Introduction to Data Structure and Algorithm in Python

LAB MANUAL 3

Activities for this lab:

- **Explain the concepts of array (List, Array, Tuple and Dictionary).**
- ***Array, tuple and dictionary creation and implementation***
- ***Lab exercise***

Array

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

In Python, the closest object to **an array is a list**

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5];

list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list **indices start at 0**, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7];


print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result –

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example –

```
list = ['physics', 'chemistry', 1997, 2000];


print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Note: append() method is discussed in subsequent section.

When the above code is executed, it produces the following result –

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];


print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

When the above code is executed, it produces following result –

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Note: remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

	[:]	Slicing, Extract a part of a sequence

Indexing, Slicing, and Matrices

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods:

Python includes the following list functions –

SN	Function with Description
1	**cmp(list1, list2)** Compares elements of both lists.
2	**len(list)** Gives the total length of the list.

3	**max(list)** Returns item from the list with max value.
4	**min(list)** Returns item from the list with min value.
5	**list(seq)** Converts a tuple into list.

Python includes following list methods

SN	Methods with Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list

	List.split()
	
	`>>> s= 'this is my string'`
	`>>> s`
	`'this is my string'`
	`>>> s.split()`
	`['this', 'is', 'my', 'string']`

	List.join()
3	**list.extend(seq)** Appends the contents of seq to list

4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

Sample code on remove, del and pop

```
>>> a=[1,2,3]
>>> a.remove(2)
>>> a
[1, 3]
>>> a=[1,2,3]
>>> del a[1]
>>> a
[1, 3]
>>> a= [1,2,3]
>>> a.pop(1)
2
>>> a
[1, 3]
>>>
```

Use `del` to remove an element by index, `pop()` to remove it by index if you need the returned value, and `remove()` to delete an element by value. The

latter requires searching the list, and raises `ValueError` if no such value occurs in the list.

Sample code on repetition

Try the sample codes

```
1. myList = [1,2,3,4]
2. A = [myList]*3
3. print(A)
4. myList[2]=45
5. print(A)
```

Sample code on extend

```
1. # language list
2. language = ['French', 'English', 'Malay', 'Chinese']

3. # another list of language in Nigeria
4. Nigerialanguage1 = ['Hausa', 'Yoruba']

5. language.extend(Nigerialanguage1)

6. # Extended List
7. print('Language List: ', language)
```

Sample code on python methods

Try the sample codes

```
1. myList = [624, 6, 'ECE3104', 19.5,7]
2. myList.append('Algorithms')
3. print(myList)
4. myList.insert(2,4.5)
5. print(myList)
6. print(myList.pop())
7. print(myList)
8. print(myList.pop(1))
9. print(myList)
10.     myList.pop(2)
11.     print(myList)
12.     myList.sort()
13.     print(myList)
14.     myList.reverse()
15.     print(myList)
16.     print(myList.count(6.5))
17.     print(myList.index(4.5))
```

```
18.     myList.remove(6.5)
19.     print(myList)
20.     del myList[0]
21.     print(myList)
```

Sample code on range

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

List comprehensions: When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print squares   # Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print squares   # Prints [0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print even_squares  # Prints "[0, 4, 16]"
```

TUPLES

Tuples are very similar to lists in that they are heterogeneous sequences of data. The difference is that a tuple is immutable, like a string. A tuple cannot be changed. Tuples are written as comma-delimited values enclosed in parentheses
The main difference between lists and tuples are – Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

```
myTuple = (5,'ECE3104',5.96)
>>> myTuple
(5, ECE3104, 5.96)
>>> len(myTuple)
3
>>> myTuple[0]
5
>>> myTuple * 3
(5,'ECE3104',5.96, 5,'ECE3104',5.96, 5,'ECE3104',5.96)
>>> myTuple[0:2]
(5, ECE3104)
>>>
```

However, if you try to change an item in a tuple, you will get an error. Note that the error message provides location and reason for the problem.

```
>>> myTuple[1]='ECE3105'
```

Using List to Create Matrix

```
>>> x=[[2,3,4,5],[2,4,6,1]]
>>> x
[[2, 3, 4, 5], [2, 4, 6, 1]]
>>> x[0][1]
3
>>> x[1][3]
1
>>> x[2][1]
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    x[2][1]
IndexError: list index out of range
Its error because accessing more than x
```

Dictionary

A dictionary maps a set of objects (keys) to another set of objects (values). A Python dictionary is a mapping of unique keys to values.

Creating a dictionary is as simple as placing items inside curly braces {} and [] square brackets to index it. Separate the key and value with colons : and with commas , between each pair

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Two ways to create dictionary in Python

1. use curly braces {}

2. use the keyword `dict` with bracket ()

```
# empty dictionary example 1
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'Jackfruit', 2: 'Mango'}

# dictionary with mixed keys
my_dict = {'name': 'Aida', 1: [2, 4, 3]}

# using dict() keyword example 2
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Access elements from a dictionary

dictionary uses keys. Key can be used either inside square brackets or with the `get()` method.

The difference while using `get()` is that it returns `None` instead of `KeyError`, if the key is not found.

Try the code segment either in script of shell

```
1. my_dict = {'name':'Arif', 'age': 21}

1. print(my_dict['name'])

2. print(my_dict.get('age'))

3. # Trying to access keys which doesn't exist throws error

4. # Get syntax 1.
5.
6. Print (my_dict.get('address'))
7. Print (my_dict.get('address', "Address not found"))
8.
9. # Get syntax 2.
10.     Print (my_dict['address'])
```

Output?

Add or change a value to the dictionary

```
2. my_dict = {'name':'Arif', 'age': 21}

3. # update value
4. my_dict['age'] = 22

5. print(my_dict)

6. # add item
7. my_dict['address'] = ' Gombak '
8. print(my_dict)
```

Output?

Remove a key and it's value

You can remove key-value pairs with the `del` operator

```
1. # create a dictionary
2. squares = {1:1, 2:4, 3:9, 4:16, 5:25}

3. # remove a particular item
```

```
4. print(squares.pop(4))
5. print(squares)

6. # remove an arbitrary item
7. print(squares.popitem())

8. print(squares)

9. # delete a particular item
10.     del squares[5]

11.     print(squares)

12.     # remove all items
13.     squares.clear()

14.     print(squares)

15.     # delete the dictionary itself
16.     del squares

17.     # Throws Error
18.     # print(squares)
```

Output?

Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}

print(squares)
```

output ?

For-loop To iterate items in a dictionary can be directly enumerated with a for-loop.

```
plants ={"spinach": 2,"squash":
3, "carrot": 6}

#    Loop    over    dictionary
directly.

# ... This only accesses keys.

for plant in plants:

    print(plant)
```

output ?

In-keyword. A dictionary may (or may not) contain a specific key. Often we need to test for existence. One way to do so is with the in-keyword.In

True: This keyword returns 1 (meaning true) if the key exists as part of a key-value tuple in the dictionary.

False: If the key does not exist, the in-keyword returns 0, indicating false. This is helpful in if-statements.

```
1. animals = {}
2. animals["orang utan"] = 1
3. animals["lion"] = 2
4. animals["giraffe"] = 4

5. # Use in.
6. if "lion" in animals:
7. print("Has lion")
8. else:
9. print("No lion")

10. # Use in on nonexistent
   key.
11. if "elephant" in
   animals:
12. print("Has elephant")
13. else:
14. print("No elephant")
```

Output ?

Keys, values. A dictionary contains keys. It contains values. And with the keys() and values() methods, we can store these elements in lists. Example

```
hits = {"home": 125,
"sitemap": 27, "about": 43}
keys = hits.keys()
values = hits.values()

print("Keys:")
print(keys)
print(len(keys))

print("Values:")
print(values)
print(len(values))
```

Output ?

Try other dictionary methods, this is related to Hash tables in our next lab

Python Dictionary Methods	
Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Return the value of `key`. If `key` doesnot exit, return `d`(defaults to `None`).

items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if keyis not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert keywith a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

Importing modules

Allow use of other python files and libraries with imports: import math -
Named imports: import math as m –
Specific imports: from math import pow –
Import all: from math import *

```
So we see the following pattern for lists:

Operation           Big-O Efficiency
index []              O(1)
index assignment      O(1)
append              O(1)
pop()               O(1)
pop(i)              O(n)
insert(i,item)      O(n)
del operator        O(n)
iteration             O(n)
contains (in)       O(n)
get slice [x:y]     O(k)
del slice             O(n)
set slice             O(n+k)
reverse             O(n)
concatenate           O(k)
sort                O(n log n)
multiply              O(nk)
"""
```

remove	O(n)