UNIT IV

COMPOUND DATA

**LISTS, TUPLES, DICTIONARIES**

Lists, list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples, tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension, Illustrative programs: selection sort, insertion sort, merge sort, quick sort.

Objective:

To use Python data structures –- lists, tuples, dictionaries

Outcome:

Represent compound data using Python  lists, tuples, dictionaries

Table of Contents

[4.1 Lists 3](#_Toc485107229)

[4.1.1 List operations 3](#_Toc485107230)

[4.1.2 List Slices 5](#_Toc485107231)

[4.1.3 List Methods 6](#_Toc485107232)

[4.1.4 List Loop 8](#_Toc485107233)

[4.1.5 Mutability 9](#_Toc485107234)

[4.1.6 Aliasing 11](#_Toc485107235)

[4.1.7 Cloning Lists 11](#_Toc485107236)

[4.1.8 List parameters 13](#_Toc485107237)

[4.2 Tuples 14](#_Toc485107238)

[4.2.1 Tuple Assignment 14](#_Toc485107239)

[4.2.2 Tuple as return value 15](#_Toc485107240)

[4.3 Dictionaries 15](#_Toc485107241)

[4.3.1 Operations and methods 15](#_Toc485107242)

[4.4 Advanced List Processing 17](#_Toc485107243)

[4.4.1 List Comprehension 17](#_Toc485107244)

[4.5 Illustrative Programs 18](#_Toc485107245)

[4.5.1 Selection sort 18](#_Toc485107246)

[4.5.2 Insertion sort 21](#_Toc485107247)

[4.5.3 Merge sort 23](#_Toc485107248)

[4.5.4 Quick Sort 26](#_Toc485107249)

[Syllabus 29](#_Toc485107250)

# 4.1 Lists

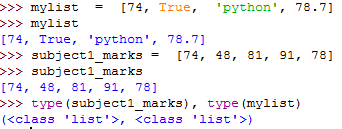
List is the collection (bag) of objects. We extensively use list to store and manipulate data in everyday computing.

Examples:

1. List of web pages matching the keyword (google)
2. List of friends (facebook)
3. List of products prices (amazon)
4. List of tasks to do
5. List of grocery items to be purchased
6. List of students enrolled in a class

The objects in the list can be of same type or of different types.

Example:



Lists may be constructed in several ways:

* Using a pair of square brackets to denote the empty list: []
* Using square brackets, separating items with commas: [a], [a, b, c]
* Using a list comprehension: [x for x in iterable]
* Using the type constructor: list() or list(iterable)

## 4.1.1 List operations

|  |  |
| --- | --- |
| **List Operations** | **Example** |
| \*  **repeat** |  |
| +  **concatenate** |  |
| []  **Empty list** |  |
| **List[index]** |  |

Exercises:

1. What is the output?

>>> a = 10

>>> mylist = [a]\*5

>>> mylist[3]

1. What is the output?

>>> mylist1 = ['In', 'python']

>>> mylist2 = ['explicit','is','better']

>>> mylist = mylist1 + mylist2

>>> mylist += ['than','implicit']

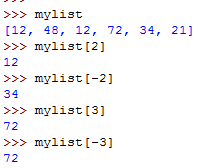
>>> mylist

## 4.1.2 List Slices

We can select the specific subset from the list using slicing. We can either use a positive index (forward) or negative index(reverse) to refer the particular element or slice in the list.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Positive index | 0 | 1 | 2 | 3 | 4 | 5 |
| mylist | 12 | 48 | 12 | 72 | 34 | 21 |
| Negative index | -6 | -5 | -4 | -3 | -2 | -1 |

Example:

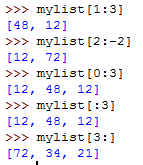


Slicing of the list has the following format:

mylist[start:end:step]

If step is not mentioned, it is taken as 1 as default. The slice includes the elements from start index upto end (**excluding end**). The elements are picked in step.

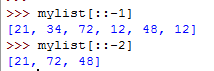
Example:



Elements at odd indices can be sliced as follows:



The list can be revered as follows:



Exercises

1. What is the output?

>>> mylist = [12, 48, 34, 72, 56]

>>> mylist = mylist[1:-2]\*2

1. What is the output?

>>> mylist = [12, 48, 34, 72, 56]

>>> mylist = mylist[::2] + mylist[-1::-2]

## 4.1.3 List Methods

|  |  |  |
| --- | --- | --- |
| **Method** | **Use** | **Example** |
| count(x) | Number of occurrences of x |  |
| index | return first index of value |  |
| insert | insert object before index |  |
| append | append object to end |  |
| extend | extend list by appending elements from the iterable |  |
| L.copy() | a shallow copy of L |  |
| reverse | reverse \*IN PLACE\* |  |
| sort | sort \*IN PLACE\* |  |
| remove | remove first occurrence of value |  |
| pop | remove and return item at index  (last by default) |  |

Associated methods and attributes of a list may be viewed with **dir(mylist)**.

Exercises:

1. What is the error?

>>> mylist = [12, 48, 34, 72, 56]

>>> mylist.pop(2)

>>> mylist.append(mylist.index(34))

1. What is the output?

>>> mylist = [12, 48, 34, 72, 56]

>>> mylist.remove(34)

>>> mylist.insert(2,2)

>>> mylist.sort()

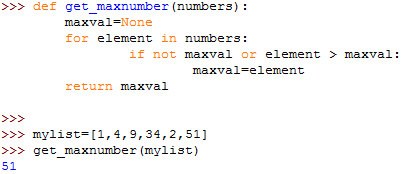
>>> mylist.reverse()

>>> mylist.append(mylist.count(2))

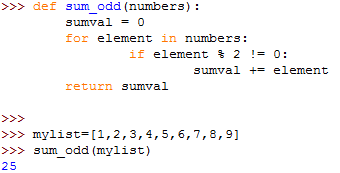
>>> mylist

## 4.1.4 List Loop

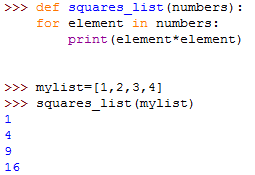
List is the collection of iterable items. Using for loop, you can process each element in the list.



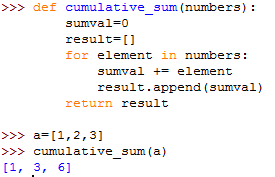
Example: Find the sum of odd numbers in the list



Example: Find the square of each element of the list



Example: Find the cumulative sum of a list. For example, [1,2,3] returns [1,3,6]

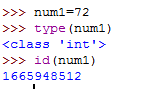


Exercise:

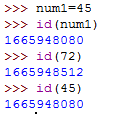
1. Find the sum of N numbers (using List)
2. Create list with the following pattern for the input num:  
   Eg:   
   num = 4 mylist = [4, 8, 12, 16, 12, 8, 4]  
   num=3 mylist = [ 3, 6, 9, 6, 3]
3. Create list with the following pattern for the input num:  
   Eg:   
   num = 4 mylist = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15]  
   num = 3 mylist = [1, 2, 4, 5, 7, 8]
4. Write a function to find the factorial of ‘n’?
5. Find the sum of ‘n’ terms of the series  
   f = 0! + 1! + 2! + … + n! (n >= 0)
6. Find whether ‘n’ is the factorial number

## 4.1.5 Mutability

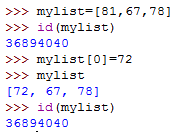
Everything is an object in python. For example, when a variable ‘num1’ is created, python allocates the memory location for the constant 72 and creates the unique identifier to refer that location. The variable num1 just refers to the memory location of the object 72.



When you assign another object to the same variable, it is saved in new location, without modifying the old object (72).



Thus the data is **‘immutable’** (not changeable or erasable). All datatypes in python such as int, bool, tuple are immutable. Only exceptions are **lists** and dictionaries, which are **mutable** (changeable).



Exercises:

1. What is the output?

>>> a = 72

>>> mylist = [a]

>>> before = id(mylist)

>>> mylist[0] = 34

>>> after = id(mylist)

>>> before == after

1. What is the output?

>>> a = 72

>>> before = id(a)

>>> a = 34

>>> after = id(a)

>>> before == after

1. What is the output

>>> a = 72

>>> mylist = [[a]\*2]

>>> a = 34

>>> mylist += [[a]\*3]

>>> before = id(mylist[1])

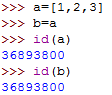
>>> mylist[1][2] = 45

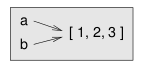
>>> after = id(mylist[1])

>>> before == after

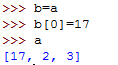
## 4.1.6 Aliasing

If an object is referred by more than one variable name, it is aliased.





As list is mutable, a change by one reference is reflected in other reference, as both refer to the same list object.



Exercises:

1. What is the output?

>>> a = [12,'python',True]

>>> b = a

>>> b[2] = False

>>> id(a) == id(b)

## 4.1.7 Cloning Lists

|  |  |  |
| --- | --- | --- |
| L.copy() | create a shallow copy of L |  |
| deepcopy |  |  |

In shallow copy, the nested sublists are not cloned (same id). In deep copy, they are cloned (different id).

Exercise:

1. Modify the program to get the desired output

>>> old\_stock = [['item1',23],['item2',34],['item3',45]]

>>> new\_stock = old\_stock.copy()

# Add 10 to each item

>>> for i in range(3):

new\_stock[i][1] += 10

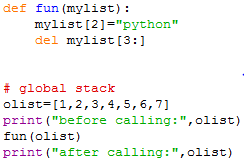
# old\_stock should not be changed

## 4.1.8 List parameters

When the list is passed to a function as parameter, the parameter refers to the same object.

Hence any change in the function gets reflected in the calling stack as well.

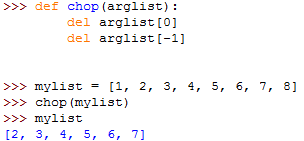
Example:



Output:



Example: Write the function ‘chop’ that takes a list, modifies it by removing the first and last elements and returns None.



Exercise:

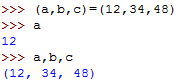
1. Write a function cat\_num which takes a list, say, [1,2,3,4,5] and modifies to [11,22,33,44,55] (concatenates each element itself) and returns None.

# 4.2 Tuples

|  |  |
| --- | --- |
| **List** | **Tuple** |
| created with [ ] | created with ( ) |
| Indexing and slicing | Same as in list |
| Mutable sequence  (append, remove, insert, pop, reverse, sort, extend and copy methods modify the list) | Immutable sequence  (No methods to modify the tuple) |
| Common methods are index() and count() | index() and count() methods are available, as they don’t modify the tuple. |

## 4.2.1 Tuple Assignment

Multiple variables can be assigned using tuple assignment (tuple unpacking). Parentheses are optional.



Exercise

1. What is the output

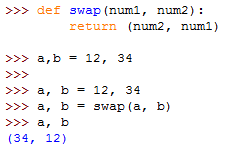
>>> a,b,c = 10, 00, 000

>>> (a, b, c)\*2

>>> a,b,c

## 4.2.2 Tuple as return value

Mutiple variables can be returned from the function using tuple. Parantheses are optional.



Exercise

1. Write the function quotient\_reminder to return quotient and reminder of a/b

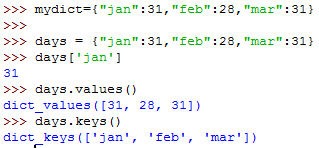
# 4.3 Dictionaries

Lists and tuples are ordered sequence. The elements are accessed using index.





Dictionary is the unordered sequence. The elements are accessed using key.

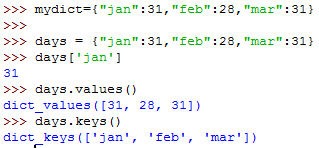


## 4.3.1 Operations and methods

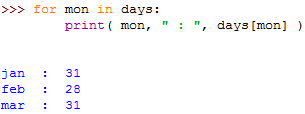
In dictionaries, the elements are stored as “key-value” pair.

keys() return all the keys in the dictionary.

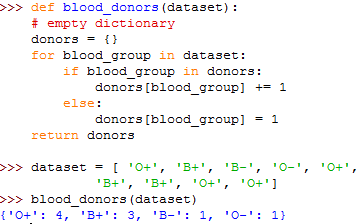
values() return all the values in the dictionary.



All the items are iterable in dictionary.



Example: Find the number of donors – blood group wise.



Exercise:

1. Write the function letters\_freq to find the frequency of letters in a string. Return the result as the dictionary.
2. Find the capital for the given country from the imported dictionary capital

from country import capital

def find\_capital(country):

# your code

1. Find the country for the given capital.

from country import capital

def find\_country(capital):

# your code

1. Find the countries for the given capitals.

from country import capital

def find\_countries(capitals):

# your code

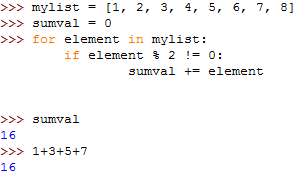
Example: input = [‘New Delhi’,’Washington DC’] output = [‘India’,’US’]

# 4.4 Advanced List Processing

## 4.4.1 List Comprehension

List comprehension is the pythonic way (one liner) to write the list loop.

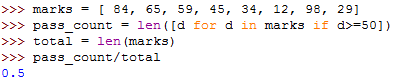
Example: Find the sum of odd numbers in the list.



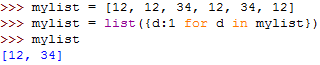
This can be written in one line using list comprehension.



Example: Find the pass percentage



Example: Remove duplicates from the list (using dictionary)



Exercises:

1. What is the output?

>>> num = [ d\*\*3 for d in range(1,10)]

>>> num

1. What is the output?

>>> cols = [1,2]

>>> row = [12, 'medium', 23.25, True]

>>> [ row[col] for col in cols]

1. Split the time “7:20 am” and store it in hour and min as integers.(in 24 hour format). Then find the time elapsed and return the value.  
   Example:   
   start: “7:20 am”, end: “12.30 pm” output: 5 hr 10 min

def elapsed(start,end):

#your code here

# 4.5 Illustrative Programs

## 4.5.1 Selection sort

Exercises:

1. Assume that first number in the list is minimum. Exchange, if first> second

Example

input = [12,3,15,7,23] output = [3,12,15,7,23]

1. Assume that first element in the list is minimum. Compare it with every other element. Exchange if it is greater. (index selected = 0)

Example:

[12,23,15,7,3] As 12<23, don’t exchange.

[12,23,15,7,3] As 12<15, don’t exchange.

[12,23,15,7,3] As 12>7, exchange

[7,23,15,12,3] As 7>3, exchange

[3,23,15,12,7] Stop.

1. Now, the first element is the minimum. Now, bring the next minimum value in the list as the second element. (index selected = 1)

Example:

[3,23,15,12,7] As 23>15, exchange

[3,15,23,12,7] As 15>12, exchange

[3,12,23,15,7] As 12>7, exchange

[3,7,23,15,12] stop

If we continue to place the subsequent minimum values, we get the sorted list.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| selected index  (outer loop) | numbers | | | | |
| Before sorting | 12 | 3 | 45 | 17 | 15 |
| 0 | 3 | 12 | 45 | 17 | 15 |
| 1 | 3 | 12 | 45 | 17 | 15 |
| 2 | 3 | 12 | 15 | 45 | 17 |
| 3 | 3 | 12 | 15 | 17 | 45 |

Selected index: 2 sorted in steps

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| After inner  Iteration  (j) | numbers | | | | |
| before sorting | 3 | 12 | 45 | 17 | 15 |
| 3 | 3 | 12 | 17 | 45 | 15 |
| 4 | 3 | 12 | 15 | 45 | 17 |

Algorithm

1. Select an index (i) successively from 0 to len(numbers)-2
2. Compare numbers[i] with each element in the remaining list
3. Swap numbers[i] with the element whenever numbers[i] is larger

Pseudocode

selection\_sort(numbers):

N=len(numbers)

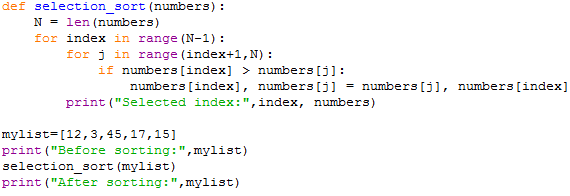
for index in range(N-1):

for j in range(index+1,N):

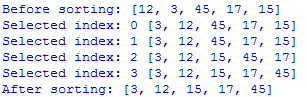
If numbers[index] > numbers[j]:

swap (numbers[index], numbers[j])

Implementation



Output



## 4.5.2 Insertion sort

Exercises:

1. Consider the second element in the list num. Insert at index 0, if element < first. hint: use insert()
2. Remove element if it is inserted. hint: use pop() or remove
3. Now num[0:1] is in sorted order. Now, consider the third element in the list (num[2]). Compare with first two elements. Insert at 0, if element is less than first. Insert at 1, if element is less than second. Remove num[2], if it is inserted.

Subsequently, the list num gets sorted.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | position to be inserted | num | | | | |
| 1 | 0 | 12 | 3 | 45 | 17 | 15 |
| 2 | 2 | 3 | 12 | 45 | 17 | 15 |
| 3 | 2 | 3 | 12 | 45 | 17 | 15 |
| 4 | 2 | 3 | 12 | 17 | 45 | 15 |
| sorted | | 3 | 12 | 15 | 17 | 45 |

Pseudocode

insertion\_sort(num):

for i in range(1,len(num)):

element = num[i]

inserted = False

for j in range(i):

if element < num[j]

insert element at j   
and break loop

if inserted:

remove element from i

Implementation

def insertion\_sort(num):

for i in range(1,len(num)):

element = num[i]

for j in range(i):

if element < num[j]:

print(num,"insert",element,"at",j)

num.insert(j,element)

num.pop(i+1)

break

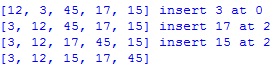
#Test

mylist = [12,3,45,72,15]

insertion\_sort(mylist)

print(mylist)

Output



Pseudocode (v2)

insertion\_sort(num):

for i in range(1,len(num)):

element=num[i]

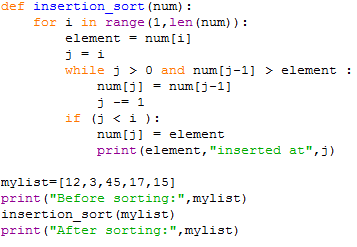
j=i

while j > 0 and num[j-1] > element:

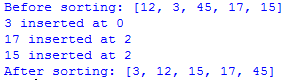
num[j]= num[j-1]

num[j]= element

Implementation (v2)



Output



## 4.5.3 Merge sort

It is divide recursively and conquer approach.

Exercise:

1. Consider left and right lists of size 1. Merge them in a sorted order.

Example:

left = [12] right = [3]

merged = [3,12]

1. Now consider the two sorted lists of unspecified size. Merge them in a sorted order.

Example:

left = [12,45] right = [3,17]

merged = [3,12,17,45]

1. Divide the list num into left and right halves.
2. Recursively divide, till the partition size is 1

Example:

num = [12,3,45,17,15]

left = [12,3]

left = [12]

right = [3]

right = [45, 17, 15]

left = [45]

right = [17,15]

left = [17]

right = [15]

Algorithm:

1. Divide the list recursively to left and right halves, till the partition size is 1
2. Merge the left and right halves in the sorted order

Algorithm for merge

1. Remove the minimum of two lists left and right   
   and add it to the merged list  
   till left or right becomes empty.
2. Append the remaining elements of left and right to merged list

Note: Both left and right are in sorted order, before merging.

Pseudo code

**merge\_sort(num)**

return divide(num)

**divide(num)**

if num is empty or len(num) is 1:

return num

mid = len(num)/2

left = divide(num[:mid])

right = divide(num[mid:])

merge(left,right)

**merge(left,right)**

merged\_list = [ ]

while left and right are not empty:

if left[0] < right[0]:

Pop left[0] and add it to merged\_list

else:

Pop right[0] and add it to merged\_list

Append remaining left and right to merged\_list

Implementation

def merge\_sort(num):

return divide(num)

def divide(num):

print(num)

if not num or len(num) == 1:

return num

else:

mid = len(num)//2

print("divide left:", end=' ')

left = divide(num[:mid])

print("divide right:", end=' ')

right = divide(num[mid:])

return merge(left,right)

def merge(left, right):

merged\_list = []

print("merging:",left,right,end=' ')

while left and right:

if left[0] < right[0]:

merged\_list += [left.pop(0)]

else:

merged\_list += [right.pop(0)]

merged\_list += left

merged\_list += right

print("merged:",merged\_list)

return merged\_list

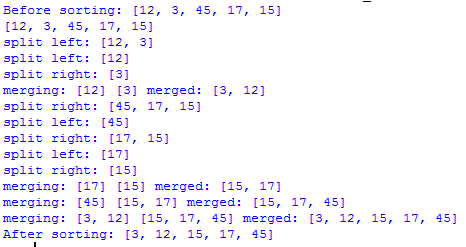
mylist=[12,3,45,17,15]

print("Before sorting:",mylist)

mylist = merge\_sort(mylist)

print("After sorting:",mylist)

Output:



## 4.5.4 Quick Sort

Exercises

1. Select last element of the list num as pivot.
2. Find from the front, which element is larger than or equal to pivot (num[front]) Find from the rear next to pivot, which element is smaller than pivot (num[rear]) Swap num[front] and num[rear] if front < rear
3. Repeat step 2 till front <= rear
4. Now the first half of num holds values smaller than pivot. Second half of num excluding pivot holds vlaues larger than pivot. Now, front points to the start of the larger partition. Swap pivot and num[front]. to bring pivot to the middle.

Example

num = [12,3,17,45,15,12]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 3 | 17 | 45 | 15 | 12 |
| front |  |  |  | rear | pivot |
| 15 |  |  |  | 12 |  |
|  | front |  |  |  |  |
|  |  | front |  |  |  |
|  |  |  | rear |  |  |
|  |  | rear |  |  |  |
|  | rear |  |  |  |  |
|  |  | 12 |  |  | 17 |
| 12 | 3 | 12 | 45 | 15 | 17 |
| partition with values  ‘smaller’ than pivot | | pivot | large | | |

Algorithm

1. Pick last element as a pivot from the num list
2. Divide num into small and large partitions  
   which contain elements smaller or larger than pivot
3. Recursively divide till partition size becomes 1

Pseudocode:

Qsort(num,firt,last):

pivot=last

front=first

rear=last-1

while front < rear:

increment front till num[front]< pivot

decrement rear till num[rear] >= pivot

if front <rear:

swap num[front],num[rear]

else:

break

swap num[front],pivot

Qsort(num,first, front-1) # partition small recursively

Qsort(num,front+1,last) # partition large recursively

Implementation

def quick\_sort(num):

if len(num)<=1:

return

Qsort(num,0,len(num)-1)

def Qsort(num,first,last):

print(num[first:last+1])

if first >= last:

return

pivot=last

front=first

rear=last-1

print("pivot=",num[pivot])

while front <= rear:

while num[front] < num[pivot] and front <= last:

front += 1

while num[rear] >= num[pivot] and rear >= first:

rear -= 1

if front < rear :

num[front],num[rear] = num[rear],num[front]

else:

break

num[front],num[pivot] = num[pivot],num[front]

if first <= front-1:

print("partition small", end=' ')

Qsort(num,first, front-1)

if front+1 <= last:

print("partition large", end=' ')

Qsort(num,front+1,last)

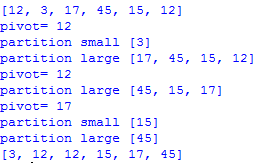
# Test

num=[12,3,17,45,15,12]

quick\_sort(num)

print(num)

Output



# Syllabus

**PROBLEM SOLVING AND PROGRAMMING**                                                              L T P C  
  
                                                                                                                                                3  0 0 3  
COURSE OBJECTIVES  
  
To develop an understanding of algorithmic problem solvingTo read and write simple Python programs.To develop Python programs with conditionals and loops.To define Python functions and call them.To use Python data structures –- lists, tuples, dictionaries.To do input/output with files in Python.  
  
UNIT I           ALGORITHMIC PROBLEM SOLVING                        9  
  
 Algorithms, building blocks of algorithms (instructions/statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing algorithms (iteration, recursion)  
   
UNIT II            DATA, EXPRESSIONS, STATEMENTS                        9  
  
Python interpreter and interactive mode; values and types: int, float, booleans, strings, and lists; variables, expressions,  statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments;  Illustrative programs: exchange the values of two variables, circulate the values of n variables, test for leap year.  
  
 UNIT III           CONTROL FLOW, FUNCTIONS                  9  
  
Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions: return values, parameters, scope: local and global, composition, recursion; Strings:  string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum the array of numbers, linear search, binary search.  
  
 UNIT IV           COMPOUND DATA: LISTS, TUPLES, DICTIONARIES        9  
  
Lists, list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples, tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension, Illustrative programs: selection sort, insertion sort, merge sort, quick sort.  
  
 UNIT V            FILES, MODULES, PACKAGES                          9  
  
Files and exception: text files, reading and writing files, format operator, command line arguments,  errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file,  
  
TOTAL:45 PERIODS

COURSE OUTCOMES  
  
Upon completion of the course, students will be able to  
  
Develop algorithmic solutions to simple computational problemsRead, write, execute by hand simple Python programs.Structure simple Python programs for solving problems.Decompose a Python program into functions.Represent compound data using Python  lists, tuples, dictionaries.Read and write data from/to files in Python Programs.  
  
 TEXT BOOKS:  
  
Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist’’,  2nd edition, Updated for Python 3, Shroff/O’Reilly Publishers, 2016  (<http://greenteapress.com/wp/think-python/>)  
  
   
  
REFERENCES:

1. John V Guttag, “Introduction to Computation and Programming Using Python’’, Revised and expanded Edition, MIT Press , 2013
2. Robert Sedgewick, Kevin Wayne, Robert Dondero, “Introduction to Programming in Python: An Inter-disciplinary Approach, Pearson India Education Services Pvt. Ltd., 2016.
3. Timothy A. Budd, “Exploring Python”, Mc-Graw Hill Education (India) Private Ltd.,, 2015.
4. Kenneth A. Lambert,  “Fundamentals of Python: First Programs”, CENGAGE Learning, 2012.
5. Charles Dierbach, “Introduction to Computer Science using Python: A Computational Problem-Solving Focus, Wiley India Edition, 2013.
6. The Python Tutorial, <https://docs.python.org>