**UNIT IV: COMPOUND DATA**

**LISTS, TUPLES, DICTIONARIES**

**Content**

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

**4 COMPOUND DATA**

Primitive data types are basic data types such as int, bool and float. Compound data is any data type which is constructed using primitive data types and other compound data types. Python offers different compound data types (sequences) such as lists, tuples and dictionaries.

A compound data type is one that holds multiple independent values.

**Example**

Create a compound data type to represent a point in the cartesian coordinates.

A point may be represented with a list, tuple, dictionary or a class in python.

x, y = 0, 1 # index

pointA = [2, 3] # list

print(pointA[x])

pointB = (4, 5) # tuple

print(pointB[y])

pointC = {'x':6, 'y': 7} # dictionary

print(pointC['x'])

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

pointD = Point(x=8, y=9)

print(pointD.x)

**4.1 LISTS**

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

A List is the built-in ordered sequence. A sequence is an iterable which supports efficient element access using integer indices via the getitem() special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and bytes.

Despite its name, list is more akin to an array in other languages than to a linked list since access to elements are O(1).

**Examples**

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

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

**Example**

>>> grocery = ['bread', 'butter', 'milk']

>>> absentees = [3, 14, 24, 35, 37, 41]

>>> movie\_review = ['enthiran', {'5-rating':344, '4-rating': 28, '3-rating':0}]

>>> my\_friends = ['akil', 'kapil', 'dhoni']

>>> my\_favorite\_menu = ['idli','dhosa','pongal']

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**

Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.

**repeat (\*)**

>>> mylist = [1, True, 'python']

>>> mylist \* 2

[1, True, 'python', 1, True, 'python']

**concatenate (+)**

>>> part1 = ['python','is']

>>> part2 = ['all', 'purpose', 'language']

>>> part1 + part2

['python','is','all', 'purpose', 'language']

**empty list**

>>> a = []

>>> not a

True

**index**

>>> mylist = [12, 48, 12, 72, 34, 21]

>>> mylist[1]

48

>>> mylist[0]

12

**Exercises**

1. What is the output for the following code?

>>> a = 10

>>> mylist = [a]\*5

>>> mylist[3]

1. What is the output for the following code?

>>> 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.

A slice is an object usually containing a portion of a sequence. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable\_name[1:3:5]. The bracket (subscript) notation uses slice objects internally.

| **Forward index** | **0** | **1** | **2** | **3** | **4** | **5** |
| --- | --- | --- | --- | --- | --- | --- |
| mylist | 12 | 48 | 12 | 72 | 34 | 21 |
| Reverse index | -6 | -5 | -4 | -3 | -2 | -1 |

**Example**

>>> mylist = [12, 48, 12, 72, 34, 21]

>>> mylist

[12, 48, 12, 72, 34, 21]

>>> mylist[2]

12

>>> mylist[-2]

34

>>> mylist[3]

72

List may be sliced into part, from start till end.

mylist[start:end:step]

The elements are picked in steps from start. If step is not mentioned, it is taken as 1 as default. The element at end is not included.

**Example**

>>>mylist = [12, 48, 12, 72, 34, 21]

>>> mylist[1:3]

[48, 12]

>>> mylist[2:-2]

[12, 72]

>>> mylist[0:3]

[12, 48, 12]

>>> mylist[:3]

[12, 48, 12]

>>> mylist[3:]

[72, 34, 21]

# Elements at odd indices

>>> mylist[::2]

[12, 12, 34]

# In reverse order

>>> mylist[::-1]

[21, 34, 72, 12, 48, 12]

>>> mylist[::-2]

[21, 72, 48]

Slice can be passed as an object as well.

>>>mylist = [12, 48, 12, 72, 34, 21]

>>> start, end, step = 1, None, 2

>>> items = slice(start, end, step)

>>> mylist[items]

[48, 72, 21]

**4.1.3 LIST METHODS**

**count(x)**

return the number of times x appears in the list.

>>> mylist = [12, 12, 34, 34, 34]

>>> mylist.count(34)

3

**index(x)**

return: the index of first occurence of x

>>> mylist.index(34)

2

**insert(index,x)**

insert an item at a given position(index).

>>> mylist

[12, 12, 34, 34, 34]

>>> mylist.insert(3,10)

# insert 34 at position 3

>>> mylist

[12, 12, 34, 10, 34, 34]

list.append(x) Add an item to the end of the list; equivalent to a[len(a):] = [x].

>>> mylist

[12, 12, 34, 34, 34]

>>> mylist.append([2,3])

>>> mylist

[12, 12, 34, 34, 34,[2,3]]

list.extend(L) Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

>>> mylist

[12, 12, 34, 34, 34]

>>> mylist.extend([2,3])

>>> mylist

[12, 12, 34, 34, 34, 2, 3]

list.remove(x) Remove the first item from the list whose value is x.

>>> mylist

[12, 12, 34, 34, 34]

>>> mylist.remove(34)

>>> mylist

[12, 12, 34, 34]

list.pop([i]) Remove the item at the given position in the list, and return it. If i is not mentioned, the last element is popped out.

>>> mylist

[12, 12, 34, 34, 34]

>>> mylist.pop(2)

34

>>> mylist

[12, 12, 34, 34]

>>> mylist.pop()

>>> mylist

[12, 12, 34]

list.sort() Sort the items of the list in place

list.reverse() Reverse the elements of the list, in place.

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

Exercises:

1. What is the error in the following code?

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

>>> mylist.pop(2)

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

1. What is the output of the following code?

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

>>> mylist.remove(34)

>>> mylist.insert(2,2)

>>> mylist.sort()

>>> mylist.reverse()

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

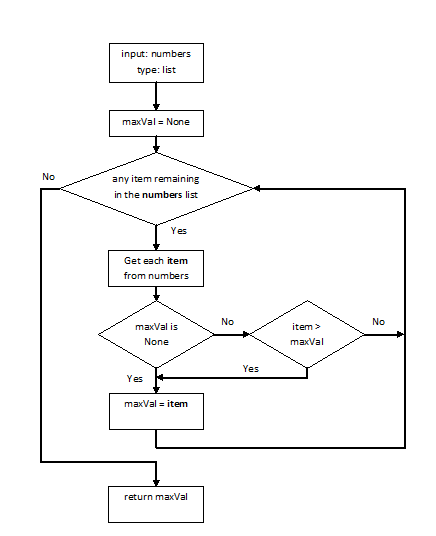
>>> mylist

**4.1.4 LIST LOOP**

List is the collection of iterable items. Using for construct, you can process each element in the list. Do not add or remove an element from the list in its for loop.

**Example**

Find the maximum number in the list

[](https://github.com/ashok-cs/PSP/raw/master/img/get_maxnumber.png)

def get\_maxnumber(numbers):

maxval = None

for element in numbers:

if not maxval or element > maxval:

maxval = element

return maxval

# test

mylist = [1, 5, 67, 34, 128]

print(get\_maxnumber(mylist))

The **in** construct on its own is an easy way to test if an element appears in a list.

key = 5

if key in mylist:

print("Key found")

You can also use for/in to work on a string type.

**Range**

The combination of the for-loop and the range() function allow you to build a traditional numeric for loop.

## print the numbers from 0 through 99

for i in range(100):

print i

range(100) creates a range object, yielding the elements from 0 to 99 (excluding 100). range(5,10) creates a range object, yielding the elements from 5 to 9 (excluding 10). range(1, 100, 2) yields odd numbers till 100.

The general syntax for range is range(start, stop, step)

**while Loop**

Python also has the standard while-loop, and the *break* and *continue* statements.

## Access every 3rd element in a list

i = 0

while i < len(a):

print a[i]

i = i + 3

**Exercise**

1. Find the sum of N numbers (using List)
2. Create list with the following pattern for the input num:

Example:

num = 4 mylist = [4, 8, 12, 16, 12, 8, 4]

num = 3 mylist = [ 3, 6, 9, 6, 3]

1. Create list with the following pattern for the input num:

Example

num = 4 mylist = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15]

num = 3 mylist = [1, 2, 4, 5, 7, 8]

1. Write a function to find the factorial of ‘n’?
2. Find the sum of ‘n’ terms of the series

f = 0! + 1! + 2! + … + n! (n >= 0)

1. Find whether n is the factorial number

**4.1.5 MUTABILITY**

Everything is an object in python. Every object has an identity(id) and value(mutable or immutable). \*\*Immutable \*\*: The value of the object can not be changed.

Mutable objects can change their value but keep their id(). An immutable object has a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

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.

>>> num1 = 72

>>> type(num1)

<class 'int'>

>>> id(72)

1695636320

>>> id(num1)

1695636320

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

>>> num1 = 40

>>> id(40)

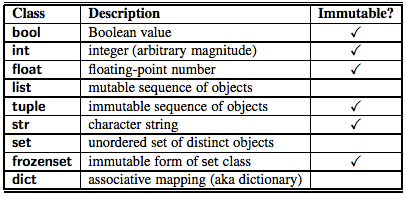
1695635808

>>> id(num1)

1695635808

>>>

Thus the data is ‘immutable’ (not changeable or erasable). All datatypes in python such as int, bool, tuple are immutable.

[](https://github.com/kgisl/pythonFDP/raw/master/img/mutablePython.png)

Only exceptions are lists,sets and dictionaries, which are mutable (changeable). \*\*Mutable \*\*: The id of the object is the same even after the value is changed.

>>> mylist = [1, 2, 3]

>>> id(mylist)

39075392

>>> mylist[1] = 100

>>> mylist

[1, 100, 3]

>>> id(mylist)

39075392

The effect of immutability:

>>> a = 45

>>> b = a

>>> b = 52

>>> a,b

(45,52)

a is not altered, when b is changed.

The effect of immutability:

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

>>> b = a

>>> b[0] = 50

>>> a

[50, 2, 3]

a is affected, when b is changed.

**4.1.6 Aliasing**

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

Assignment with lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

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

>>> b = a

>>> id(a), id(b)

(140143212216136, 140143212216136)

[](https://github.com/ashok-cs/PSP/raw/master/img/aliasing.jpg)

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

>>> b[1] = 100

>>> a

[1, 100, 3]

**Exercise**

What is the output for the following code?

>>> 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

>>> a = [2,3,4,[5,6]]

>>> b = a.copy()

>>> b[3][1] = 8

>>> a

[2, 3, 4, [5, 8]]

>>> id(b[3]),id(a[3])

(140522783809160, 140522783809160)

**deepcopy**

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

>>> from copy import deepcopy

>>> a = [2,3,4,[5,6]]

>>> b = deepcopy(a)

>>> b[3][1] = 8

>>> a

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

>>> id(b[3]),id(a[3])

(140522755061704, 140522783746376)

**Exercise**

1. Modify the program such that the addition to the new\_stock won't change the elements in old\_stock.

>>> 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 a parameter, the parameter refers to the same object (argument). Hence any change in the function gets reflected in the calling stack as well.

**Example**

def fun(mylist):

mylist[2] = 'python'

del mylist[3:]

# Test

olist=[1,2,3,4,5,6,7]

print("before calling:",olist)

fun(olist)

print("after calling:",olist)

ouput:

before calling: [1, 2, 3, 4, 5, 6, 7]

after calling: [1, 2, 'python']

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

>>> def chop(arglist):

del arglist[0]

del arglist[-1]

>>> mylist = [1, 2, 3, 4, 5, 6, 7, 8]

>>> chop(mylist)

>>> mylist

[2, 3, 4, 5, 6, 7]

**Useful links**

* **Python Visualizer** - [http://pythontutor.com](http://pythontutor.com/) visualizes the program execution statement by statement.
* Example: <http://j.mp/greatestKG>
* Online Python interpreter - [http://repl.it](http://repl.it/)
* Python inbuilt function documentation - <http://j.mp/pythonDoc>

**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.

**Additional questions**

1. What is the output of the following python code?

for elem in [ 1, 2, 3, 'abc', 99]:

print (elem\*2)

1. A list contains n elements (where n is a positive integer and 0 > n > 10. Write the necessary python code to produce a list that contains only the last n-1 elements. Is there a version of the code that does not use any list methods whatsoever to achieve the same result?
2. What is the value of L after you run the code below?

L = ['c', 'better', 'than', 'assembly', 0]

for item in L:

if item == 0:

L[0] = 'python'

L[3] = 'language'

L.pop()

elif item == 'assembly':

L[1] = 'multi'

L[2] = 'paradigm'

1. If n is the size of the list, and 1 < k < n, how will you find the kth largest number in the list of numbers? (a **#GTG**challenge, a Google interview question).
2. Complete the Hackerrank problem - <https://www.hackerrank.com/challenges/python-lists>

**4.2 Tuples**

List is the mutable sequence (append, remove, insert, pop, reverse, sort, extend and copy methods modify the list). Tuple is the immutable sequence. Only common methods for tuple and list are index() and count().

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 and tuples use parentheses, whereas lists use square brackets.

**4.2.1 Tuple Assignment**

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

>>> (a,b,c) = (12,34,48)

>>> a

12

>>> a,b,c

(12, 34, 48)

**Exercise**

1. What is the output for the following code

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

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

>>> a,b,c

**4.2.2 Tuple as a return value**

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

>>> def myswap(num1, num2):

return (num2, num1)

>>> a,b = 12,34

>>> a,b = myswap(a,b)

>>> a,b

(34, 12)

**Exercise**

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

def quotient\_reminder(a, b):

# 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.

A dictionary is an associative array, where arbitrary keys are mapped to values. The keys can be any object with hash() and eq() methods. Called a hash in Perl.

>>> days = {'jan':31, 'feb':28, 'mar':31}

>>> days['jan']

31

**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.

The objects returned from dict.keys(), dict.values(), and dict.items() are called dictionary views. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use list(dictview).

All the items are iterable in dictionary. Each key shall be accessed using for construct.

>>> for mon in days:

days[mon]

31

28

31

**Example**

Find the number of donors – blood group wise.

def blood\_donors(dataset):

# empty dictionary

donors = {}

for blood\_group in dataset:

if blood\_group in donors:

donors[blood\_group] += 1

else:

donors[blood\_group] = 1

return donors

#global frame

dataset = [ 'O+', 'B+', 'B-', 'O-', 'O+',

'B+', 'B+', 'O+', 'O+']

print(blood\_donors(dataset))

**Output**

{'B+': 3, 'O+': 4, 'O-': 1, 'B-': 1}

**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 in the format {capital: country}.

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. It gives the shorter and cleaner code.

result = []

for item in mylist:

if condition:

result.append(expression(item))

This can be rewritten using list comprehension.

result = [ expression(item) for item in mylist if condition ]

A list comprehension is the compact way to process all or part of the elements in a sequence and return a list with the results.

result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The if clause is optional. If omitted, all elements in range(256) are processed.

**Example**

Find the sum of odd numbers in the list.

>>> mylist = [1, 2, 3, 4, 5, 6, 7, 8]

>>> sumval = 0

>>> for element in mylist:

if element % 2 != 0:

sumval += element

>>> sumval

16

>>> 1+3+5+7

16

This can be written in one line using list comprehension.

>>> sumval = sum([d for d in mylist if d % 2 != 0])

>>> sumval

16

Find the pass percentage

>>> marks = [ 84, 65, 59, 45, 34, 12, 98, 29]

>>> pass\_count = len([d for d in marks if d>=50])

>>> total = len(marks)

>>> pass\_count/total

0.5

Example: Remove duplicates from the list (using dictionary)

>>> mylist = [12, 12, 34, 12, 34, 12]

>>> mylist = list({d:1 for d in mylist})

>>> mylist

[12, 34]

>>>

**4.5 Illustrative Programs**

**Select vs Insert vs Merge vs Quick**

In **selectsort**, the position of the update is pre-determined, starting from the end of the list. We then go select the maximum value among the unsorted elements of the list, and swap it with the element in the pre-determined location.

In **insertsort**, given a key, a copy of a pre-determined element in the list, we insert it at the appropriate location after comparing it with the unsorted elements of the list.

In **mergesort**, a divide-and-conquer partitioning algorithm (which more often requires extra memory), the input array is divided in two halves, calls itself recursively for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.

In **quicksort**, also a divide-and-conquer partitioning algorithm (lends itself to be efficiently implemented in-place without extra memory), the choice of the pivot element determines how the elements get partitioned, and calls itself recursively for the two partitions.

**Sorting**

Sorting is the arrangement of set in order

* sorting a set of numbers
* sorting a set of words

**Sorting a set of numbers**

Consider the list of prices of mobile phones of different brands (in thousands).

price = [ 54, 23, 45, 12, 14, 11, 10, 12]

Now, we want to find the first three mobile phones with least prices. (Example: sort by price option in Amazon)

Before sorting,

price = [ 54, 23, 45, 12, 14, 11, 10, 12]

After sorting,

price = [ 10, 11, 12, 12, 14, 23, 45, 54]

**Question**

Write the python code to display the price of first three items.

**Sorting a set of words**

Consider the set of new students joined in a program,

students = [ ‘akil’, ‘kapil’, ‘aakash’, ‘aswin’]

We want the computer to allot rollno for each student. For this, we need to sort the list first.

After sorting,

students = [‘aakash’, ‘akil’, ‘aswin’, ‘kapil’]

**Question**

Write the python code to allot rollno for each student.

students = {‘aakash’: '17CSE01', ‘akil’: '17CSE02',

‘aswin’: '17CSE03' , ‘kapil’: '17CSE04'}

**Importance**

* Sorting is the essential part of computing. Sorting takes ~ 30% of all the computing time spent.
* It is easier to search an item in the sorted set. Thus, sorting is the pre-requistie for the search algorithms.

**4.5.1 Selection sort**

In selection sort, select the minimum value from the unordered list and place it in the ordered list (index).

**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. Find the position of minimum element in the list (positionOfMin). Swap it with the element at the selected index 0.

Example:

[12, 23, 15, 7, 3]

index = 0

positionOfMin = 4

After swapping it with the element at the selected index 0,

[3, 23, 15, 7, 12]

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, 7, 12]

index = 1

positionOfMin = 3

After swapping it with the element at the selected index 1,

[3, 7, 15, 23, 12]

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

| **selected index** | **before swapping** | **positionOfMin** | **after swapping** |
| --- | --- | --- | --- |
| Before sorting | [12, 23, 15, 7, 3] |  |  |
| 0 | [12, 23, 15, 7, 3] | 4 | [3, 23, 15, 7, 12] |
| 1 | [3, 23, 15, 7, 12] | 3 | [3, 7, 15, 23, 12] |
| 2 | [3, 7, 15, 23, 12] | 4 | [3, 7, 12, 23, 15] |
| 3 | [3, 7, 12, 23, 15] | 4 | [3, 7, 12, 15, 23] |
| After sorting: | [3, 7, 12, 15, 23] |  |  |

**Algorithm**

1. Select an index successively from 0 to N-1 where N = len(numbers)-1
2. Find the positionOfMin element in the list from index to N
3. Swap the element at index with that of positionOfMin, if index != positionOfMin

**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**

def selection\_sort(numbers):

N = len(numbers)

for index in range(N-1):

positionOfMin = index

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

if numbers[positionOfMin] > numbers[j]:

positionOfMin = j

# Fill the selected index with minimum value

if (positionOfMin != index):

temp = numbers[positionOfMin]

numbers[positionOfMin] = numbers[index]

numbers[index] = temp

print("Selected index:",index, numbers)

#Test

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

mylist = [12,23,15,7,3]

print("Before sorting:",mylist)

selection\_sort(mylist)

print("After sorting:",mylist)

**Output**

Before sorting: [12, 23, 15, 7, 3]

Selected index: 0 [3, 23, 15, 7, 12]

Selected index: 1 [3, 7, 15, 23, 12]

Selected index: 2 [3, 7, 12, 23, 15]

Selected index: 3 [3, 7, 12, 15, 23]

After sorting: [3, 7, 12, 15, 23]

**4.5.2 Insertion sort**

**Exercises**

1. Consider the second element in the list num. Swap, if second < first.
2. Take the third element. Insert it such that first three elements are in sorted order.

Insertion sort always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.

Get the unsorted array [12, 3, 45, 17, 15] as input

Pick element 3 at index 1 and insert it at 0

[3, 12, 45, 17, 15]

such that [3,12] is in sorted order.

Pick element 45 at index 2.

[3, 12, 45, 17, 15]

No change as [3,12, 45] is already in sorted order

Pick element 17 at index 3 and insert it at 2

[3, 12, 17, 45, 15]

such that [3, 12, 17, 45] is in sorted order

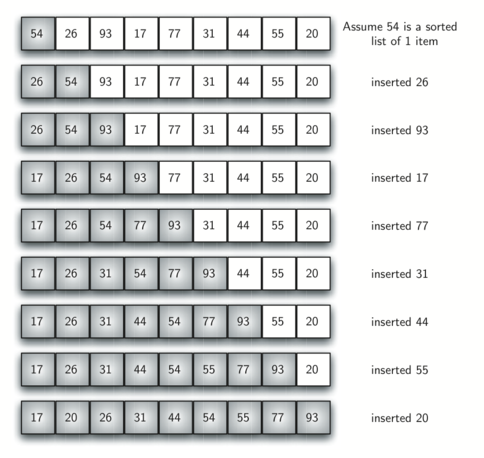
Pick element 15 at index 4 and insert it at 2

[3, 12, 15, 17, 45]

such that [3, 12, 15, 17, 45] is in sorted order

Now, the array is sorted completely.

[3, 12, 15, 17, 45]

[](https://camo.githubusercontent.com/b0a5ae82f41e33276a04382efab3461ad0bec3b6/687474703a2f2f696e746572616374697665707974686f6e2e6f72672f72756e6573746f6e652f7374617469632f707974686f6e64732f5f696d616765732f696e73657274696f6e736f72742e706e67)

**Pseudocode**

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**

def 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]

j -= 1

if (j < i ):

num[j] = element

print(element,"inserted at",j)

Output

[12, 3, 45, 17, 15] insert 3 at 0

[3, 12, 45, 17, 15] insert 17 at 2

[3, 12, 17, 45, 15] insert 15 at 2

[3, 12, 15, 17, 45]

**Related Material**

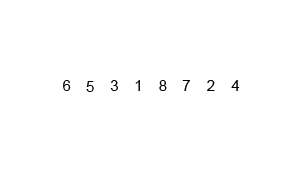
* <http://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html>
* <http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html>
* <http://bit.ly/insertVisualizer> - insertSort in the visualizer

**Exercises**

* Part 1 : <https://www.hackerrank.com/challenges/insertionsort1>
* Part 2 : <https://www.hackerrank.com/challenges/insertionsort2>
* Part 3 : <https://www.hackerrank.com/challenges/correctness-invariant>

**4.5.3 Merge sort**

It is divide recursively and conquer approach.

[](https://camo.githubusercontent.com/64ba2bcbd5c11779657e40a1d03d0ea691f6fa57/68747470733a2f2f75706c6f61642e77696b696d656469612e6f72672f77696b6970656469612f636f6d6d6f6e732f632f63632f4d657267652d736f72742d6578616d706c652d33303070782e676966)

Before sorting: [6, 5, 3, 1, 8, 7, 2, 3]

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

divide left: [6, 5, 3, 1]

divide left: [6, 5]

divide left: [6]

divide right: [5]

merging: [6] [5] merged: [5, 6]

divide right: [3, 1]

divide left: [3]

divide right: [1]

merging: [3] [1] merged: [1, 3]

merging: [5, 6] [1, 3] merged: [1, 3, 5, 6]

divide right: [8, 7, 2, 3]

divide left: [8, 7]

divide left: [8]

divide right: [7]

merging: [8] [7] merged: [7, 8]

divide right: [2, 3]

divide left: [2]

divide right: [3]

merging: [2] [3] merged: [2, 3]

merging: [7, 8] [2, 3] merged: [2, 3, 7, 8]

merging: [1, 3, 5, 6] [2, 3, 7, 8] merged: [1, 2, 3, 3, 5, 6, 7, 8]

After sorting: [1, 2, 3, 3, 5, 6, 7, 8]

**Exercises**

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.

**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

# Test

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

print("Before sorting:",mylist)

mylist = merge\_sort(mylist)

print("After sorting:",mylist)

**Output**

Before sorting: [12, 3, 45, 17, 15]

[12, 3, 45, 17, 15]

divide left: [12, 3]

divide left: [12]

divide right: [3]

merging: [12] [3] merged: [3, 12]

divide right: [45, 17, 15]

divide left: [45]

divide right: [17, 15]

divide left: [17]

divide right: [15]

merging: [17] [15] merged: [15, 17]

merging: [45] [15, 17] merged: [15, 17, 45]

merging: [3, 12] [15, 17, 45] merged: [3, 12, 15, 17, 45]

After sorting: [3, 12, 15, 17, 45]

**4.5.4 Quick Sort**

**Algorithm**

1. Pick a random 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

**Pseudo code**

quicksort(s):

if s contains less than 2 elements:

return s

pick an random element in s as pivot

small = elements in s that are smaller than pivot

equal = elements in s that are equal to pivot

large = elements in s that are larger than pivot

return quicksort(small) + equal + quicksort(large)

**Implementation**

import random

def quicksort(s):

len\_s = len(s)

if len\_s < 2:

return s

random\_index = random.randrange(0,len\_s)

pivot = s[random\_index]

small = [ x for x in s if x < pivot]

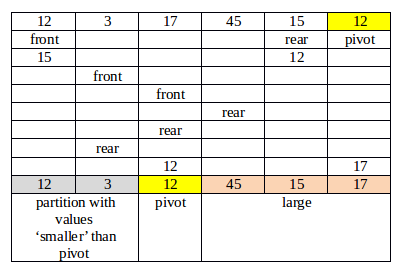
equal = [ x for x in s if x == pivot]

large = [ x for x in s if x > pivot]

return quicksort(small) + equal + quicksort(large)

**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]

[](https://github.com/ashok-cs/PSP/raw/master/img/quick_sort_pass.png)

**2 Mark Questions**

1. Define list
2. Define tuple
3. What any four list methods and their use
4. What is compound data
5. How do you concatenate two lists
6. How do you reverse a list using slice.
7. What is immutable object
8. What are the mutable data types in python
9. Is a string object mutable?
10. What is an alias. How do you create an alias for a list.
11. How do you clone a list
12. What is the risk in passing mutable objects such as list to a function.
13. What is the output of the following python code?

for elem in [ 1, 2, 3, 'abc', 99]:

print(elem\*2)

1. What is meant by tuple unpacking.
2. What are the ordered sequences in python
3. Define dictionary
4. How do you view all the keys of a dictionary.
5. What is list comprehension
6. What is the difference between selection sort and insertion sort.
7. What is divide-and-conquer algorithm. Give an example.
8. What is the difference between merge sort and quick sort.
9. Write a python code to display the sum of even numbers upto 100.
10. Write a function to find the factorial of N. (N!)
11. What will happen, if an variable refering to an immutable object is assigned with a new value.
12. Write a function to count the number of occurances of a key element in a list.

**Part B Questions**

1. Explain the use of different list methods with suitable examples.
2. Describe list comprehension with suitable examples.
3. Explain how selection sort algorithm works and implement the same in python.
4. Describe the insertion sort algorithm with python code.
5. Explain the merge sort algorithm and implement it in python.
6. Explain the quick sort algorithm and write the python code for the same.
7. Differentiate the data types list, tuple and dictionary with suitable examples.
8. Illustrate list loop and list mutability with suitable examples.
9. Differentiate aliasing and cloning of lists with examples.