Data Structures

Introduction

A data structure is a construct to store data and organize that data efficiently to perform computations. In Python, data structures can be created using the following types:

- Lists
- Tuples
- Sets
- Dictionaries

Lists

List is an ordered collection of objects. Lists are mutable. Individual elements in a list are accessed through offset (index). Starting index is zero. Lists are variable-length, heterogeneous, and can be nested. Lists are maintained as arrays inside Python interpreter. Not as linked lists. Following are various examples of list literals and functions that can be applied on a list:

Operation	**Description**
L = [ ]	Empty list
L = [1,3.8,'hai',[1,2]]	A list with four items. Last item is nested list.
L=list('Python')	List of iterable items
L=list(range(10))	List of successive integers
L[i]	Item at index i
L[i][j]	Index of index
L[i:j]	Slice from i to j-1
len(L)	Length of list L
L1 + L2	Concatenate
L*4	Repeat
8 in L	Membership
L.append(10)	Appending element at end
L.extend([3, 7, 1])	Extending the list L
L.insert(i, X)	Insert element X at index i
L.index(X)	Find the index of element X
L.count(X)	Count the occurrences of X in the list
L.sort()	Sort the list

L.reverse()	Reverse the list
L.copy()	Copying the list
L.clear()	Empty the list
L.pop(i)	Remove and print an item at index i. If index i is not given, it removes and returns the last element.
L.remove(X)	Remove an item X from the list
del L[i]	Remove an item at index i in list L
del L[i:j]	Remove all elements from index i to j-1 in list L
L[i:j] = [ ]	Remove all elements from index i to j-1 in list L
L[i] = 10	Index assignment
L[i:j] = [2,5,8]	Slice assignment
L = [x**2 for x in range(5)]	List comprehension
list(map(ord, 'spam'))	List maps

Tuples

Tuple is an ordered collection of items. Tuples are immutable. Tuples are fixed in length, heterogeneous, and can be nested. Individual elements can be accessed through offset (index). Following are various examples of tuple literals and functions that can be applied on a tuple:

Operation	**Description**
( )	Empty tuple
T = (1,)	Tuple with one-item
T = (1,4,6,2)	Tuple with four items
T = 1,4,6,2	Same as above. Creates a tuple with four items
T = (2, 6, (4,8))	Nested tuples
T = tuple('Python')	Tuple of items in an iterable
T[i]	Access tuple item at index i
T[i][j]	Access tuple at row index i and column index j
T[i:j]	Access a slice of elements in the tuple
len(T)	Length of tuple
T1 + T2	Concatenate two tuples
T*3	Repeat tuple elements 3 times
'hi' in T	Finding membership of 'hi' in the tuple
T.index('Py')	Returns the index of substring. Otherwise error.

T.Count(‘Py’)	Returns the no. of occurences of a item in tuple
namedtuple(‘Emp’, [‘name’, ‘jobs’])	Named tuple extension type

The main difference between a list and tuple is, a list is not write protected; i.e., we can add or remove elements from a list. But, a tuple is write protected. Once a tuple is created it is not possible to add or remove elements from it.

Sets

Set is an unordered collection of unique and immutable objects, i.e., a set cannot contain lists or byte arrays etc. Set itself is mutable. Sets are useful for performing operations corresponding to mathematical set theory. Following are examples of different operations that can be performed on sets:

Consider two sets s1 and se:

```
>>> s1 = {'a','b','c','d'}
>>> s2 = {'c','d','e'}
```

Set difference operation can be performed on sets s1 and s2 as follows:

```
>>> s1-s2
{'a', 'b'}
```

Union operation can be performed on sets s1 and s2 as follows:

```
>>> s1|s2
{'a', 'b', 'c', 'e', 'd'}
```

Intersection operation can be performed on sets s1 and s2 as follows:

```
>>> s1&s2
{'d', 'c'}
```

Symmetric difference can be performed on sets s1 and s2 as follows:

```
>>> s1^s2
{'a', 'b', 'e'}
```

Using the *intersection* function, we can find the common elements between two sets as follows:

```
>>>s1.intersection(s2)
{'d', 'c'}
```

We can add an element to the set using *add* function as follows:

```
>>>s2.add('f')
{'f','c','d','e'}
```

An important property of set is, it doesn't allow duplicate elements. Whenever there is a requirement of storing unique values, we can always use a set.

Dictionaries

Dictionary is an unordered collection of objects. Objects in a dictionary are accessed through keys instead of by position (like in a list). Dictionaries are mutable, heterogeneous, and nestable. Each key can have only one object associated to it. Internally dictionaries are implemented as hash tables. Following are various examples of dictionary literals and functions that can be applied on dictionaries:

Operation	**Description**
D = { }	Empty dictionary
D = {‘id’:101, ‘name’:‘Ramesh’}	Dictionary with two items
D = {‘name’:{‘fname’:’Ramesh’, ‘lname’:’Kumar’}}	Nested dictionaries
D = dict(id=101, name=‘Ramesh’)	Alternative way for creating a dictionary
D[‘name’]	Indexing by key
D[‘name’][‘lname’]	Accessing element in nested dictionary
‘name’ in D	Membership test for key
D.keys()	Returns all the keys in dictionary
D.values()	Returns all values in dictionary
D.items()	Returns all key-value tuples
D.copy()	Copies a dictionary
D.clear()	Empties a dictionary
D.update(D2)	Merge by keys
D.popitem()	Remove or returns any (key, value) pair
len(D)	Returns number of items
D[key] = value	Adding/changing keys
del D[key]	Deleting entries by key
D={x:x+2 for x in range(10)}	Dictionary comprehension

A dictionary is unique when compared to other types in Python. In a dictionary each element is made of two things: a *key* and a *value*. A dictionary is represented using braces.

Sequences

A sequence is a positionally ordered collection of objects. Sequences maintain a left-to-right order. Items (objects) are fetched based on their relative position from the left end. Sequences support operations like indexing and slicing. Examples of sequences in Python are: strings, lists, and tuples.

Comprehensions

Python comprehensions are constructs that create sequences from existing sequences in a clear and concise manner. Comprehensions are of three types:

- list comprehensions
- set comprehensions
- dict comprehensions

List comprehensions were introduced in Python 2.0; while set and dict comprehensions have been introduced in Python 2.7.

List Comprehensions

List comprehension is the most popular Python comprehension. It allows us to create a new list of elements that satisfy a condition from an iterable. An iterable is any Python construct that can be looped over like lists, strings, tuples, sets. In list comprehensions we use square brackets.

General syntax of a list comprehension is as follows:

```
[expression for item1 in iterable1 if cond1
        for item2 in iterable2 if cond2
        ...
        for itemN in iterableN if condN]
```

Following is an example of list comprehension for creating a list of squares of numbers in range 1-10:

```
s=[x*x for x in range(1,11)]
```

In the previous example, we can also use if condition to generate only squares of even numbers:

```
s=[x*x for x in range(1,11) if x%2==0]
```

Result of above comprehension will be:

```
[4, 16, 36, 64, 100]
```

List comprehensions can be useful to perform matrix computations. Consider following matrix:

```
m=[[1,1,1],[2,2,2],[3,3,3]]
```

We can print the diagonal elements as follows:

```
[m[i][i] for i in range(0,3)] which prints [1 2 3]
```

We can calculate sum of elements in each row in a matrix as follows:

```
[sum(row) for row in m] which prints [3 6 9]
```

Let our matrix be:

```
m=[[1,2,3],[4,5,6],[7,8,9]]
```

We can perform transpose of above matrix by writing:

[[row[i] for row in m] for i in range(0,3)] which gives:

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

Set Comprehensions

Set comprehensions were added to python in version 2.7. In set comprehensions, we use the braces rather than square brackets. For example, to create the set of the squares of all numbers between 0 and 10 the following set comprehension can be used:

```
>>> x = {i**2 for i in range(10)}
>>> x
set([0, 1, 4, 81, 64, 9, 16, 49, 25, 36])
```

Dict Comprehensions

Just like set comprehensions, dict comprehensions were added to python in version 2.7. Below we create a mapping of a number to its square using dict comprehension:

```
>>> x = {i:i**2 for i in range(10)}
>>> x
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```