [34, 31, 32]]
Changed Deep Copied List:  [[10, 11, 12], [20, 21, 22], [30, 31, 32]]
```

Objective Questions

```
nameList = ['Harsh', 'Pratik', 'Arvind', 'Dhruv']
print (nameList[1][-1])
```

```
Lst = [1, 2, 3, 4]
Lst.append([5,6,7,8])
print (len(Lst))
```

```
list = ['a', 'b', 'c', 'd', 'e']
print (list[10:])
```

```
l=[1, 0, 2, 0, 'hello', '', []]
list(filter(bool, l))
```

Tuples

Operations on Tuples

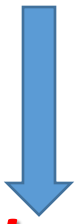
- Indexing e.g., T[i]
- Slicing e.g., T[1:5]
- Concatenation e.g., T + T
- Repetition e.g., T * 5
- Membership test e.g., ‘a’ in T
- Length e.g., len(T)

Tuples

- Like a list, tuples are iterable arrays of objects
- Tuples are **immutable** –
  once created, unchangeable
- To add or remove items, you must redeclare

Tuples

Pay attention



- Tuples are ***immutable*** **versions of lists**
- *One strange point is the format to make a tuple with one element:*

  , is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

We create a tuple with one element

Tuples

- What is a tuple?
  - A tuple is an ordered collection which cannot be modified once it has been created.
  - In other words, it's a special array, a read-only array.
- How to make a tuple? In round brackets
  - E.g.,

```
>>> t = ()
>>> t = (1, 2, 3)
>>> t = (1, )      # creating a tuple with one element is tricky
>>> t = 1,
>>> a = (1, 2, 3, 4, 5)
>>> print a[1]  # 2
```

Not the same

Creating the Tuples

- Empty Tuple

```
>>> T=()
>>> T
()
```

- Tuples can be created from List

```
>>> Li=[1,2,3,4,5]
>>> Li
[1, 2, 3, 4, 5]
>>> T=tuple(Li)
>>> T
(1, 2, 3, 4, 5)
```

- Nested tuples with mixed data types

```
>>> T= ("Pune", [8, 4, 6], (1, 2, 3))
>>> T
('Pune', [8, 4, 6], (1, 2, 3))
```

Tuples packing and unpacking

- A tuple can also be created without using parentheses. This is known as tuple packing.

```
>>> T = 1,101,"Aurangabad"
>>> T
(1, 101, 'Aurangabad')
```

- Tuple unpacking

```
>>> a,b,c=T
>>> a
1
>>> b
101
>>> c
'Aurangabad'
```

Tuples are Immutable

```
>>> t=(1,2,3,4,5)
>>> t
(1, 2, 3, 4, 5)
>>> t[2]=56
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    t[2]=56
TypeError: 'tuple' object does not support item
 assignment
```

You're not allowed to change a tuple *in place* in memory

```
>>> t = (1, 2, 3, 4, 5)
>>> t1=(10,23,45)
>>> t=t1
>>> t
(10, 23, 45)
```

Tuple with mutable elements

```
import copy
T1=(1,2,3,[],4,5)
print("Tuple T1 - ",T1)
T2=tuple(T1)                    #copy tuple using tuple(), same as T2=T1
T3=copy.deepcopy(T1)            #deep copy
T4=copy.copy(T1)                #shallow copy
T1[3].append(['abc','def'])
print("Tuple T1 - Original         ",id(T1),"  ",T1)
print("Tuple T2 - using tuple() ",id(T1),"  ",T2)
print("Tuple T3 - deep copy      ",id(T1),"  ",T3)
print("Tuple T4 - shallow          ",id(T1),"  ",T4)
```

```
Tuple T1 -  (1, 2, 3, [], 4, 5)
Tuple T1 - Original         2987484794208    (1, 2, 3, [['abc', 'def']], 4, 5)
Tuple T2 - using tuple()  2987484794208    (1, 2, 3, [['abc', 'def']], 4, 5)
Tuple T3 - deep copy       2987484794208    (1, 2, 3, [], 4, 5)
Tuple T4 - shallow          2987484794208    (1, 2, 3, [['abc', 'def']], 4, 5)
```

Tuples vs. Lists, conversions

- Lists are slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.
- We can always ***convert*** between tuples and lists using the list() and tuple() functions.

```
li = list(tu)
tu = tuple(li)
```

```
>>> a=[1,2,3,4]
>>> a
[1, 2, 3, 4]
>>> t=tuple(a)
>>> t
(1, 2, 3, 4)
```

List vs. Tuple

- What are common characteristics?
  - Both store arbitrary data objects
  - Both are of sequence data type

- What are differences?
  - Tuple **doesn't allow modification**
  - Tuple doesn't have methods
  - Tuple supports variable length parameter in function call.
  - Tuples are **slightly faster**

Tuple Methods- count, index

```
>>> T=(1,2,3,4,5,3,4,3,6)
>>> T.count(3)
3
>>> T.index(5)
4
```

Deleting a tuple

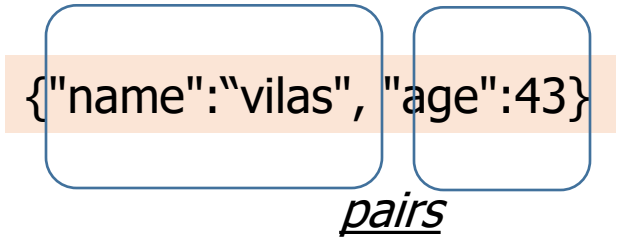
```
>>> T=(1,2,3)
>>> T
(1, 2, 3)
>>> del T
>>> T
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    T
NameError: name 'T' is not defined
```

Dictionaries

Dictionaries

- Dictionaries are sets of key & value pairs
- Allows you to identify values by a descriptive name instead of order in a list
- Keys are unordered unless explicitly sorted
- Keys are unique:
  - var[‘item’] = “apple”
  - var[‘item’] = “banana”
  - print var[‘item’] prints just banana

Dictionaries



- Dictionaries: curly brackets
  - What is dictionary?
    - Refer value through key; "associative arrays"
  - Like an array indexed by a string
  - An unordered set of *key: value* pairs
  - *Values* of any type; keys of almost any type
    - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

```
d = { "One" : 1, "Two" : 2 }
print d["Two"]    # 2
some_dict = {}
some_dict["One"] = "yow!"
print some_dict.keys() # ["One"]
```

Dictionary details

- Dictionaries are mutable
- Keys must be **immutable**:
  - numbers, strings, tuples are immutables
    - these cannot be changed after creation
  - reason is ***hashing*** (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Dictionaries

- A set of key-value pairs

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

modify

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

adds

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

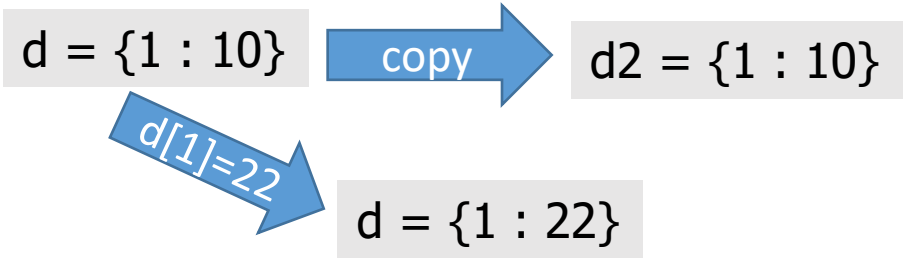
The whole pair is removed

Copying Dictionaries and Lists

- The built-in **list** function will ***copy a list***
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```



```
d = {"john":40, "peter":45}
"john" in d
```

```
d = {"john":40, "peter":45}
print(list(d.keys()))
```

```
a={1:"A",2:"B",3:"C"}
print(a.get(1,4))    A
```

```
test = {1:'A', 2:'B', 3:'C'}
test = {}
print(len(test))
```

```
a={}
a['a']=1
a['b']=[2,3,4]
print(a)
```

Sets

Creating Sets

A set is an unordered collection of items. Every set element is unique (no duplicates).Set can have only immutable elements as its member

```
>>> s={1,6,3,5}
>>> s
{1, 3, 5, 6}
```

Mix data type

```
>>> s={11,23.45,"hello",(101,102,103)}
>>> s
{11, (101, 102, 103), 'hello', 23.45}
```

Creating set from list

```
>>> L=[1,4,2,3,9]
>>> S=set(L)
>>> S
{1, 2, 3, 4, 9}
```

Creating empty Set

```
>>> s={}
>>> type(s)
<class 'dict'>

>>> s=set()
>>> type(s)
<class 'set'>
```

Updating Set

```
#updating set
>>> s={1,2,3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.update([5,6,7])
>>> s
{1, 2, 3, 4, 5, 6, 7}
```

```
#Removing an element
>>> s.discard(3)
>>> s
{1, 2, 4, 5, 6, 7}
>>> s.remove(5)
>>> s
{1, 2, 4, 6, 7}
>>> s.pop()   #Removes random element
1
>>> s
{2, 4, 6, 7}
>>> s.clear()
>>> s
set()
```

Set operations

```
#Union Operations
>>> S1={1,2,3}
>>> S2={4,5,6}
>>> S1|S2
{1, 2, 3, 4, 5, 6}
>>> S1.union(S2)
{1, 2, 3, 4, 5, 6}
>>> S2.union(S1)
{1, 2, 3, 4, 5, 6}
```

```
#Intersection Operations
>>> S1={1,2,3,4,5}
>>> S2={4,5,6,7,8}
>>> S1 & S2
{4, 5}
>>> S1.intersection(S2)
{4, 5}
>>> S2.intersection(S1)
{4, 5}
>>> S1
```

Set operations

```
#Set Difference
>>> S1={1,2,3,4,5}
>>> S2={4,5,6,7,8}
>>> S1-S2
{1, 2, 3}
>>> S1.difference(S2)
{1, 2, 3}
>>> S2-S1
{8, 6, 7}
>>> S2.difference(S1)
{8, 6, 7}
```

```
#Set symmetric  difference
>>> S1={1,2,3,4,5}
>>> S2={4,5,6,7,8}
>>> S1^S2
{1, 2, 3, 6, 7, 8}
>>> S2^S1
{1, 2, 3, 6, 7, 8}
>>>
S1.symmetric_difference(S2)
{1, 2, 3, 6, 7, 8}
```

Set operations

```
>>> S1={1,2,3}
>>> S2={4,5,6}
>>> S1.isdisjoint(S2)
True
>>> S1={1,2,3,4,5,6}
>>> S2={1,2,3}
>>> S2.issubset(S1)
True
>>>
S=set("Aurangabad")
>>> 'a' in S
True
>>> 'Z' in S
False
```

Other Operations

len()
max()
min()
sorted()
sum()

Set operations