Python Programming

by Narendra Allam

Chapter 4

Data Structures

Topics covering in this chapter

- list
    - list Operations and functions
        - Finding length of list
        - Modifying value at index
        - Adding an element at the end
        - Adding an element at a specific location
        - Deleting an element from the end
    - Itarating a list using while
    - enumerate()
    - List functions
    - Creating a Stack (LIFO) using list
    - Creating a Queue (FIFO) using list
    - Find the index of the given element
    - Reversing a list
    - Reversing list using slicing
    - Sorting a list
    - Unpacking
    - Slicing
    - List Comparisions
- tuple
    - Differences with list
        - Brackets
        - Mutability
    - Similarities with list
        - Declaration
        - Indexing
        - Scallar Multiplication
        - Itaration
        - Slicing and -ve indexing

Tuple unpacking

- List vs Tuple

- list of tuples - frequently used construct
  - iterating
  - sorting
  - largest
  - smallest
  - enumerate
  - zip and unzip
- Set
  - Introduction to set
  - How set removes duplicates?
  - Set functions
    - Searching for an element
      - 'in' operator - The fastest
    - Adding an element
      - add()
    - Removing an element
      - remove()
      - discard()
      - pop()
    - Relation between two sets
      - intersection()
      - union()
      - difference()
      - isdisjoint()
      - issubset()
      - issuperset()
    - Merging two sets
      - update()
  - Why tuples are hashable but not lists?
  - Set Use-Cases
    - Removing duplicates from a list
    - Fastest lookups
    - Intersections, Unions, Difference and set relations
- Dictionary
  - Introduction of Dictionary - Associative data structure
  - Creating a Dictionary
  - Adding elements to Dictionary
  - Deleting key value pair
  - Updating / extending a Dictionary
  - Iterating through a Dictionary
  - Tuple unpacking method
  - Converting list of tuples into Dictionary
  - Converting Dictionary to List of tuples
  - Lambda introduction
  - Sorting List of tuples and dictionaries

   - Finding max(), min() in a dict
   - Wherever you go, dictionary follows you!
   - Dictionary Use-Cases
     - Counting Problem
     - Grouping Problem
     - Always Latest
     - Caching
- Counter()
   - simplest counting algorithm
- DefaultDict
   - Always has a value
- OrderedDict
   - Maintains order
- Dequeue
   - Short time memory loss
- Heapq
   - efficient in-memory min-heap()
   - heapify()
   - nlargest()
   - nsmallest()
   - heappush()
   - heappop()
- ForzenSet
   - Hashable set
   - Use-Cases
     - Set of sets
     - Set as Key in Dict
- Packing and Unpacking
   - Swapping two values
   - List packing and Unpacking
   - Tuple packing and Unpacking
   - String packing and Unpacking
   - Set packing and Unpacking
- Iterating containers using iter() and next()

Introduction

Data structure is a particular way of organizing data in memory, so that it can be searched, retrieved, stored and processed efficiently. Any data structure is designed to organize data to suit a specific purpose. General data structure types include the list, the tree, the graph and so on. Python has its own set of efficiently implemented butil-in data structures.

List

List is a collection of elements(python objects). Purpose of list is, to group up the things, which falls under same category. e.g,

List of grocery items,
List of employee ids,
list of book names etc.

As group of similar elements stored in a list, mostly those are homogenous(of same data type).

Creating a list in python is putting different comma-separated values, between square brackets.

Eventhough list principle suggests homogeneous data items in it, it is not mandatory and still allowed to have different types.

For example −

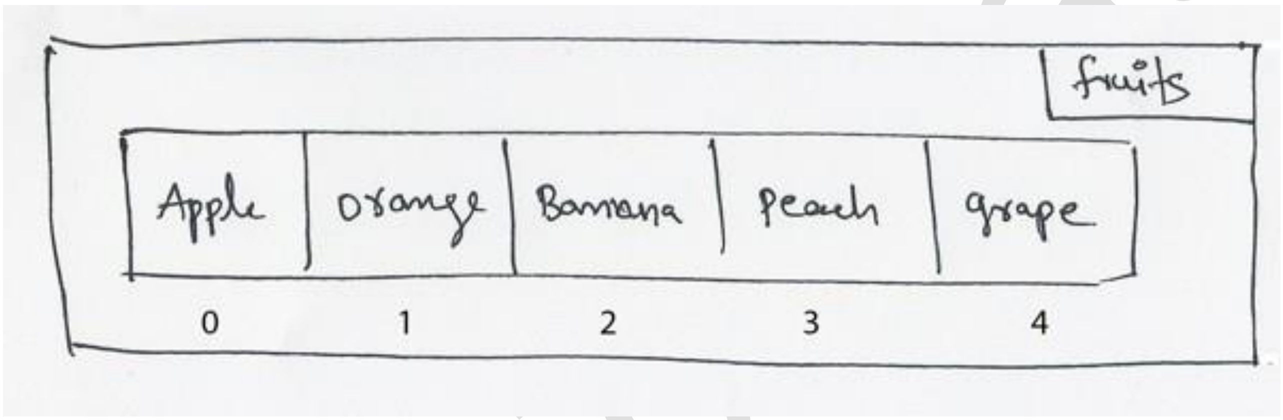
```
l1 = [30, 32, 31, 35, 30, 36, 34]
l2 = [1234, 'John', 230000.05, True]
l3 = []
l4 = [99]
l5 = list()
l6 = list([4, 5, 6])
```

- List is mutable.
- **IQ: Python list is implemented using dynamically resizable array(vector in C++,Java etc.).
- List uses indexing to access values.
- Search operation on an unsorted list is O(n) operation.
- **IQ: lists are un-hashable
- type of list is 'list'

Each element in a list can be accessed using square brackets enclosing its positional value called indexing.
In the below list fruits,

```
fruits = ["Apple", "Orange", "Banana", "Peach", "grape"]
```

fruits[0] refers "Apple",
fruits[1] refers "Orange",
fruits[2] refers "Banana",
and so on..



```
In [ ]: fruits = ["Apple", "Orange", "Banana", "Peach", "grape"]
        print fruits[0]
```

```
In [ ]: print fruits[1]
```

```
In [ ]: print fruits[5]
```

As starting index is 0, Last item index is 4, not 5, so we get IndexError.

list Operations and Functions

Finding length of a list:

```
In [ ]: l = [3, 4, 5, 8, 2, 1]
        print len(l)
```

modifying value at index i:

```
In [ ]: i = 3
        l = [6, 4, 5, 8, 2, 1]
        l[i] = 99
        print l
```

Adding an element at the end:

```
In [ ]: l = [6, 4, 5, 8, 2, 1]
        l.append(99)
        print l
```

Adding an element at a specific location:

insert(index, value): takes index and value

```
In [ ]: l = [6, 4, 5, 8, 2, 1]
        l.insert(3, 99)
        print l
```

Deleting an element from the end:

pop() removes the elements from the end by default, and returns

```
In [ ]: l = [6, 4, 5, 8, 2, 1]
        rem = l.pop()
        print 'Element removed is:', rem
        print l
```

Deleting an element from a specific location:

pop() also takes an index, removes the element and returns. Throws error if index is invalid.

```
In [ ]: l = [6, 4, 5, 8, 2, 1]
        rem = l.pop(3)
        print 'Element removed is:', rem
        print l
```

Find and delete an element with specified value:

remove() doesn't return a value. It simply removes the first occurance of the value. Throws ValueError if element not found.

```
In [ ]: l = [6, 4, 5, 8, 2, 1, 8, 7]
        l.remove(8)
        print l
```

```
In [ ]: l = [6, 4, 5, 8, 2, 1, 8, 7]
        l.remove(99)
        print l
```

Iterating a list using while:

```
In [ ]: i = 0
        l = [6, 4, 5, 8, 2]
        while i < len(l):
            print l[i]
            i += 1
```

Iterating a list using for: Pythonic Way!

```
In[] l = [6, 4, 5, 8, 2]
     for x in l:
         print x
```

```
6
4
5
8
2
```

Program: Find the biggest element in a list.

```
In[] l = [6, 4, 5, 8, 2]
     biggest = l[0]

     for x in l:
         if biggest < x:
             biggest = x

     print biggest
```

```
8
```

Program: Square each element in the list and print.

```
In[] l = [6, 4, 5, 8, 2]
     for x in l:
         print x*x
```

```
36
16
25
64
4
```

Program: Square each element in the list and save it back to its location.

```
In[] l = [6, 4, 5, 8, 2]
     for x in l:
         x = x*x
     print l
```

```
[6, 4, 5, 8, 2]
```

Original list cannot be changed as x is just a copy of each element in that iteration.

Solution.1:

```
In [ ]: i = 0
        l = [6, 4, 5, 8, 2]
        while i < len(l):
            l[i] = l[i]*l[i]
            i += 1
        print l
```

enumerate(): enumerate function adds a sequence number starts from zero, to each item in the sequence, packs as a tuple and returns in each iteration. In each iteration enumerate() retruns tuple([seq_num, cur_item]). This is very useful when we want to track the indices while iterating sequence.

```
In[] fruits = ["Apple", "Orange", "Grape", "Banana", "Peach"]

     for idx, fruit in enumerate(fruits):
         print idx, fruit
```

```
0 Apple
1 Orange
2 Grape
```

```
3 Banana
4 Peach
```

We can also have a custom start value for sequence as below,

```
In[]  fruits = ["Apple", "Orange", "Grape", "Banana", "Peach"]

      for idx, fruit in enumerate(fruits, start=1):
          print idx, fruit
```

```
1 Apple
2 Orange
3 Grape
4 Banana
5 Peach
```

Solution.2: Using for loop

```
In[]  l = [6, 4, 5, 8, 2]
      for i, x in enumerate(l):
          l[i] = x*x
      print l
```

```
[36, 16, 25, 64, 4]
```

Multiplying list with a scalar:

```
In[]  l = [3, 4, 6]
      print l * 3
```

```
[3, 4, 6, 3, 4, 6, 3, 4, 6]
```

Concatenating two lists:

```
In[]  l1 = [3, 4, 6]
      l2 = [7, 8, 9]

      print l1 + l2
```

```
[3, 4, 6, 7, 8, 9]
```

List functions:

Searching for an element: the 'in' operator

```
In[] l = [3, 4, 5, 6, 1, 9, 10, 8]
     x = 7
     print x in l
```

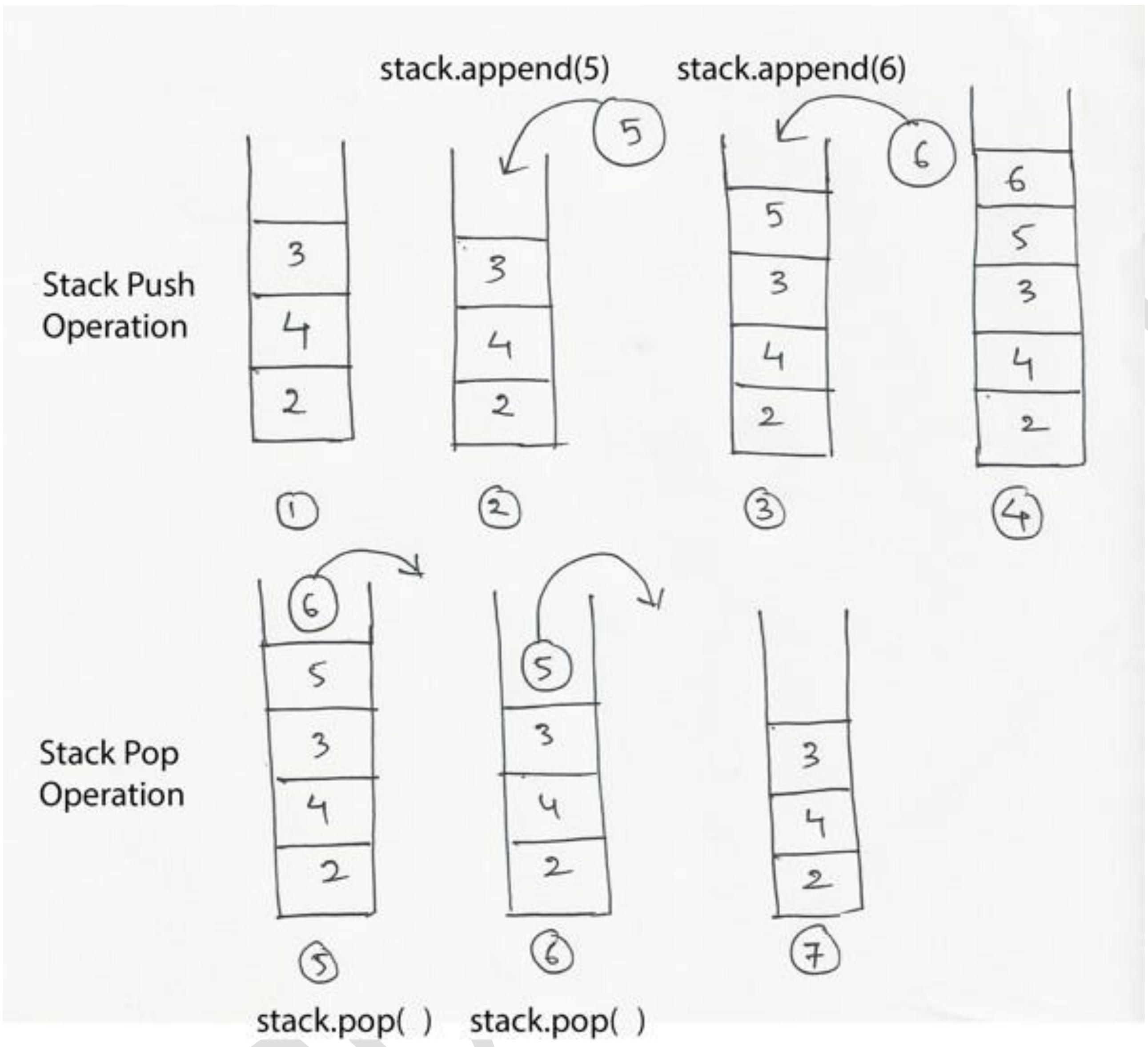
```
False
```

Creating a Stack (LIFO) using list:

Stack is a data structure in which, insertion and deletion operations follow the pattern, Last-In-First-Out. A list, in which, insertion and deletion operations are restricted to one end (front or rear) is called as Stack. We can achieve this using l.append() and l.pop(). Generally insertion is called 'push' operation and deletion is called 'pop' operation.

```
In[] stack = [2, 4, 3]
     print stack
     stack.append(5)
     print stack
     stack.append(6)
     print stack
     stack.pop()
     print  stack
     stack.pop()
     print stack
```

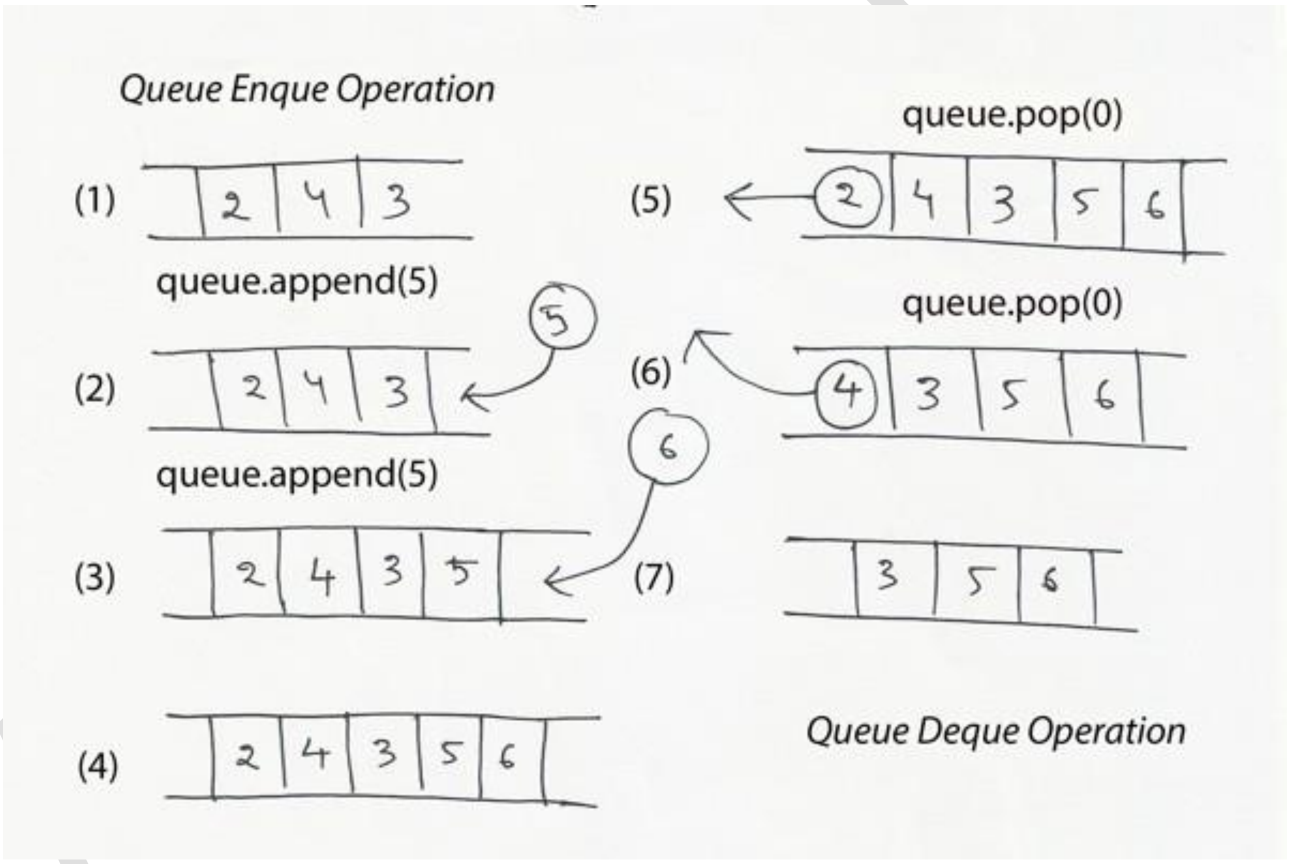
```
[2, 4, 3]
[2, 4, 3, 5]
[2, 4, 3, 5, 6]
[2, 4, 3, 5]
[2, 4, 3]
```



Creating a Queue (FIFO) using list: Queue is a data structure in which, insertion and deletion operations follow the pattern, First-In-First-Out. A list, in which, insertion and deletion operations are restricted to seperate ends (generally delete front and insert rear) is called as Queue. We can achieve this using l.append() and l.pop(0). Generally insertion is called 'enque' operation and deletion is called 'deque' operation.

```
In[]  queue = list([2, 4, 3])
      print queue
      queue.append(5)
      print queue
      queue.append(6)
      print queue
      queue.pop(0)
      print queue
      queue.pop(0)
      print queue
```

```
[2, 4, 3]
[2, 4, 3, 5]
[2, 4, 3, 5, 6]
[4, 3, 5, 6]
[3, 5, 6]
```



Extending a list with other:

```
In[]  l = [3, 4, 5]
      s = [99, 55, 88]
      l.extend(s)
      print l
```

```
[3, 4, 5, 99, 55, 88]
```

Instead of extend, if we use append(), list s, becomes an individual element in the list l.

```
In[]  l = [3, 4, 5]
      s = [99, 55, 88]
      l.append(s)
      print l

      [3, 4, 5, [99, 55, 88]]
```

now type(l[3]) is a list instead an int

```
In[]  type(l[3])
Output: list
```

Find the index of the given element

If element found, index() function returns the index of first occurance, else 'ValueError'

```
In[]  l = [6, 7, 9, 5, 2]
      print l.index(5)

      3
```

Reversing a list:

reverse() function changes the list in-place.

```
In[]  l = [3, 4, 5, 2, 1]
      l.reverse()
      print l

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

Reversing list using slicing

This doesn't change original list, afterall, it is just a view of the original.

```
In[]  l = [3, 4, 5, 2, 1]
      print l[::-1]
      print l

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

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

Sorting a list

****Note:** Python uses 'Tim Sort' algorithm, which is one of the stable sorting algorithms. It is a combination of 'merge sort' and 'insertion sort'.

```
In[]  l = [6, 7, 9, 5, 1, 2, 5, 4, 3]
      l.sort()
      print l
```

```
[1, 2, 3, 4, 5, 5, 6, 7, 9]
```

Sorting in decreasing order:

```
In[]  l = [6, 7, 9, 5, 1, 2, 5, 4, 3]
      l.sort(reverse=True)
      print l
```

```
[9, 7, 6, 5, 5, 4, 3, 2, 1]
```

Unpacking:

Unpacking is the process of extracting values from a sequence and assigning them to correponding variables on the other side.

```
In[]  l = [2, 3, 4]
      x, y, z = l
      print "x:{} y:{} z:{}".format(x, y, z)
```

```
x:2 y:3 z:4
```

Slicing:

```
In[]  l = [2, 4, 3, 1, 7, 9, 8, 0, 6, 5]
      print l[:5]
```

```
[2, 4, 3, 1, 7]
```

```
In[]  l = [2, 4, 3, 1, 7, 9, 8, 0, 6, 5]
      print l[7:]
```

```
[0, 6, 5]
```

```
In[]  print l[-3:]
```

```
[0, 6, 5]
```

List Comparisions:

'==' operator: == operator checks the equality of each element in both lists.

```
In[]  l1 = [1, 2, 4, 7, 8, 9]
      l2 = [1, 2, 4, 7, 8, 9]
      l3 = [7, 8, 9, 1, 2, 4]
```

```
In[]  l1 == l2
```

```
Output:  True
```

```
In[]  l2 == l3
```

```
Output:  False
```

```
In[]  mid = len(l1)//2
      l1[:mid] == l3[mid:]
```

```
Output: True
```

cmp():

cmp() function returns 0 if both are equal else returns -1

```
In[]  cmp(l1, l2)
```

```
Output: 0
```

```
In[]  cmp(l1, l3)
```

```
Output: -1
```

```
In[]  cmp(l1[:mid], l3[mid:])
```

```
Output:  0
```

Note: is operator doesn't work on lists, lists with same content have different ids(addresses).

```
In[]  l1 = [1, 2, 4, 7, 8, 9]
      l2 = [1, 2, 4, 7, 8, 9]
      l1 is l2
```

```
Output: False
```

Tuple

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values enclosed in paranthesis. Some times paranthesis is optional.

For example −

```
tup1 = (1234, 'John wesley', 240000.0, True)
tup2 = (1, 2, 3, 4, 5)
tup3 = 1, 3, 2, 4
tup4 = ()
tup5 = (3,)
```

- Tuple is immutable.
- Tuple values can be of multiple types.
- Tuple internally uses array of constant references.
- tuple uses indexing to access value like list.
- Search operation is always O(n).
- Tuples are hashable.
- type of tuple is 'tuple'

Apart from immutability, tuples mostly behave like a list.

Differences with List

Brackets:

Tuples uses paranthesis in declaration

```
In [ ]: l = [3, 5, 4, 2, 1]
        t = (3, 5, 4, 2, 1)
```

Mutability:

Elements cannot be modified after initilization.

```
In[]  l = [3, 5, 4, 2, 1]
      l[3] = 99
      print l
```

```
[3, 5, 4, 99, 1]
```

```
In[]  t = (3, 5, 4, 2, 1)
      t[3] = 99
      print t
```

```
-----------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-34-35f2bcb689dc> in <module>()
      1 t = (3, 5, 4, 2, 1)
----> 2 t[3] = 99 # is NOT OK
      3 print t

TypeError: 'tuple' object does not support item assignment
```

When having single element: We put a comma at the end, when there is one element in the tuple, Why?

This is required, to differentiate with an expression.

```
x = (9)
y = (9,)
```

x is an integer and y is a tuple

```
In[]  x = (9)
      y = (9,)

      print x*3
      print y*3
```

```
27
(9, 9, 9)
```

Similarites with List

Declaration:

```
In[] l = [3, 5, 4, 2, 1]
     t = (3, 5, 4, 2, 1)

     print type(l), type(t)
```

```
<type 'list'> <type 'tuple'>
```

Indexing:

```
In[] l = [3, 5, 4, 2, 1]
     t = (3, 5, 4, 2, 1)
     print l[3], t[3]
```

```
2 2
```

Scalar Multiplication:

```
In[] print [1, 4, 2] * 3
     print (1, 4, 2) * 3
```

```
[1, 4, 2, 1, 4, 2, 1, 4, 2]
(1, 4, 2, 1, 4, 2, 1, 4, 2)
```

Iteration:

```
In[]  l = [3, 5, 4, 2, 1]
      print 'list Iteration:'
      for x in l:
          print x

      t = (3, 5, 4, 2, 1)
      print 'tuple Iteration:'
      for x in t:
          print x
```

```
list Iteration:
3
5
4
2
1
tuple Iteration:
3
5
4
2
1
```

Slicing and -ve Indexing:

```
In[]  t = (1234, 'John', 25000, True)
      l = [8, 2, 5, 4, 9, 1, 3, 7, 10, 6]

      print "-------"
      print "Slicing"
      print "-------"

      print t[2:7:2]
      print l[2:7:2]

      print "------------"
      print "-Ve Indexing"
      print "------------"

      print t[::-1]
      print l[::-1]
```

```
-------
Slicing
-------
(25000,)
[5, 9, 3]
------------
-Ve Indexing
------------
(True, 25000, 'John', 1234)
[6, 10, 7, 3, 1, 9, 4, 5, 2, 8]
```

Tuple unpacking

Unpacking:

```
In[]  t = 3, 4, 5
      x, y, z = t

      print "x:{}, y:{}, z:{}".format(x, y, z)
```

```
x:3, y:4, z:5
```

Initilizing values at a time:

This is possible because in python comma seperated values are treated as a tuple.

```
In [ ]:  x, y, z = 7, 8, 9
         print "x:{}, y:{}, z:{}".format(x, y, z)
```

Swapping two values in python:

```
In[]  x = 20
      y = 30

      x, y = y, x

      print "x:{}, y:{}".format(x, y)
```

```
x:30, y:20
```

**** Iterating list of tuples:**

```
In[]  lt = [('Apple', 30), ('Grape', 20), ('Mango', 25)]

      for tpl in lt:
          print tpl[0], tpl[1]
```

```
Apple 30
Grape 20
Mango 25
```

We can use list un packing method to write clean code, as below

```
In[]  fruit_bucket = [('Apple', 30), ('Grape', 20), ('Mango',  25)]

      for fruit, count in fruit_bucket:
          print fruit, count
```

```
Apple 30
Grape 20
Mango 25
```

In each iteration one tuple will be unpacked to 'fruit' and 'count' variables.

****Difference between List and Tuple**

List	Tuple
mutable	immutable
dynamically resizable array	fixed in size
* emphasizes on quantity	* emphasizes on the structure
** unhashable	** hashable
use square brackets	use paranthesis (optional some times)
comma not required when having single element	comma is required when having single element

Built-in functions on sequences

Finding length of the sequence

len():

```
In[] l = [7, 8, 9, 3, 2]
     t = (7, 8, 9, 3, 2)
     s = "NEWYORK"

     print len(l), len(t), len(s)
```

```
5 5 7
```

sorting the sequence

sorted():

List has its own sort() function. sort() function sorts elements in-place. But tuple and str are immutable types, we cannot sort them in-place. We need an external function, and we have one. sorted() function takes a sequence and returns a sorted list of items.

```
In[] l = [7, 8, 9, 3, 2]
     t = (7, 8, 9, 3, 2)
     s = "NEWYORK"

     print sorted(l)
     print sorted(t)
     print sorted(s)
```

```
[2, 3, 7, 8, 9]
[2, 3, 7, 8, 9]
```

```
['E', 'K', 'N', 'O', 'R', 'W', 'Y']
```

Finding maximum:

max():

```
In[]  l = [7, 8, 9, 3, 2]
      t = (7, 8, 9, 3, 2)
      s = "NEWYORK"

      print max(l)
      print max(t)
      print max(s)
```

```
9
9
Y
```

Finding minimum:

min():

```
In[]  l = [7, 8, 9, 3, 2]
      t = (7, 8, 9, 3, 2)
      s = "NEWYORK"

      print min(l)
      print min(t)
      print min(s)
```

```
2
2
E
```

Sum of the numbers

sum():

```
In[]  l = [7, 8, 9, 3, 2]
      t = (7, 8, 9, 3, 2)
      print sum(l)
      print sum(t)
```

```
29
29
```

More built-in functions in python

abs() :- returns absolute value

```
In[]  print abs(-13), abs(13)

      13 13
```

chr() :- takes ASCII code and returns character

```
In[]  print chr(65), chr(97)

      A a
```

ord() :- takes character and returns ASCII code

```
In[]  print ord('A'), ord('a')

      65 97
```

List of tuples - Frequently used construct

In non-object-oriented environments, list of tuples is generally used to represent a list of database records. Let's take an example of list of employee records. We have employee id, name, salary and age in each row in the same order. Below construct is widely used represenation of list of employee records. To represent a row we are using tuple here.

```
In[]  employees = [
                  (1237, 'John', 23000, 25),
                  (1235, 'Samantha', 40000, 41),
                  (1238, 'Amanda', 45000, 30),
                  (1239, 'Alex', 57000, 31),
                  (1236, 'Vicky', 40000, 24)
                  ]
```

How do you sort above list of tuples, on their salaries?

```
In[]  employees = [
                  (1239, 'John', 23000, 25),
                  (1235, 'Samantha', 13000, 21),
                  (1238, 'Amanda', 45000, 30),
                  (1237, 'Alex', 57000,  31),
                  (1236, 'Vicky', 40000,  24)
                  ]

      sorted_records = sorted(employees)
      for rec in sorted_records:
          print rec
```

```
(1235, 'Samantha', 13000, 21)
(1236, 'Vicky', 40000, 24)
(1237, 'Alex', 57000, 31)
(1238, 'Amanda', 45000, 30)
(1239, 'John', 23000, 25)
```

By default sorted() method takes first value of each tuple as the comparsion criteria. To change this behaviour we have to pass the comparision criteria explicitely using a callable object (function ,lambda function etc.)

Introduction to lambda: lambda function is an one line function. Which expands the expression given. syntax:

```
    lambda parameters: expression
```

```
In[]  f = lambda x, y: x + y
      print f(4, 5)
```

```
9
```

in the above code, **f(4, 5)** replaced by **4 + 5**, thus resulting **9**

sorted(), max() and min() functions have a second parameter which is **key**. *key* is a lambda function, which is internally used by above three functions when two tuples are being compared(< or >). Comparing two tuples directly with less than or greater than operators is meaning less. So, key function recieves each tuple and returns first item in the tuple. A typical key lambda function looks like below.

```
In[]  key = lambda x: x[0]
```

Lets apply thsi key on two tuples,

```
In[]  key = lambda x: x[0]

      t1 = (1235, 'Samantha', 53000, 21)
      t2 = (1236, 'Vicky', 40000, 24)

      print key(t1) < key(t2)
```

```
True
```

in the above code **key(t1) < key(t2)** is replaced with t1[0] < t2[0]. What we should understand is first item of the tuple(index o) is being compared not the tuple itself. So, result is True.

How do we change key lambda to consider salary as the comparision criteria? simple, define key as below.

```
key = lambda x: x[2]
```

x[2] means, taking 3rd item in the list as comparision criteria.

```
In[]  key = lambda x: x[2]

      t1 = (1235, 'Samantha', 53000, 21)
      t2 = (1236, 'Vicky', 40000, 24)

      print key(t1) < key(t2)
```

```
False
```

in the above code **key(t1) < key(t2)** is replaced with t1[2] < t2[2], thus resulting True. Now it is time to apply a lambda to *sorted()* function

Sorting list of tuples on salary:

```
In[] employees = [
                 (1239, 'John', 23000, 25),
                 (1235, 'Samantha', 13000, 21),
                 (1238, 'Amanda', 45000, 30),
                 (1237, 'Alex', 57000,  31),
                 (1236, 'Vicky', 40000,  24)
                 ]

     sorted_records = sorted(employees, key=lambda x:x[2],  reverse=True)
     for rec in sorted_records:
         print rec
```

```
(1237, 'Alex', 57000, 31)
(1238, 'Amanda', 45000, 30)
(1236, 'Vicky', 40000, 24)
(1239, 'John', 23000, 25)
(1235, 'Samantha', 13000, 21)
```

Employees with max salary:

```
In[] employees = [
                 (1239, 'John', 23000, 25),
                 (1235, 'Samantha', 13000, 21),
                 (1238, 'Amanda', 45000, 30),
                 (1237, 'Alex', 57000,  31),
                 (1236, 'Vicky', 40000,  24)
                 ]

     print 'Max sal:', max(employees, key=lambda  x:x[2])
```

```
Max sal: (1237, 'Alex', 57000, 31)
```

Employee with min age:

```
In[] employees = [
                 (1239, 'John', 23000, 25),
                 (1235, 'Samantha', 13000, 21),
                 (1238, 'Amanda', 45000, 30),
                 (1237, 'Alex', 57000,  31),
                 (1236, 'Vicky', 40000,  24)
                 ]

     print 'Min age:', min(employees, key=lambda  x:x[3])
```

```
Min age: (1235, 'Samantha', 13000, 21)
```

Set

A set contains an unordered collection of unique objects. The set data type is, as the name implies, a mathematical set. Set does not allow duplicates. Set does not maintain an order. This is because, the placement of each value in the set is decided by arbitary index prodcuded by hash() function. So, We should not relie on the order of set elements, even though some times it looks like ordered.

Internally uses a hash table. Values are translated to indices of the hash table using hash() function .
When a collison occures in the hash table, it ignores the element.
This explains, why sets unlike lists and tuples can't have multiple occurrences of the same element.
type() of set is 'set'.

Set Operations

Creating a set:

```
In[]  s = {2, 3, 1, 2, 1, 3}
      print s
```

```
set([1, 2, 3])
```

Creating an empty set:

```
In[]  s = set()
      print s
```

```
set([])
```

The below syntax is not an empty set, it is empty dictionary, which we will be discussing later.

```
In[]  s = {}
      print type(s)
```

```
<type 'dict'>
```

Converting a list to set:

```
In[] l = [2, 6, 3, 2, 6, 3, 2, 4, 1, 3]
     s = set(l)
     print s

     set([1, 2, 3, 4, 6])
```

Set doesn't allow duplicates:

```
In[] s = {2, 6, 3, 2 ,6, 3, 2, 4, 1, 3}
     print s

     set([1, 2, 3, 4, 6])
```

```
In[] s = {"Apple", "Orange", "Banana", "Orange", "Apple",  "Banana"}
     print s

     set(['Orange', 'Apple', 'Banana'])
```

```
In[] s = {2.3, 4.5, 3.2, 2.3, 5.3}
     print s

     set([4.5, 2.3, 5.3, 3.2])
```

Adding an element to a set():

```
In[] s = {2, 5}
     s.add(3)
     print s

     set([2, 3, 5])
```

Removing an element from set():

Using remove() function:

```
In[] s = {3, 4, 5}
     x = 4
     s.remove(x)
     print s

     set([3, 5])
```

If element not present, throws a 'KeyError'

```
In[]  s = {3, 4, 5}
      x = 99
      s.remove(x)
      print s
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-56-473c74b9d9c9> in <module>()
      1 s = {3, 4, 5}
      2 x = 99
----> 3 s.remove(x)
      4 print s

KeyError: 99
```

Using discard() function:

Removes x from set s if present. If element not existing, doesn't throw any error, it just keeps quite.

```
In[]  s = {3, 4, 5}
      x = 99
      s.discard(x)
      print s
```

```
set([3, 4, 5])
```

Using pop() function:

pop() removes and return an arbitrary element from s; raises 'KeyError' if empty

```
In[]  s = {3, 4, 5}
      print s
      s.pop()
      print s
```

```
set([3, 4, 5])
set([4, 5])
```

Updating a set:

```
In[] s1 = {4, 5, 2, 1}
     s2 = {7, 8, 5, 6}
     s1.update(s2)
     print s1
```

```
set([1, 2, 4, 5, 6, 7, 8])
```

update() funcion adds all the elements in s2 to s1.

Iterating through a set:

```
In[] s = {'Apple', 'Orange', 'Peach', 'Banana'}
     for x in s:
         print x
```

```
Orange
Apple
Peach
Banana
```

Set unpacking:

```
In[] s = {'Apple', 'Ball', 'Cat'}
     print s
     x, y, z = s
     print x, y, z
```

```
set(['Ball', 'Apple', 'Cat'])
Ball Apple Cat
```

Use-Cases

1. Set removes duplicates

Set uses hash-table data structure internally. Hashing is the process of translating values into array indices. Placement of each value in the set is decided by an arbitary index prodcuded by hash() function. hash() function always ensures producing same index for a given value. Still there is a chance that, two values may get same index. This is called hash() collision. Hash-table generally stores all the values with the same hash code, in the same bucket. Before storing in same bucket, to make sure not to have any duplicates in the bucket, it compares current element with each existing element in the bucket. If an element with same value is existintg in the bucket, it ignores current element. Thus removing duplicates. </p>

```
In[] s = {35, 92, 51, 35, 42, 92}
```

```
In[] print s
```

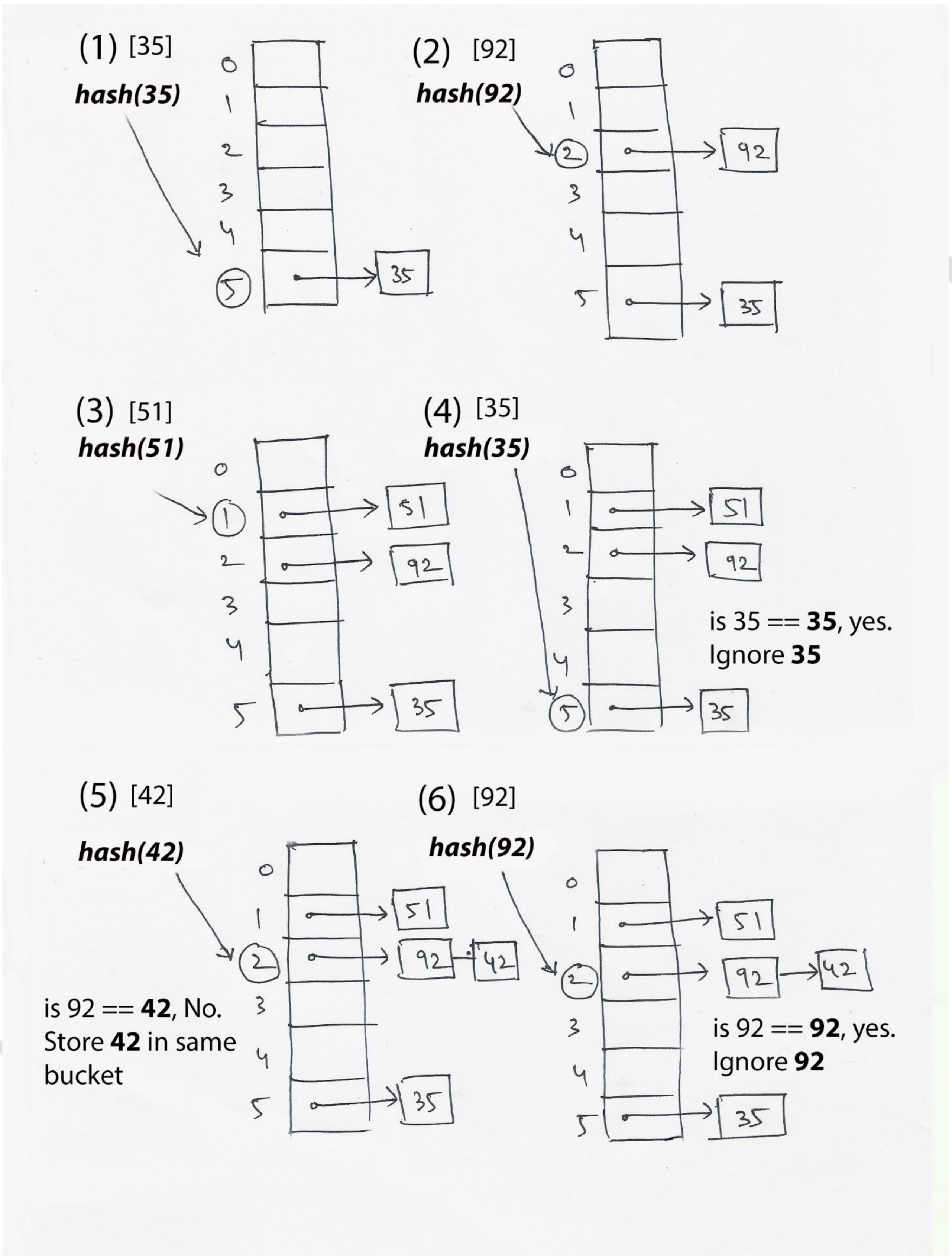
```
set([51, 42, 35, 92])
```

Python has a built-in function **hash()** which returns an unque identifier for each value we pass. This hash code is unique for every value in the lifetime of a program. As the implementation of the butil-in hash() function is complex to understand now. To make it simple, assume that, when we pass 'n' to **hash()** function,i.e, calling **hash(n)**, returns n%10.

For example, calling **hash(35)**, results 35%10, which is 5.

Hashing is the process of translating values to unqiue numbers, generally called as hash code. These numbers are utilized by other data structures like sets and dictionaries to allocate a slot(bucket) in an array.

Let's see how set removes duplicates from a list of values.


(1) [35]
hash(35)
0 1 2 3 4 5
35
(2) [92]
hash(92)
0 1 2 3 4 5
92
35
(3) [51]
hash(51)
0 1 2 3 4 5
51
92
35
(4) [35]
hash(35)
0 1 2 3 4 5
51
92
35
is 35 == 35, yes.
Ignore 35
(5) [42]
hash(42)
0 1 2 3 4 5
51
92
42
35
is 92 == 42, No.
Store 42 in same bucket
(6) [92]
hash(92)
0 1 2 3 4 5
51
92
42
35
is 92 == 92, yes.
Ignore 92

2. Faster Look-ups, O(1):

We know that Set stores elements in a hash table. Searching(look-up) operation is always constant and mostly just involves one operation. As we know that set is unordered, due to arbitary value of hash code. We cannot access the individual elements, as there is no fixed index, we can only check element existance.

```
In[]  s = {35, 67, 92, 42, 77}
      k = 42
      print k in s

      True
```

3. Relations between sets

Union of two sets: All the unqiue elements in both the sets.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      all_values = s1.union(s2)
      print all_values

      set([3, 4, 5, 6, 8, 9])
```

Above union() function is equivalent of applying '|' operator.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      all_values = s1 | s2
      print all_values

      set([3, 4, 5, 6, 8, 9])
```

Intersection of two sets: Common elements in both the sets.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      common = s1.intersection(s2)
      print common

      set([5, 6])
```

Above intersection() function is equivalent of applying '&' operator.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      common = s1 & s2
      print common

      set([5, 6])
```

Difference of two sets: Elements which are present in one set but not in the other.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      diff = s1.difference(s2)
      print diff

      set([3, 4])
```

Above intersection() function is equivalent of applying '&' operator.

```
In[]  s1 = {3, 4, 5, 6}
      s2 = {5, 9, 6, 8}
      diff = s1 - s2
      print diff

      set([3, 4])
```

Program:
Given customer ids who deposited the money, for the last three days.

1. Find the customer ids who deposited on 1st and 3rd days but not on the 2nd day.
2. Find the customer id, who deposited all the days
3. Customer ids, who did deposites atleast 2 of the 3 days
4. Total number of customers who did deposites

day1 = {1122, 1234, 1256, 1389}
day2 = {1134, 1256, 1399, 1455}
day3 = {1256, 1455, 1122, 1899}

Solution:

```
In[] day1 = {1122, 1234, 1256, 1389}
     day2 = {1134, 1256, 1399, 1455}
     day3 = {1256, 1455, 1122, 1899}

     print 'Customer who deposited on day 3 and day 1 but not on day  2:'
     , (day1 & day3) - day2
```

```
Customer who deposited on day 3 and day 1 but not on day 2: set([1
122])
```

```
In[] print 'Customers who did deposites all the threee days:', day1 & da
     y2 & day3
```

```
Customers who did deposites all the threee days:  set([1256])
```

```
In[] customers = (day1 & day2) | (day2 & day3) | (day3 &  day1)
     print 'Customers who did deposotes atleast 2 days out of 3  days'
```

```
Customers who did deposotes atleast 2 days out of 3  days
```

```
In[] all_cust = day1 | day2 | day3
     print 'Number of customers who did deposites:',  len(all_cust)
```

```
Number of customers who did deposites: 8
```

Some more functions on sets

```
In[] s1 = {3, 4, 5, 6}
     s2 = {5, 6, 4}
     s3 = {8, 7, 9}
```

```
In[] s1.isdisjoint(s3)
Output: True
```

```
In[] s2.issubset(s1)
Output: True
```

```
In[] s1.issuperset(s2)
Output: True
```

Why tuple is hashable, but not list?

List is dynamically resizable array, and elements can be changed, deleted and added at any time. On sequences like list and tuple, hash() is computed on individual elements, then it is combined generally using xor operator to resolve the index. This hash code is unique for it's life time. Dynamic containers like list, varies in size and elements gets modifed. As elements are varying, evaluating a constant hash() is impossible. Tuples are immutable computing a constant hash() is possible.

```
In[]  s = {(1, 2), (3, 4), (1, 2), (4, 3), (2, 1)}
      print s
```

```
set([(1, 2), (3, 4), (2, 1), (4, 3)])
```

```
In[]  l = [1, 2]
      s = {[3, 4], l, [2, 1], [1,2], [4, 3]}
      print s
```

```
-----------------------------------------------------------------
---------
TypeError                              Traceback (most recent c
all last)
<ipython-input-80-dd928c97c932> in <module>()
      1 l = [1, 2]
----> 2 s = {[3, 4], l, [2, 1], [1,2], [4, 3]} # lists are un-hash
able
      3 print s

TypeError: unhashable type: 'list'
```

Note: The main reason to have tuple in python is, hashability. List is hashable when it is immutable. A constant list is tuple. Similarly, a set() is not hashable, we cannot store, a set in another set, as it is mutable. So our only option is to have a constant set. Actucally we have one; ForzenSet() from 'collections' module. Hashability is very important property in programming.
"All mutable types are unhashable"
We discuss more on this in the next sections.

Dictionary

In python list() is sequence of elements. Each element in list() can be accesed using its index(position). Let's take a list,

```
l = [34, 32.5, 33, 35, 32, 35.1, 33.6]
```

Suppose the above list is representing maximum temperatures of the last week, from Sunday to saturday.
How do you access max temperature on Thursday.

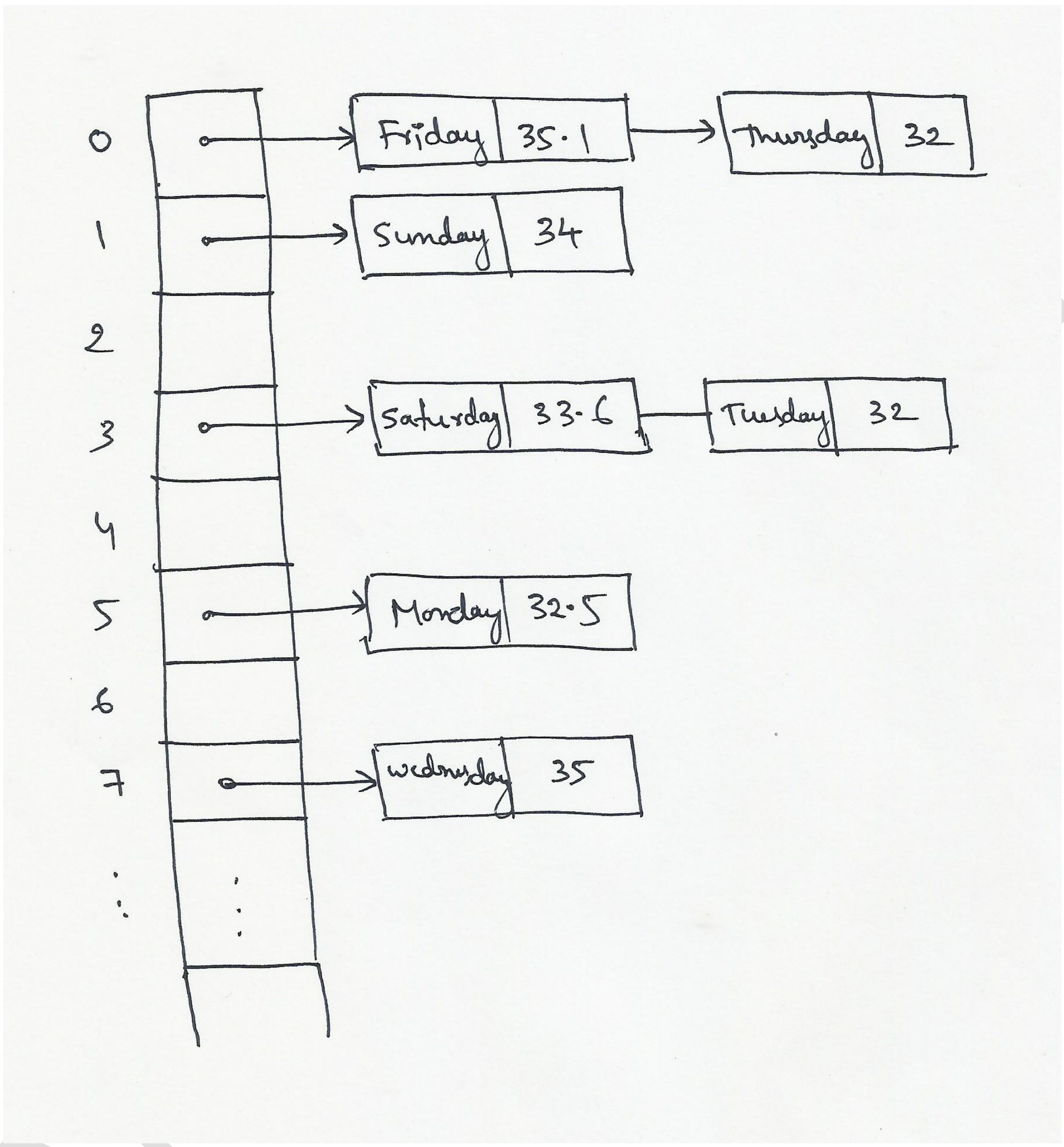
As Index 0 represents max temperature on Sunday and
1 represents Monday,
2 represents Tuesday,
and so on,

```
    print l[4]
```

Gives max temperature on Thursday. Associating temperatures and indices(numeric) in this way, gives unrealistic perspective on the problem. If there is a way to access each temperature in the list with meaningful indices, like, l['Sunday'], l['Monday'] etc; This makes associations more lively, problem-solving more realistic. This is where dictionary can really help us.

Dictionary is an associative container, which has a set of Key-Value pairs. 'Key' is the 'Index', through which we access associated value.

```
    d = {'Sunday':34, 'Monday':32.5, 'Tuesday':33, 'Wednesday':35, 'Thursda
    y':32, 'Friday':35.1, 'Saturday':33.6}
```



in the above dictionary, element which is on the left side of colon(':') is the **Key** also referred as **Index** and right side element is the **Value**

Dictionary internally uses hash-table data structure.
type() of dictionary is 'dict'.

Note: Like set, dictionary also an unordered data strucure.

Creating an empty dictionary:

```
In[]  d = {}
      print d
```

```
{}
```

```
In[]  d = dict()
      print d
```

```
{}
```

Creating dict and initilizing with key-value pairs:

```
In[]  d = {1:'One', 2: 'Two', 3:'Three'}
      print d

      d = {'Hyderabad': 500001, 'Chennai': 400001, 'Delhi':  100001}
      print d
```

```
{1: 'One', 2: 'Two', 3: 'Three'}
{'Delhi': 100001, 'Hyderabad': 500001, 'Chennai': 400001}
```

Imp Note: Key can be any hashable type, where as for value there is no data-type restriction.

Retreiving value from dictionary:

Syntax: d[Key] To retrieve a value, **hash(Key)** is called and an index is prodcued, where the the key-value should be found. If a bucket has multiple key-value pair, equality check happens on each key, and assocaited value is returned, else throws a 'KeyError'.

```
In[]  d = {'Mango': 30, 'Banana': 15, 'Peach': 20}

      print d['Peach']
```

```
20
```

Adding key-value pair to a dictionary:

```
Syntax: d[Key] = Value
```

To store a key-value pair, **hash(Key)** is called and an index is prodcued. If same key not existing, key-value pair is stored there, else old value is replaced with new value.

```
In[]  d = {'Mango': 30, 'Banana': 15, 'Peach': 20}
      d['Orange'] = 40
      print d
```

```
{'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach': 20}
```

In the above dictionary d, if key is already existing, value is replaced.

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      d['Orange'] = 100
      print d
```

```
{'Orange': 100, 'Mango': 30, 'Banana': 15, 'Peach': 20}
```

Accessing a key, which doesn't exist:

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      print d['Grape']
```

```
-----------------------------------------------------------------
---------
KeyError                                  Traceback (most recent c
all last)
<ipython-input-87-1b9c11bc78b2> in <module>()
      1 d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
----> 2 print d['Grape']

KeyError: 'Grape'
```

Above program throws 'KeyError' when key deosn't exist. Some times, this behaviour is not accepted, instead program should continue by assuming a default value.

Using get() function:

d.get(Key) : If key doesn't exist, get() returns None, instead of throwing KeyError.

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      print d.get('Orange')
```

```
40
```

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      print d.get('Grape')
```

```
None
```

We can also specify, default value to return, if key deosn't exist.

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      print d.get('Grape', 10)
```

```
10
```

In the above example, if key 'Grape' exists, get() returns associated value else, default value which is 10

Checking Key existance in a dictionary

Using 'in' operator

```
In[]  d = {'Apple': 20, 'Orange': 15, 'Peach': 10}
      key = 'Peach'
      print key in d
```

```
True
```

Using hash_key() function

```
In[]  d = {'Apple': 20, 'Orange': 15, 'Peach': 0}
      print d.has_key('Peach')
```

```
True
```

Do not use get() function to check key's existance

Note: This is an amateaur coding practice, which leads to catastrophic system failures some times .

```
In[]  d = {'Apple': 20, 'Orange': 15, 'Peach': 0}

      if d.get('Peach'):
          print 'Key exists'
      else:
          print "Key doesn't exist"
```

```
Key doesn't exist
```

In the above example 'Peach' is existing but returns 0, which coerced(implicit type conversion) to False, and produce output, "Key doesn't exist". We are supposed to check key's existance here. To do so, we should not depend on the value returned by get() function.

Imp Note: To check key's existance in a dictionary, We should either use 'in' operator or has_key() function but, using get() function is not suggested.

Removing a key-value pair

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach': 20}
      key = 'Banana'

      ret = d.pop(key)

      print 'Returned value:', ret
      print 'dict after removing the key:', d
```

```
Returned value: 15
dict after removing the key: {'Orange': 40, 'Mango': 30, 'Peach':
20}
```

pop() function removes the key and its associated value, ('Banana' and 15) and returns 15. This throws 'KeyError' when key deosn't exist.

Iterating through a dictionary

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}

      for x in d:
          print x
```

```
Orange
Mango
Banana
Peach
```

By default dictionary provides an iterator to list of keys to for loop. That is the reason, we are seeing only keys in the above example. However we can access value, if we have a key.

```
In[]  d = {'Orange': 40, 'Mango': 30, 'Banana': 15, 'Peach':  20}
      for x in d:
          print x, d[x]
```

```
Orange 40
Mango 30
Banana 15
Peach 20
```

Some dict functions:

d.keys(): Returns list of all keys

```
In[]  d.keys()
```

```
Output: ['Orange', 'Mango', 'Banana', 'Peach']
```

d.values(): Returns list of values

```
In[]  d.values()
```

```
Output: [40, 30, 15, 20]
```

d.items(): Returns key value pairs as a list of tuples.

```
In[]  d.items()
```

```
Output: [('Orange', 40), ('Mango', 30), ('Banana', 15), ('Peach', 20)]
```

We have seen how to iterate through a list of tuples in the previous sections.

```
In[]  for fruit, quantity in d.items():
          print fruit, quantity
```

```
Orange 40
Mango 30
Banana 15
Peach 20
```

Converting a list of tuples to a dict:

In[]
```
lt = [('Apple', 30), ('Orange', 20), ('Peach', 40)]
d = dict(lt)
print d
```

```
{'Orange': 20, 'Apple': 30, 'Peach': 40}
```

Converting list of lists to a dict:

In[]
```
ll = [['Apple', 30], ['Orage', 20], ['Peach', 40]]
d = dict(ll)
print d
```

```
{'Orage': 20, 'Apple': 30, 'Peach': 40}
```

Note:

Python understands developers intenetion. list of lists or list of tuple, when inner sequence has two elements, dict() converts it into a dcitionary

Updating/extending a dictionary

In[]
```
d1 = {'Hyd': 1234, 'Mum': 1235}
d2 = {'Blr': 1236, 'Delhi': 1237, 'Hyd':1999}
d1.update(d2)
print d1
```

```
{'Mum': 1235, 'Delhi': 1237, 'Hyd': 1999, 'Blr': 1236}
```

Value can be of any type

type constraints in Keys and Values:

Keys must be hashable types.
E.g int, str, float, bool, complex, tuple, frozenset, user defined objects etc,
For values, there is no restriction on type.

Note: set is not hashable. Below is an example dict for Student (or student group) ids and courses registered.

```
In[] d = {
         1234        : ['C++', 'Java'],
         (1299, 1289): ('Python', 'SQL'),
         1288        : {'C#', 'Python'},
         (1266, 1277): 'Pyhon'
         }

     print d
```

```
{1288: set(['C#', 'Python']), 1234: ['C++', 'Java'], (1299, 1289):
('Python', 'SQL'), (1266, 1277): 'Pyhon'}
```

Use-Cases:

1. SQL databases primary key and indexing

It is mandatory to have a primary key in any SQL database table. The reason is , we can easily search for entire record by using primary key. The secrect here is again the hash table. Which is called index for the table. In a typical scenorio of banking customer-care, a customer generally calls the Customer-Care and enquires about a particular transaction. He gives the transaction id. In banking system millions of transactions can happen in a day or a week. But retrieving one record among them by performing linear search is time taking process. Banking systme takes advantage of the hash-table(dictionary in python) and builds index based on the transaction id for quickest response from the system. As part of database tuning, to improve performance, some times, apart from primary key, these indexes also built on other columns(composite keys, secondary keys). Below is an example index implemenation of customer transaction table in SQL databases.

```
In[] txn_table = {
          'TXN1234': ('TXN1234', 'CUSTID123564', 23000, 'WITHDRAWAL', '1
     2/08/2015:11:32:21'),
          'TXN1235': ('TXN1235', 'CUSTID123897', 34000, 'CASHDEPOSIT', '
     08/02/2016:14:51:02'),
          'TXN1266': ('TXN1266', 'CUSTID122938', 16000, 'CHEQUECLR',  '21
     /11/2015:09:13:53')
          }
```

Querying details for transaction 'TXN1266':

```
In[] print 'Txn details:', txn_table['TXN1266']
```

```
Txn details: ('TXN1266', 'CUSTID122938', 16000, 'CHEQUECLR', '21/1
1/2015:09:13:53')
```

2. Word Counting Problem

PROGRAM: Find the occurance of each word in the given list.

```
words = ["Apple", "Orange", "Apple", "Banana", "Peach",
         "Banana", "Apple", "Peach", "Apple", "Banana"]
```

Without dictionaries:</l>

```
for      in
    print
```

With dictionaries:</l>

```
for      in
    if      not in

    else


print
```

```
for      in

print
```

Sorting a dictionary:</l>

```

```

```

```

```
                                        lambda
```

Finding the word with maximum frequency:

```
                                    lambda
```

*Using **enumerate()** to get the indices of a sequence/iterable*

```




for               in
    print
```

3. Grouping Program:

Program: *List out all the indices of each word.*

```






for               in

    if        not in



    else


for                   in
    print
```

4. Caching

This is the 4th use-case for dict, which will be discussed along with recursion topic in functions Chapter.

5. Keep the latest

Dictionaries can also be use to maintain the latest data at any instance

Program: *Latest balances of the all customers by now.*

```
for                 in

print
```

Collections

Word counting problem can be easily solved by simply using builtin data structure ***Counter*** *from 'collections' module.*

```
from collections  import



print
```

most_common():

Counter has most_common() function, which lists the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
print
```

OrderedDict

OrderedDict retains the order in which key-value pairs are added to the dict. Same order is maintained while iterating the dict.

```
from collections import

print









for       in
    print

print  \n









for       in
    print
```

In the above example built-in dict stores (e, E) before (d, D)
But in OrderedDict it stores in the same order given.

Deque

Dequeue is a list like data struture which supports append operation, but only retains last 'maxlen' number of elements. This data structre belongs to collections module.

Use-Case: *When we want to keep track of last n elements, use deque*

```
from collections import


for   in

print

print
```

Defaultdict

```
from collections import
```

defaultdict() returns a dict. When a key is encountered for the first time, it is not already in the dict; so an entry is automatically created using the key and value(0 value of the type). if int is the type provided to defaultdcit, it assigns '0' as the value for the key first time.

int - 0
float - 0.0
bool - False
list - []
set - set()

```
from collections import
```

```
          in
```

In the above statement, key 'Apple'is not there, so adds 'Apple' as key and '0' as value.

In the above statment, default dict adds key 'Orange' and retruns 0and the adds 1 and stores it back to dict 'd'

defaultdict adds an amepty list and returns, to which we are appending a '0' in the above statement.

Word Counting Program revisited:

Find the count of each word in the given list.

```
from collections  import




for       in


print
```

Index Grouping Program revisited:

Find the indices of each word in the given list

```
from collections  import







for              in



```

Exercise Program:~~*~~*</br> We are expecting some packets from 5 different ip addresses. We are asked to collect them and remove duplicates and sort them in increasing order.*

```
from collections  import











for             in


for
```