MUTABLE AND IMMUTABLE OBJECTS

Elementary forms of data such as numeric and Boolean are called scale data types.

LISTS

A list is an ordered sequence of values. It is a non-scalar type. Values stored sequence closed in a list can be of any type such as string, integer, float, or list,

```
>>> subjects=('Hindi', 'English', 'Maths', 'History']
```
the elements of a list are enclosed in square brackets, separated by commas.
```
>>> id (subjects)
57135752
```

Next, examine the following statements:

```
>>> temporary = subjects
>>> id (temporary)
57135752
```

Note that each of the names **subjects** and **temporary** is associated with the same list object having object id **57135752**. Python Tutor representation of the lists **subjects** and **temporary**.

This method of accessing an object by different names is known as aliasing. Unlike strings, lists are mutable, and therefore, one may modify individual elements of a list.

```
>>> temporary [0] = 'Sanskrit'
>>> print (temporary)
['Sanskrit', 'English', 'Math', 'History' ]
>>> print (subjects)
['Sanskrit, 'English', 'Math', 'History’]
>>> print (id (subjects), id (temporary) )
57135752 57135752
```

As each of the two names **temporary** and **subjects** refers to the same list, on modifying the component **temporary [0]** of the list temporary, the change is reflected in **subjects [0]**
Next, we create a list of subjects and their corresponding subject codes. For this purpose, we represent each pair of subject and subject code as a list, and form a list of such lists:

```
>>> subjectCodes = [['Sanskrit', 43], ['English' 85], ['Maths', 65], ['History', 36]]
```

A list of lists such as **subjectCodes**, each of whose elements itself is a list, is called a two-dimensional list. Thus, **subjectcodes (1)** being a list, its components may be accessed as **subjectcodes (1] [0]** and **subjectCodes [1] [1]:**

```
>>> subjectCodes[1]
['English', 85]
>>> print (subjectCodes [1] [0], subjectCodes [1] [1])
English 85
```

In general, when we write **subjectCodes [i] Lj],** the first index i yields the list **subjectCodes [i]** in **subjectCodes**, and the second index yields the element **subjectCodes[i]lj]** within the list **subjectCodes[i]**

```
>>>details = ['Megha Verma', 'C-55, Raj Nagar, Pitam Pura, Delhi - 110034', 9876543210]
```

Often times, we need to take a list as an input from the user. For this purpose, we use the function input for taking the input from the user, and subsequently apply the function eval for transforming the raw string to a list:

```
>>> details = eval (input ('Enter details of Megha: ’))
Enter details of Megha: ['Megha Verma' 'C-55, Raj Nagar, Pitam Pura, Delhi – 110034’, 9876543210]
>>> details
 ['Megha Verma', 'C-55, Raj Nagar, Pitam Pura, Delhi - 110034', 9876543210]
```

The membership operator in may be used in a for loop for sequentially iterating over each element in the list, for example:

```
>>> students = ('Ram', 'Shyam', 'Gita', 'Sita']
>>> for name in students:
        print (name)
Ram
Shyam
Gita
Sita
```

Function list

The Function **list** takes a sequence as an argument and returns a list. For example, given a string of vowels **'aeiou'**, We can convert it to the list of vowels for further use as follows:

```
>>> vowels = 'aeiou'
>>> list (vowels)
[‘a’, 'e’, ‘i', 'o', 'u']
```

Functions append, extend, count, remove, index, pop, and insert

append: The function **append** insert the object passed to it at the end of the list, for example:

```
>>> a = [10, 20, 30, 40]
>>> a. append (35)
>>> a
[10, 20, 30, 40, 35]
```

extend: The function **extend** accepts a sequence as an argument and puts the elements of the sequence at the end of the list, for example:

```
>>> a = [10, 20, 30]
>>> a. extend ([35, 40] )
>>>a
[10, 20, 30, 35, 40]
>>> a. extend ('abc")
>>> a
[10, 20, 30, 35, 40, 'a', ‘b’, ‘c’]
```

count: The function **count** returns the count of the number of times the of occurrences of an object in a list object passed as an argument appears in the list, for example:

```
>>> a= [10, 20, 30, 10, 50, 20, 60, 20, 30, 55]
>>> a. count (20)
3
```

pop: The function **pop** returns the element from the list whose index is passed as an argument, while removing it from the list, for example:

```
>>> a = [10, 20, 30, 10, 50, 20, 60, 20, 30, 55]
>>> a.pop (3)
10
>>> a. pop (3)
50
>>> a
[10, 20, 30, 20, 60, 20, 30, 55]
```

remove: The function **remove** takes the value to be removed from the list as an argument, and removes the first occurrence of that value from the list, for example:

```
>>> a. remove (20)
>>> a
[10, 30, 20, 60, 20, 30, 55]
```

Next, suppose we have two lists, **names** - the list of the names of students and **rollNums**- the list of corresponding roll numbers:

```
>>> names = ['Ram', ‘Sita’,  'Shyam', 'Sita', 'Gita', ‘Sita’]
>>> rollNums = [1, 2, 3, 4, 5]
```

To remove a student, ' **Shyam'** from the lists, we cannot apply the function remove as we do not know the roll number of ' **Shyam'**. However, since there is a correspondence between student names and their roll numbers, knowing the index of a student in the list **names** will solve the problem. Python function **index** can be used for finding the index of a given value in the list.

```
>>> rollNums.pop (names . index ( 'Shyam' ) )
2
>>> names. remove ('Shyam" )
>>> print (names, rollNums)
['Ram', 'Sita', 'Gita', 'Sita'] [1, 3, 4, 5]
```

del: The **del** operator is used to remove a subsequence of elements (start: end: increment) from a list, for example:

```
>>> a = [10, 20, 30, 20, 60, 20, 30, 55]
>>> del a(2:6:3]
>>> a
[10, 20, 20, 60, 30, 55]
>>> del a
>>> a
```

```
Traceback (most recent call last) :
File "‹pyshel1#14>", line 1, in <module>
a
NameError: name ‘a' is not defined
```

Note that on execution of statement **del a**, name **a** no more refers to the list object. Informally, we may say that the object a has been deleted or **a** has been deleted.

Function insert

the **insert** function can be used to insert an object at a specified index. This function takes two arguments: the index where an object is to be inserted and the object itself, for example:

```
>>> names = ['Ram' 'Sita', ‘Gita', ‘Sita']
>>> names. insert (2, ‘Shyam' )
>>> names
['Ram', ‘Sita, 'Shyam', ‘Gita', 'Sita']
```

Function reverse

The function **reverse** reverses the order of the elements in a list, for example:

```
>>> names = ['Ram', 'Sita’, ‘Sita’, 'Anya' ]
>>> names. Reverse()
>>> names
['Anva', 'Sita’, 'Sita’, ‘Ram' ]
```

Functions sort and reverse

The function **sort** can be used to arrange the elements in a list in ascending order. For example, the names of students in the list names can be arranged in an ascending order using the function sort:

```
>>> names = ['Ram', ‘Sita’, ‘Sita, 'Anya' ]
>>> names . sort ()
>>> names
['Anya', 'Ram' 'Sita', 'Sita']
```

If we wish to sort the list of names in descending order, we may use the argument **reverse = True** while invoking the function **sort**:

```
>>> names =['Ram', 'Sita’, 'Sita', 'Anya' ]
>>> names. sort (reverse = True)
>>> names
['Sita'', ‘Sita', 'Ram', 'Anya' ]
```

Sort the strings in a list, based on their length. To achieve this, we need to provide an argument of the form **key=function** as illustrated below:

```
>>> names = ['Ram', ‘Sita', ‘Purushottam', 'Shyam' ]
>>> names. sort (key = len)
>>>names
['Ram', 'Sita', ‘Shyam', 'Purushottam']
```

The expression **[x**3 for × in range (1, end + 1)]** creates a list containing cube of each element in the **range (1, end + 1)**. Next, suppose we have a list comprising information about names and height of the students. The name of each student is followed by the height of the corresponding student in centimetres. We wish to create a list comprising the students in the list Ist whose height exceed or equal a threshold.

```
>>> lst = I['Rama', 160], ['Anya', 159], ['Mira', 158], ['Sona’, 161]]
>>> threshold = 160
```

The desired list can be easily created using comprehensions:

```
>>> tall = [x for x in lst if x[1] >= threshold]
>>> tall
[['Rama' , 160], ['Sona', 161]]
```

Next, given a list **s1** of alphabets and another list **s2** of numbers, we wish to create a cross product of **s1** and **s2**. Each element of the list **crossProduct** is a two-element list of the type [alphabet, number]. This is easily achieved using comprehensions:

```
>>> s1 = ['a',  'b', 'c']
>>> s2 = [3, 5]
>>> crossProduct = [[x, y] for x in s1 for y in s2]
>>> crossProduct
[['a', 3], ['a', 5], ['b', 3], ['b', 5], ['c', 3], ['c', 5]]
```

Lists as Arguments

Let us examine the script in Fig, 7.4. The function **listUpdate** updales the list **a** by replacing the object **a[i]** by **value**. The mail unction invokes the function **listUpdate** with the arguments **lst, 1** and **15** corresponding to the formal parameters **a**, **i**, and **value** As arguments are passed by reference, during execution of the functio **listUpdate**, an access to the formal parameter a means access to the list **lst** created in the **main** function, Consequently, when we update le list a in the function **listUpdate**, it results in the corresponding update or the list **lst**, as visualized in Python Tutor (Pig, 7,5). Thus, the value at index **1** of the list **lst** gets updated to the value **15.**

```
def listUpdate (a, 1, value):
   a [i] = value

def main) :
   1st = [10, 20, 30, [40, 50]]
   listUpdate (lst, 1, 15)
   print (lst)

if __name__== ‘__main__’:
   main ()
```

Copying list objects

As the names **list1** and **list2** refer to the same list object any changes made in the list will relate to both the names **list1**and **list2**.

However, sometimes we need to create a copy of the list as a distinct abject. Any create another instance of the list object having different storage, we need o import the copy module and invoke the function **copy.copy**

```
>>> import copy
>>> list1= = [10, 20, [30, 40]]
>>> list3 = copy.copy (list1)
```

Note that the **copy** function creates a new copy **list3** of the **list1**.
consequently, on modifying **list1**[**1**]**, list3**[**1**] remains unaltered:

However, the **copy** function creates a shallow copy i.e. it does not create copies of the nested objects. Thus, the two lists share the same nested Objects. Thus, **list1[2]** and **list3[2]** refer to the same nested list **[30, 40].** Next, let us modify **list1[2] [0]** in sub-list **list1[2]** of list **list1** to value **35**.

```
>>> list1(2] [0] = 35
>>> list1
[10, 25, [35, 40]]
>>> list3
[10, 20, [35, 40]]
```

If we wish to create a copy of a list object so that the nested objects (at all levels) get copied to new objects, we use the function **deepcopy** of the **copy** module:

```
>>> import copy
>>> list1 = [10, 20, [30, 40]]
>>> list4 = copy. deepcopy (list1)
```

map, reduce, and filter Operations on a Sequence

The function **map** is used for transforming every value in a given séquence by applying a function to it. It takes two input arguments: the iterable object (i.e. object which can be iterated upon) to be processed and the function to be applied, and returns the map object obtained by applyin, the function to the list as follows:

result = map (function, iterable object)

The function to be applied may have been defined already, or it maybe defined using a lambda expression which returns a **function** object. A lambda expression consists of the **lambda** keyword followed by a comma separated list of arguments and the expression to be evaluated using the list of arguments in the following format:

lambda arguments: expression

For example, let us define a lambda function that takes a number **x** as an input parameter and returns the cube of it. The following statement defines a lambda function and assigns it the name cube. The function cube can now be used as usual:

```
>>> cube = lambda x: X ** 3
>>> cube (3)
27
```

Similarly, the function sum2Cubes may be used to compute the sum of cubes of two numbers:

```
>>> sum2Cubes = lambda x, y: x**3 + y**3
>>› sum2Cubes (2, 3)
35
```

Next, suppose we wish to compute the cubes of all values in the list **lst**. The following statements use the map function to map every value in the list **lst** to its cube. Since the function map returns a **map** object, we have used the function **list** to convert it to list.

```
>>> 1st = [1, 2, 3, 4, 5]
>>> list (map (lambda x: x ** 3, lst))
[1, 8, 27, 64, 125]
>›› list (map (cube, lst) )
[1, 8, 27, 64, 125]
```

We may also use any system defined or user-defined function as an argument of the **map** function, for example:

```
>>> list (map (abs, [-1, 2, -3, 4, 5]))
 [1, 2, 3, 4, 5]
```

Suppose we wish to compute the sum of cubes of all elements in a list. Note that adding elements of the list is a repetitive procedure of adding two elements at a time. For this purpose the function **reduce** available in **functools** module is used. It takes two arguments: a function, and a iterable jot and applies the function on the iterable object to produce a single value

```
>>> lstcubes = list (map (lambda *: * ** 3,  lst))
>>> import functools
>>> sumCubes= functools.reduce (lambda x, y: x + y, lstCubes)
>>> sumCubes
225
```

Next, we compute the sum of cubes of only those elements in the list **lstcubes** that are even. Thus, we need to filter the even elements from the list **lstCubes** and before applying the function **reduce**. Python provides the function **filter** that takes a function and an iterable object is the input parameters and returns only those elements from the iterable object for which function returns **True**. In the following statement, the **filter** function returns an iterable object of even elements of the lis **lstCubes**, which is assigned to variable **evenCubes**. Since the function **filter** returns a filter object, we have used the function **list** in convert it to list.

Sets

A set in Mathematics refers to an unordered collection of objects without any duplicates. An object of type set may be created by enclosing the elements of the set within curly brackets. The set objects may also be created by applying the set function to lists, strings, and tuples. Some examples of sets:

```
>>> {1, 2, 3}
{1, 2, 3}
>>> vowels = set('aeiou")
>>> vowels
{'e', 'a', 'o', 'u', 'i'}
>>> vehicles = set (['Bike', 'Car', 'Bicycle', 'Scooter'])
>>> vehicles
{'Scooter', 'Car, 'Bike', 'Bicycle'}
>>> digits = set ( (0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
>>> digits
{10, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

The elements of a set are required to be immutable objects. Therefore, whereas objects of types such as int, float, tuple, and str can be members of a set, a list cannot be a member of a set. Unlike lists, we cannot access elements of a set through indexing or slicing. Also, we cannot apply + and * operators on sets. However, using the membership operator in, we can iterate over elements of a set:

```
>>> for v in vowels:
            print(v, end= ' ')
e a o u i
```

The functions min, max, sum, manner as defined for lists. and len work for sets in the same

```
>>> min (digits)
0
>>> max (digits)
9
>>> sun (digits)
45
>>> len (vehicles)
4
```

The membership operator a n checks whether an object is in the set.

```
>>> 'Bike' in vehicles
True
```

The **add** function is used to add an element to a set. The input argument to the **update** function maybe an object such as list, set, range, or tuple, for example:

```
>>> vehicles. add ('Bus')
>>> vehicles
{'Car', 'Bike', 'Bicycle', 'Bus', 'Scooter'}
>>> vehicles.update (['Truck', 'Rickshaw', 'Bike'])
>>> vehicles
{'Car',  'Bike', 'Bicycle', 'Rickshaw', 'Bus', 'Truck', 'Scooter'}
```

It is important to point out that an attempt to add a duplicate value (‘Bike’ in the above case) is just ignored by Python. Next, we define the set **heavyVehicles** as follows;

```
>>> heavyVehicles = ('Truck', "Bus', 'Crane")
```

An element may be removed from a set using the function remove, for example:

```
>>> heavyVehicles.remove ('Crane")
>>> heavyVehicles
{'Bus', 'Truck' }
>>> heavyVehicles.remove ( 'Car')

Traceback (most recent call last):
  File "<pyshell#432>", line 1
    heavyVehicles. remove ('Car' )
keyError: 'Car’
```

Note that an attempt to remove an element from a set, which is not a member of the set, yields an error. Although the function **pop** is defined for objects of type set, the element removed by the function **pop** is unpredictable.

```
>>> heavyVehicles = {Truck', 'Bus’, ‘Crane’}
>>> heavyVehicles.pop()
‘Bus’
```

To remove all the elements of a set, without deleting the set object we use the **clear** function**:**

```
››› heavyVehicles.clear ()
>›› heavyVehicles
Set()
```

Python provides functions **union**, **intersection**, and **symmetric_difference** to carry out these mathematical operations. We define sets **fruits** and **vegetables**:

```
>>> fruits = {'Apple', 'Orange', 'Tomato', ‘Cucumber’, ‘Watermelon’ }
>>> vegetables = {'Cucumber', ‘Potato’, ‘Tomato', 'Watermelon', ‘Cauliflower'}
```

Now, to determine all eatables whether Fruits or vegetables, we take union of the above two sets.

```
>>> eatables = fruits.union (vegetables)
>>> eatables
{'Orange', ‘Tomato', ‘Cucumber’, ‘Watermelon', ‘Cauliflower’, ‘Apple', 'Potato"}
```

Alternatively, we may use the union operator | for achieving the same result:

```
>>> eatables = fruits | vegetables
>>> eatables
{'Orange', 'Tomato', ‘Cucumber’, ‘Watermelon', 'Cauliflower', 'Apple', ‘Potato’}
```

Next, to determine those eatables which are both fruits and vegetables, we take intersection of sets **fruits** and **vegetables** using either the function **intersection** or the intersection operator as follows:

```
>>> fruitsAndVegetables = fruits.intersection (vegetables)
>>> fruitsAndVegetables
{'Tomato', ‘Cucumber', 'Watermelon" }
>>> fruitsAndVegetables = fruits & vegetables
>>> fruitsAndVegetables
{‘Tomato’, ‘Cucumber', ‘Watermelon’}
```

Next, we find the fruits which are not vegetables by taking difference of the two sets, either using the function **difference** or using the difference operator:

```
>>> onlyFruits = fruits.difference (vegetables)
>>> onlyFruits
('Orange', 'Apple }
>>> onlyFruits = fruits - vegetables
>>> onlyFruits
('Orange", 'Apple'}
```

Similarly, we can determine vegetables which are not fruits as follows:

```
>>> onlyVegetables = vegetables. difference (fruits)
>>> onlyVegetables
{'Cauliflower', 'Potato"}
>>> onlyVegetables = vegetables - fruits
>>> onlyVegetables
{‘Cauliflower, 'Potato" }
```

Next, to determine eatables which are either only fruits or only vegetables, we may take union of the sets **onlyFruits** and **onlyVegetables**:

```
>>> fruitsXORVegs = onlyFruits | onlyVegetables
›>› fruitsXORVegs
{'Orange', ‘Apple', ‘Cauliflower', 'Potato' }
```

Alternatively, we may use the function symmetric_difference, which determines the elements that are in one of the either sets but not in both:

```
>>> fruitsXORVegs = fruits.symmetric_difference (vegetables)
>>> fruitsXORVegs
{'Orange’, 'Apple', 'Cauliflower', 'Potato'}
```

The functions **set.intersection, set.union,** and **set.difference** can be applied to two or more sets to carry out **intersection**, **union**, and **difference** operations on sets:

```
>>> digits1 = {0, 1, 2, 3}
>>> digits2= {2, 4, 5, 6}
>>> digits3 = 10, 7, 8, 9, 2}
>>> set. union (digitsl, digits2, digits3)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> set. intersection (digits], digits2, digits3)
{2}
>>> set.difference (digitsl, digits2, digits3)
{1, 3}
```

The function **set.difference** works by taking the union of all sets except the first one and subtracting the union from the first set.

Function copy

We make use of the function **copy** to create a separate copy of a set so that changes in one copy are not reflected in the other:

```
>>> digits = (0, 1, 2, 3, 7, 8, 9}
>>> digits2 = digits. copy ()
>>> digits. update ({4, 5, 6})
>>> digits
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> digits2
{0, 1, 2, 3, 7, 8, 9}
```

Subset and Superset Test

To check whether a given set is the subset or superset of another set, we use the operators **<=** and **>=,** respectively. For example, let the sets **multiples2** and **multiples4** comprise multiples of 2 and 4, respectively. We can check whether **multiples2** is a subset or superset of **multiples4** using these operators:

```
>>> multiples2 = (2, 4, 6, 8, 10, 12, 14}
>>> multiples4 = (4, 8, 12}
>>> isSubset = multiples4 <= multiples2
>>> isSubset
True
>>> isSuperset = multiples2 >= multiples4
>>> isSuperset
True
```

The functions **issubset** and **issuperset** can also be used for the same purpose.

Finding Common Factors

Suppose, given two numbers **n1** and **n2**, we wish to find their common factors. To check whether a number **i** is a factor of another number, say **num**, we only need to find the remainder **num % i**. If the remainder is zero, then **i** is a factor of **num**, otherwise, **i** is not a factor of **num**. We represent the collection of common factors as a set. We also note that a common factor cannot exceed smaller of the two numbers, say **n1** and **n2**. To build a set of common factors, we make use of comprehensions. For each number **i** in **range (1, min (n1,2) + 1),** we include it in the set if it is a factor of each of **n1** and **n2**. Thus, our code comprises just one line:

```
    commonFactors= {i for i in range (1, min (n1+l, n2+1)) if n1%i == 0 and n2%i == 0}
```

It is emphasised that a list should be converted to sets only when the order of sequence is not important

```
def union (L1, H2) :
    return list (set (L1) | set (L2))

def intersection (L1, L2) :
    return list (set (Ll) & set (L2) )

def difference (L1, L2) :
    return list (set (Ll) - set (12) )
```

TUPLES

A non scalar type. Just like a list, a tuple is an ordered sequence of objects. However, unlike lists, tuples are immutable i.e., the elements of a tuple cannot be changed or overwritten. Specified by round brackets and elements are separated by commas.

```
>>> t1=(4, 6, [2,8], 'abc', {3,4})
>>> type(t1)
<class 'tuple'>
```

If a tuple comprises a single element, the element should be followed by a comma to distinguish a tuple from a parenthesized expression, for example:

```
>>> (2)
2
>>> (2,)
(2,)
```

A tuple having a single element is known as a singleton tuple. Another notation for tuple sis just to list the elements of a tuple, separated by commas:

```
>>> 2, 4, 6
(2, 4, 6)
```

Tuples being immutable, an attempt to modify the element of tuple yields an error:

```
>>> t1[1] =3
Traceback (most recent call last):
   File "<pyshell#61>", line 1, in <module>
      T1[1] =3
TypeError: 'tuple' object does not support item assignment
```

However, an element of a tuple may be an object that is mutable, for example:

```
>>> t1 = (1, 2, [3, 4])
>>> t1[2][1] = 5
>>> t1
(1, 2, [3, 5])
```

Tuple Operations

```
>>> t1 = ('Monday', 'Tuesday')
>>> t2 = (10, 20, 30)
```

Multiplication operator *****	>>> **t1 * 2**	**('Monday', 'Tuesday', 'Monday', 'Tuesday')**
Concatenation operator +	>>> **t3= t1 + ('Wednesday')**	**('Monday', 'Tuesday', 'Wednesday')**
Length operator **len**	>>> **len(t1)**	**2**
Indexing	>>> **t2[-2]**	**20**
Slicing Syntax **start:end:inc**	>>> **t1[1:2]**	**('Tuesday',)**
Function **max**	>>> **max(t2)**	**30**
Function **min**	>>> **max(t2)**	**10**
Function **sum**	>>> **sum(t2)**	**60**
Membership operator **in**	>>> **'Friday' in t1**	**False**

Functions *tuple* and *zip*

The function **tuple** can be used to convert a sequence to a tuple, for example:

```
>>> vowels = ‘aeiou’
>>> tuple(vowels)
(‘a’, ‘e’, ‘i', ‘o’, ‘u’)
>>>tuple([4, 10, 20])
(4, 10, 20)
>>>tuple(range(5))
(0, 1, 2, 3, 4)
```

The function **zip** is used to produce a zip object(iterable object), whose ith element is atuple containing ith element from each iterable object passed as argument to the **zip** function. We have applied **list** to convert the **zip** object to a list of tuples.

```
>>> colors = (‘red’, ‘yellow’, ‘orange’)
>>> fruits = (‘cherry’, ‘banana’, ‘orange’)
>>> quantity = (‘1kg’, 12, ‘5kg’)
>>> fruitColor = list(zip(colors, fruits))
>>> fruitColor
[(‘red’, ‘cherry’), (‘yellow’, ‘banana’), (‘orange’, ‘orange’)]
>>> fruitColorQuantity = list(zip(fruitColor, quantity))
>>> fruitColorQuantity
 [((‘red’, ‘cherry’), 1kg’), ((‘yellow’, ‘banana’), 12), ((‘orange’, ‘orange’), 5kg’)]
>>> list((zip(colors, fruits,quantty))
[(‘red’, ‘cherry’, 1kg’), (‘yellow’, ‘banana’, 12), (‘orange’, ‘orange’, 5kg’)]
```

The function **count** is used to find the number of occurrences of a value in a tuple, for example:

```
>>> age =(20, 18, 17, 18, 19, 18)
>>> age.count(18)
3
```

The function **index** is used to find the index of the first occurrence of a particular element in a tuple,

```
>>> age.index(18)
1
```

DICTIONARY

Unlike lists, tuples and strings, a dictionary is an unordered sequence of *key-value* pairs. Indices in a dictionary can be of any immutable type and are called keys. Enclosed within curly brackets.

```
>>> month = {}
>>> month[1] = 'Jan'
>>> month[2] = 'Feb'
>>> month[3] = 'Mar'
>>> month[4] = 'Apr'
>>> month
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}
>>> type(month)
<class 'dict'>
```

```
>>> price = {'tomato':40, 'cucumber':30, 'potato':20, 'cauliflower':70, 'cabbage':50, 'lettuce':40, 'raddish':30, 'carrot':20, 'peas':80}
>>> price['price']
20
>>> price.keys()
dict_keys(['tomato', 'cucumber', 'potato', 'cauliflower', 'cabbage', 'lettuce', 'raddish', 'carrot', 'peas'])

>>> price.values()

dict_values([40, 30, 20, 70, 50, 40, 30, 20, 80])

>>> price.items()

dict_items([('tomato':40), ('cucumber':30), ('potato':20), ('cauliflower':70), ('cabbage':50), ('lettuce':40), ('raddish':30), ('carrot':20), ('peas':80)]}
```

Note that search in a dictionary is based on keys. Therefore, in a dictionary, the keys are required to be unique. Tho same value may be assigned to different keys. **As keys in a dictionary are immutable, lists cannot be used as keys but values associated with keys can be mutable objects and thus, maybe changed at will.**

```
>>> price['tomato']=25
```

Keys maybe of heterogeneous types,

```
>>> counting = {1:'one', 'one':1, 2:'two', 'two':2}
```

<u>Dictionary</u> <u>Operations</u>

```
>>> digits = {0: 'Zero', 1:'One',  2:'Two', 3:'Three', 4:'Four', 5:'Five', 6:'Six', 7:'Seven', 8:'Eight', 9: 'Nine'}

>>> len(digits)

10

>>> digits[1]

'One'

>>> min(digits)

0

>>> max(digits)
```

```
9
>>> sum(digits)
45
>>> 5 in digits
True
>>> ‘Five’ in digits
False
```

Deletion

del operator is used to remove a *key-value* pair from a dictionary.

```
>>> del digits[0]
>>> digits
{1:‘One’,  2:‘Two’, 3:‘Three’, 4:‘Four’, 5:‘Five’, 6:‘Six’, 7:‘Seven’, 8:‘Eight’, 9: ‘Nine’}
```

To delete a dictionary

```
>>> del digits
```

To remove all *key-value* pairs in a dictionary, the **clear** function is used. Note while assigning an empty dictionary **{}** creates a new object, the function **clear** removes all *key-value* pairs from an existing dictionary.

```
>>> nums = digits
>>> nums.clear()
>>> nums, digits
({}, {})
```

Since **digits** and **nums** refer to the same dictionary object, on applying **clear** function to nums, the dictionary **digits** also becomes empty.

Functions get, update, copy

The **get** function is used to extract the value corresponding to a given key:

```
>>> digits.get(1, -1)
‘One’
>>> digits.get(10)
None
```

The first parameter specifies the key and the second parameter is used to specify the value to be returned in case the key is not found in the dictionary. In case second parameter isn’t specified the system returns **None.**

The **update** function is used to insert in a dictionary, all key-value pairs of another dictionary.

```
>>> moreDigits = {10:‘Ten’,  11:‘Eleven’, 13:‘Thirteen’}
>>> digits.update(moreDigits)
>>> digits
```

```
{1:‘One’,  2:‘Two’, 3:‘Three’, 4:‘Four’, 5:‘Five’, 6:‘Six’, 7:‘Seven’, 8:‘Eight’, 9: ‘Nine’, 10:‘Ten’,
11:‘Eleven’, 13:‘Thirteen’}
```

The **copy** function is used to create a shallow copy of a dictionary object.

```
>>> newDigits=digits.copy()
>>> id(newDigits), id(digits)
(2195411280576, 2195411280384)
```