8 Tuples

In This Chapter

8.1 INTRODUCTION

The Python tuples are sequences that are used to store a tuple of values of any type. You have learnt in earlier chapters that Python **tuples are immutable** *i.e.*, you cannot change the elements of a tuple *in place* ; Python will create a fresh tuple when you make changes to an element of a tuple. Tuple is a type of sequence like *strings* and *lists* but it differs from them in the way that lists are *mutable* but strings and tuples are *immutable*.

This chapter is dedicated to basic tuple manipulation in Python. We shall be talking about creating and accessing tuples, various tuple operations and tuple manipulations through some built-in functions.

3.2 CREATING AND ACCESSING TUPLES

A tuple is a standard data type of Python that can store a sequence of values belonging to any type. The *Tuples* are depicted through parentheses *i.e.*, round brackets, *e.g.*, following are some tuples in Python :

```
()                          # tuple with no member, empty tuple
(1, 2, 3)                   # tuple of integers
(1, 2.5, 3.7, 9)            # tuple of numbers (integers and floating point)
('a', 'b', 'c')             # tuple of characters
('a', 1, 'b', 3.5, 'zero')  # tuple of mixed value types
('One', 'Two', 'Three')     # tuple of strings
```

NOTE

Tuples are ***immutable sequences*** of Python *i.e.*, you cannot change elements of a tuple in place.

Before we proceed and discuss how to create tuples, one thing that must be clear is that **Tuples are immutable** (*i.e.*, non-modifiable) *i.e.*, you cannot change elements of a tuple in place.

3.2.1 Creating Tuples

Creating a tuple is similar to list creation, but here you need to put a number of expressions in parentheses. That is, use round brackets to indicate the *start* and *end* of the tuple, and separate the items by commas. For example :

```
(2, 4, 6)
('abc', 'def')
(1, 2.0, 3, 4.0)
()
```

Thus to create a tuple you can write in the form given below :

```
T = ()
T = (value, ...)
```

This construct is known as a **tuple display construct.**

Consider some more examples :

The Empty Tuple

The empty tuple is ( ). It is the tuple equivalent of 0 or ''. You can also create an empty tuple as :

```
T = tuple()
```

It will generate an empty tuple and name that tuple as *T*.

Single Element Tuple

Making a tuple with a single element is tricky because if you just give a single element in round brackets, Python considers it a value only, *e.g.*,

```
>>> t = (1)
>>> t
1
```

(1) was treated as an integer expression, hence t stores an integer 1, not a tuple

To construct a tuple with one element just add a comma after the single element as shown below :

```
>>> t = 3,
>>> t
(3,)
>>> t2 = (4,)
>>> t2
(4,)
```

Both these ways will create tuples

NOTE

Tuples are formed by placing a comma-separated tuple of expressions in parentheses.

3. Long tuples

If a tuple contains many elements, then to enter such long tuples, you can split it across several lines, as given below :

```
sqrs = ( 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
         256, 289, 324, 361, 400, 441, 484, 529, 576, 625 )
```

Notice the opening parenthesis and closing parenthesis appear just in the beginning and end of the tuple.

4. Nested tuples

If a tuple contains an element which is a tuple itself then it is called nested tuple *e.g.*, following is a nested tuple :

```
t1 = (1, 2, (3, 4))
```

The tuple t1 has three elements in it : 1, 2 and (3, 4). The third element of tuple t1 is a tuple itself, hence, t1 is a nested tuple.

Creating Tuples from Existing Sequences

You can also use the built-in tuple type object (*tuple( )*) to create tuples from sequences as per the syntax given below :

```
T = tuple(<sequence>)
```

where ***<sequence>*** can be any kind of sequence object including strings, lists and tuples.

Python creates the individual elements of the tuple from the individual elements of passed sequence. If you pass in another tuple, the tuple function makes a copy.

Consider following examples :

```
>>> t1 = tuple('hello')
>>> t1
('h', 'e', 'l', 'l', 'o')
```

Tuple t1 is created from another sequence - a string "hello"
It generated individual elements from the individual

```
>>> L = ['w', 'e', 'r' , 't', 'y']
>>> t2 = tuple(L)
>>> t2
('w', 'e', 'r', 't', 'y')
```

Tuple t2 is created from another sequence - a list L
It generated individual elements from the individual elements of the passed list L

You can use this method of creating tuples of single characters or single digits via keyboard input. Consider the code below :

```
t1 = tuple( input('Enter tuple elements:'))
Enter tuple elements : 234567
>>> t1
('2', '3', '4', '5', '6', '7')
```

See, it created the elements of tuple t1 using each of the character inputs

See, with tuple( ) around input( ), even if you not put parenthesis, it will create a tuple using individual characters as elements. But most commonly used method to input tuples is **eval(input())** as shown below :

```
tuple = eval(input("Enter tuple to be added:"))
print ("Tuple you entered :", tuple)
```

when you execute it, it will work somewhat like :

```
Enter tuple to be added: (2, 4, "a", "hjkjl", [3,4])
Tuple you entered : (2, 4, "a", "hjkjl", [3,4])
```

If you are inputting a tuple with **eval( )**, then make sure to enclose the tuple elements in parenthesis.

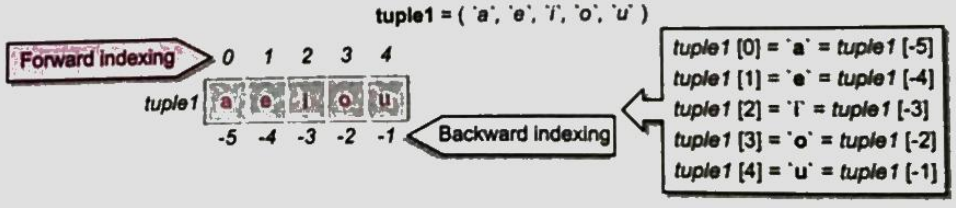
Please note sometimes (not always) **eval( )** does not work in Python shell. At that time, you can run it through a script too.

8.2.2 Accessing Tuples

Tuples are immutable (non-editable) sequences having a progression of elements. Thus, other than editing items, you can do all that you can do with lists. Thus like lists, you can access its individual elements. Before we talk about that, let us learn about how elements are indexed in tuples.

Similarity with Lists

Tuples are very much similar to lists except for the mutability. In other words, *Tuples* are immutable counter-parts of lists. Thus, like lists, tuple-elements are also indexed, *i.e.*, forward indexing as 0, 1, 2, 3,... and backward indexing as –1, –2, –3,... [See Fig 8.1(*a*)]



(*a*) Tuple Elements' two way indexing

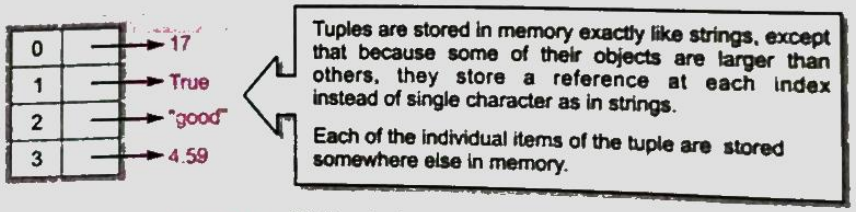


Figure 8.1 (*b*) How tuples are internally organized

Thus, you can access the tuple elements just like you access a list's or a string's elements *e.g.*, **Tuple[i]** will give you element at i^{th} index ; **Tuple[a:b]** will give you elements between indexes *a* to *b-1* and so on. Put in other words, tuples are similar to lists in following ways :

- **Length.** Function **len(T)** returns the number of items (count) in the tuple *T*.
- **Indexing and Slicing**

  T[*i*] returns the item at index *i* (the first item has index 0),

  T[*i* : *j*] returns a new tuple, containing the objects between *i* and *j* excluding index *j*,

  T[*i* : *j* : *n*] returns a new tuple containing every *n*th item from index *i* to *j*, excluding index *j* . Just like lists.
- **Membership operators.** Both **'in'** and **'not in'** operators work on *Tuples* just like they work for other sequences. That is, **in** tells if an element is present in the tuple or not and **not in** does the opposite.
- **Concatenation and Replication operators + and *.** The + operator adds one tuple to the end of another. The * operator repeats a tuple. We shall be talking about these two operations in a later section 8.3 – *Tuple Operations*.

Accessing Individual Elements

As mentioned, the individual elements of a tuple are accessed through their indexes given in square brackets. Consider the following examples :

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels[0]
'a'
>>> vowels[4]
'u'
>>> vowels[-1]
'u'
>>> vowels[-5]
'a'
```

NOTE

While accessing tuple elements, if you pass in a negative index, Python adds the length of the tuple to the index to get element's forward index.

Recall that like strings, if you pass in a negative index, Python adds the length of the tuple to the index to get its forward index. That is, for a 6-element tuple *T*, *T*[−5] will be internally computed as :

$T[-5+6] = T[1]$, and so on.

Difference from Lists

Although tuples are similar to lists in many ways, yet there is an important difference in mutability of the two. Tuples are not mutable, while lists are. You cannot change individual elements of a tuple in place, but lists allow you to do so.

That is, following statement is fully valid for lists (BUT not for tuples). That is, if we have a list L and a tuple T, then

```
L[i] = element
```

is VALID for Lists. BUT

```
T[i] = element
```

is INVALID as you cannot perform item-assignment in immutable types.

NOTE

Tuples are similar to lists in many ways like indexing, slicing and accessing individual elements but they are different in the sense that **tuples are immutable while lists are not.**

Traversing a Tuple

Recall that traversal of a sequence means accessing and processing each element of it. Thus traversing a tuple also means the same and same is the tool for it, i.e., the Python loops. The **for loop** makes it easy to **traverse** or **loop over** the items in a tuple, as per following syntax :

```
for <item> in <Tuple> :
    process each item here
```

For example, following loop shows each item of a tuple T in separate lines :

```
T = ('P', 'y', 't', 'h', 'o', 'n')
for a in T :
    print (T[a])
```

The above loop will produce result as :

```
P
y
t
h
o
n
```

How it works

The loop variable a in above loop will be assigned the Tuple elements, one at a time. So, loop-variable a will be assigned 'P' in first iteration and hence 'P' will be printed; in second iteration, a will get element 'y' and 'Y' will be printed ; and so on.

If you only need to use the indexes of elements to access them, you can use functions ***range( )*** and ***len( )*** as per following syntax :

```
for index in range(len(T)):
    process Tuple[index] here
```

Consider program 8.1 that traverses through a tuple using above format and prints each item of a tuple L in separate lines along with its index.

Program 8.1 *Program to print elements of a tuple ('Hello', "Isn't", 'Python', 'fun', '?') in separate lines along with element's both indexes (positive and negative).*

```
T = ('Hello', "Isn't", 'Python', 'fun', '?' )
length = len(T)
for a in range(length) :
    print ('At indexes', a, 'and ', (a - length), 'element :', T[a])
```

```
At indexes 0 and -5 element : Hello
At indexes 1 and -4 element : Isn't
At indexes 2 and -3 element : Python
At indexes 3 and -2 element : fun
At indexes 4 and -1 element : ?
```

Now that you know about tuple traversal, let us talk about tuple operations.

8.3 TUPLE OPERATIONS

The most common operations that you perform with tuple include joining tuples and slicing tuples. In this section, we are going to talk about the same.

8.3.1 Joining Tuples

Joining two tuples is very easy just like you perform addition, literally ;) . The + operator, the concatenation operator, when used with two tuples, joins two tuples.

Consider the example given below :

```
>>> tpl1 = ( 1, 3, 5)
>>> tpl2 = (6, 7, 8 )
>>> tpl1 + tpl2
(1, 3, 5, 6, 7, 8)
```

The + operator concatenates two tuples and creates a new tuple

As you can see that the resultant tuple has firstly elements of first tuple ***lst1*** and followed by elements of second tuple ***lst2***. You can also join two or more tuples to form a new tuple, *e.g.*,

```
>>> tpl1 = (10, 12, 14)
>>> tpl2 = (20, 22, 24)
>>> tpl3 = (30, 32, 34)
>>> tpl = tpl1 + tpl2 + tpl3
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
```

The + operator is used to concatenate three individual tuples to get a new combined tuple tpl.

The + operator when used with tuples requires that **both the operands must be of tuple types.** You cannot add a number or any other value to a tuple. For example, following expressions will result into error :

```
tuple + number
tuple + complex-number
tuple + string
tuple + list
```

Consider the following examples :

```
>>> tpl1 = (10, 12, 14)
>>> tpl1 + 2
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    tpl1+2
TypeError: can only concatenate tuple (not "int") to tuple
>>> tpl1 + "abc"
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    tpl1 + "abc"
TypeError: can only concatenate tuple (not "str") to tuple
```

See errors generated when anything other than a tuple is added to a tuple

IMPORTANT

Sometimes you need to concatenate a tuple (say tpl) with another tuple containing only one element. In that case, if you write statement like :

```
>>> tpl + (3)
```

Python will return an error like :

```
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    tuple + (3)
TypeError: can only concatenate tuple (not "int") to tuple
```

NOTE

A single value in ( ) is treated as single value not as tuple. That is, expressions (3) and ('a') are integer and string respectively but (3,) and ('a',) are examples of tuples with single element.

The reason for above error is that a number enclosed in ( ) is considered number only. To make it a tuple with just one element, just add a comma after the only element, *i.e.*, make it (3,). Now Python won't return any error and successfully concatenate the two tuples.

```
>>> tpl =(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl + (3, )
(10, 12, 14, 20, 22, 24, 30, 32, 34, 3)
```

Single element tuple

Repeating or Replicating Tuples

Like strings and lists, you can use * operator to replicate a tuple specified number of times, *e.g.*, if **tpl1** is (1, 3, 5), then

```
>>> tpl1 * 3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

*The * operator repeats a tuple specifies number of times and creates a new tuple*

NOTE

When used with tuples, the * operator requires both the operands as tuple-types; and the * operator requires a tuple and an integer as operands.

8.3.2 Slicing the Tuples

Tuple slices, like list-slices or string slices are the sub parts of the tuple extracted out. You can use indexes of tuple elements to create ***tuple slices*** as per following format :

```
seq = T[start:stop]
```

The above statement will create a *tuple slice* namely **seq** having elements of tuple T on indexes *start, start+1, start+2, ..., stop-1*. Recall that index or. last limit is not included in the **tuple slice**. The *tuple slice* is a **tuple** in itself that is you can perform all operations on it just like you perform on tuples.

Consider the following example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> seq = tpl [ 3:-3]
>>> seq
(20, 22, 24)
```

For normal indexing, if the resulting index is outside the tuple, Python raises an **IndexError exception**. Slices are treated as boundaries instead, and the result will simply contain all items between the boundaries. For the *start* and *stop* given beyond tuple limits in a tuple slice, Python simply returns the elements that fall between specified boundaries, if any.

NOTE

T[start:stop] creates a tuple slice out of tuple T with elements having between indexes *start* and *stop*, not including *stop*

For example, consider the following :

```
>>> tpl =(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl [3:30]
(20, 22, 24, 30, 32, 34)
>>> tpl [-15 :7]
(10, 12, 14, 20, 22, 24, 30)
```

Giving upper limit way beyond the size of the tuple, but Python return elements from tuple falling in range 3

Giving lower limit much lower, but python returns elements from tuple falling in range – 15 onwards < 7

Tuples also support slice steps too. That is, if you want to extract, not consecutive but every other element of the tuple, there is a way out – the *slice steps*. The *slice steps* are used as per following format :

```
seq = T[start:stop:step]
```

Consider some examples to understand this.

```
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl[0 : 10 : 2 ]
(10, 14, 22, 30, 34)
>>> tpl[2 : 10 : 3 ]
(14, 24, 34)
>>> tpl[:: 3 ]
(10, 20, 30)
```

Include every 2nd element, i.e., skip 1 element in between. Check resulting tuple slice

Include every 3rd element, i.e., skip 2 element in between

No start and stop given. Only step is given as 3. That is, from the entire tuple, pick every 3rd element for the tuple

Consider some more examples :

```
seq = T[::2]      # get every other item, starting with the first
seq = T[5::2]     # get every other item, starting with the
                  # sixth element, i.e., index 5
```

You can use the + and * operators with tuple slices too. For example, if a tuple namely Tp has values as (2, 4, 5, 7, 8, 9, 11, 12, 34), then :

```
>>> Tp[2:5] * 3
(5, 7, 8, 5, 7, 8, 5, 7, 8)
>>> Tp[2:5] + (3, 4)
(5, 7, 8, 3, 4)
```

*See, the * operation has multiplies the tuple slice and not the full tuple*

Here, the + operator has added the given tuple to the tuple slice

NOTE

T[start : stop: step] creates a tuple slice out of tuple T with elements having between indexes *start* and *stop*, not including *stop*, skipping step-1 elements in between

Comparing Tuples

You can compare two tuples without having to write code with loops for it. For comparing two tuples, you can using comparison operators, *i.e.*, <, >, ==, != etc. For comparison purposes, Python internally compares individual elements of two tuples, applying all the comparison rules that you have read earlier. Consider following code :

```
>>> a = (2, 3)
>>> b = (2, 3)
>>> a == b
True
```

*For two tuples to be equal, they must have same number of elements and matching values (recall **int** and **float** with matching values are considered equal)*

```
>>> c = ('2', '3')
>>> a == c
False

>>> a > b
False

>>> d = (2.0, 3.0)
>>> d > a
False
>>> d == a
True
```

Notice for comparison purposes, Python ignored the types of elements and compared values only

```
>>> e = (2, 3, 4)
>>> a < e
True
```

You can refer to table 7.1 that discusses non-equality comparisons of two sequences. Elements in tuples are also compared on similar lines.

Unpacking Tuples

Creating a tuple from a set of values is called **packing** and its reverse, *i.e.*, creating individual values from a tuple's elements is called **unpacking**.

Unpacking is done as per syntax :

```
<variable1>, <variable2>, <variable3>, ... = t
```

where the number of variables in the left side of assignment must match the number of elements in the tuple.

For example, if we have a tuple as :

```
t = (1, 2, 'A', 'B')
```

The length of above tuple t is 4 as there are four elements in it. Now to unpack it, we can write :

```
w, x, y, z = t
```

You may even enclosed the variables on LHS in parenthesis. It will give you same result.

NOTE

Forming a tuple from individual values is called packing and creating individual values from a tuple's elements is called unpacking.

Python will now assign each of the elements of tuple *t* to the variables on the left side of assignment operator. That is, you can now individually print the values of these variables, somewhat like :

```
print (w)
print (x)
print (y)
print (z)
```

The above code will yield the result as :

```
1
2
'A'
'B'
```

As per *Guido van Rossom*, the creator of Python language —

> *"Tuple unpacking requires that the list of variables on the left has the same number of elements as the length of the tuple."*

Deleting Tuples

The **del** statement of Python is used to delete elements and objects but as you know that tuples are immutable, which also means that individual elements of a tuple cannot be deleted, *i.e.*, if you give a code like :

```
>>> del t1[2]
```

Then Python will give you a message like :

```
Traceback (most recent call last):
  File "<ipython-input-83-cad7b2ca8ce3>", line 1, in <module>
    del t1[2]
TypeError: 'tuple' object doesn't support item deletion
```

But you can delete a complete tuple with **del** statement as :

```
del <tuple_name>
```

For example,

```
>>> t1 = (5, 7, 3, 9, 12)
>>> t1
(5, 7, 3, 9, 12)
>>> del t1
    print(t1)
. .ceback (most recent call last):
  File "<ipython-input-89-04f730070f35>," line 1, in <module>
    print(t1)
NameError: name 't1' is not defined
```

*See, after using **del** statement on a tuple, if you try to access/ print it, Python gives error as the tuple has been deleted and this objects exists no more*

8.4 TUPLE FUNCTIONS AND METHODS

Python also offers many built-in functions and methods for tuple manipulation. You have already worked with one such method **len( )** in earlier chapters. In this section, you will learn about many other built-in powerful tuple methods of Python used for tuple manipulation. Let us now have a look at some useful built-in tuple manipulation methods.

1. The len( ) method

This method returns length of the tuple, *i.e.*, the count of elements in the *tuple*.

Syntax :

```
len(<tuple>)
```

– *Takes tuple name as argument and returns an integer.*

Example :

```
>>> employee = ('John', 10000, 23, 'Sales')
>>> len(employee)
4
```

← *The len( ) returns the count of elements in the tuple*

2. The max( ) method

This method returns the element from the tuple having **maximum value**.

Syntax :

```
max(<tuple>)
```

– *Takes tuple name as argument and returns an object (the element with maximum value).*

Example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> max(tpl)
34
```

← *Maximum value from tuple* ***tpl*** *is returned*

```
>>> tpl2 = ("Karan", "Zubin", "Zara", "Ana")
>>> max(tpl2)
'Zubin'
```

← *Maximum value from tuple* ***tpl2*** *is returned*

You can use max( ) on lists too to find maximum element from a list. Please note that **max( )** applied on sequences like tuples/lists etc. will return a maximum value ONLY IF the sequence contains values of same type. If your tuple (or list) contains values of different datatypes then, it will give you an error stating that mixed type comparison is not possible :

(*a*)
```
>>> ab = (1, 2.5, "1", [3,4], (3,4) )
>>> max(ab)
Traceback (most recent call last):
  File "<ipython-input-54-fd79f7408dd4>", line 1, in <module>
    max(ab)
TypeError: '>' not supported between instances of 'str' and 'float'
```

(b)
```
>>> ab = ([3,4], (3,4) )
>>> max(ab)
Traceback (most recent call last) :
  File "<ipython-input-63-fd79f7408dd4>", line 1, in <module>
    max(ab)
TypeError: '>' not supported between instances of 'tuple' and 'list'
```

3. The min( ) method

This method returns the element from the tuple having **minimum value**.

Syntax :

```
min(<tuple>)
```

– *Takes tuple name as argument and returns an object (the element with minimum value)*

Example :

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> min(tpl)
10
```

← *Maximum value from tuple **tpl** is returned*

```
>>> tpl2 = ("Karan", "Zubin","Zara", "Ana")
>>> min(tpl2)
'Ana'
```

← *Maximum value from tuple **tpl2** is returned*

Like max( ), for min( ) to work, the elements of tuple should be of same type.

4.The index( ) method

The index( ) works with tuples in the same way it works with lists. That is, it returns the index of an existing element of a tuple.

It is used as :

```
<tuplename> . index (<item>)
```

Example :

```
>>> t1 = [3, 4, 5, 6.0]
>>> t1.index(5)
2
```

But if the given item does not exist in tuple, it raises **ValueError** exception.

5. The count( ) function

The count( ) method returns the count of a member element/object in a given sequence (list/tuple). You can use the count( ) function as per following syntax :

```
<sequence name>.count(<object>)
```

Example :

```
>>> t1 = (2, 4, 2, 5, 7, 4, 8, 9, 9, 11, 7, 2)
>>> t1.count(2)
3
>>> t1.count(7)
2
>>> t1.count(11)
1
```

There are 3 occurrence of element 2 in given tuple , hence count() return 3 here

For an element not in tuple, it returns 0 (zero).

6. The tuple( ) method

This method is actually constructor method that can be used to create tuples from different types of values.

Syntax :

```
tuple(<sequence>)
```

- *Takes an optional argument of sequence type ; Returns a tuple.*
- *With no argument, it returns empty tuple*

Example :

- *Creating empty tuple*

```
>>> tuple()
()
```

- *Creating tuple from a string*

```
>>> t = tuple("abc")
>>> t
('a', 'b', 'c')
```

- *Creating a tuple from a list*

```
>>> t = tuple([1,2,3])
>>> t
(1, 2, 3)
```

- *Creating a tuple from keys of a dictionary*

```
>>> t1 = tuple ( {1:"1", 2:"2"})
>>> t1
(1, 2)
```

As you can notice that, Python has considered only the keys of the passed dictionary to create tuple. But one thing that you must ensure is that the **tuple( )** can receive argument of sequence types only, *i.e.*, either a string or a list or even a dictionary. Any other type of value will lead to an error. See below :

```
>>> t = tuple(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t = tuple(1)
TypeError: 'int' object is not iterable
```

NOTE

With tuple( ), the argument must be a sequence type *i.e.*, a string or a list or a dictionary.

The *tuple( )* and *list( )* are constructors that let you create tuples and lists respectively from passed sequence. This feature can be exploited as a trick to modify tuple, which otherwise is not possible. Following information box talks about the same.

Check Point 8.1

1. Why are tuples called immutable types ?
2. What are mutable counterparts of tuples ?
3. What are different ways of creating tuples ?
4. What values can we have in a tuple ? Do they all have to be the same type* ?
5. How are individual elements of tuples accessed ?
6. How do you create the following tuples ?
   (a) `(4, 5, 6)`
   (b) `(-2, 1, 3)`
   (c) `(-9, -8, -7, -6, -5)`
   (d) `(-9, -10, -11, -12)`
   (e) `(0, 1, 2)`
7. If `a = (5, 4, 3, 2, 1, 0)` evaluate the following expressions :
   (a) `a[0]`
   (b) `a[1]`
   (c) `a[a[0]]`
   (d) `a[a[-1]]`
   (e) `a[a[a[a[2]+1]]]`
8. Can you change an element of a sequence ? What if the sequence is a dictionary ? What if the sequence is a tuple ?
9. What does `a + b` amount to if **a** and **b** are tuples ?
10. What does `a * b` amount to if **a** and **b** are tuples ?
11. What does `a + b` amount to if **a** is a tuple and `b = 5` ?
12. Is a string the same as a tuple of characters ?
13. Can you have an integer, a string, a tuple of integers and a tuple of strings in a tuple ?

Ind

Tuples are immutable counterparts of lists. If you need to modify the contents often of a sequence of mixed types, then you should go for lists only. However, if you want to ensure that a sequence of mixed types is not accidently changed or modified, you go for tuples. But sometimes you need to retain the sequence as tuple and still need to modify the contents at one point of time, then you can use one of the two methods given here.

(a) Using Tuple Unpacking

Tuples are immutable. To change a tuple, we would need to first unpack it, change the values, and then again repack it :

```
tpl = (11, 33, 66, 99)
```

1. First unpack the tuple :
   ```
   a,b,c,d = tpl
   ```
2. Redefine or change desired variable say, **c**
   ```
   c = 77
   ```
3. Now repack the tuple with changed value
   ```
   tpl = (a, b, c, d)
   ```

(b) Using the constructor functions of lists and tuples i.e., list( ) and tuple( )

There is another way of doing the same as explained below :

```
>>> tpl = ("Anand", 35000, 35, "Admin")
>>> tpl
('Anand', 35000, 35, 'Admin')
```

1. Convert the tuple to list using list() :
   ```
   >>> lst = list(tpl)
   >>> lst
   ['Anand', 35000, 35, 'Admin']
   ```
2. Make changes in the desired element in the list
   ```
   >>> lst[1] = 45000
   >>> lst
   ['Anand', 45000, 35, 'Admin']
   ```
3. Create a tuple from the modified list with tuple()
   ```
   >>> tpl = tuple(lst)
   >>> tpl
   ['Anand', 45000, 35, 'Admin']
   ```

Isn't the trick simple? ☺

PriP TUPLES HANDLING AND MANIPULATION — Progress In Python 8.1

This PriP session aims at strengthening skills related to Tuples handling and manipulation.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 8.1 under Chapter 8 after practically doing it on the computer.

>>>❖<<<

With this we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

LET US REVISE

- *Tuples are **immutable sequences** of Python i.e., you cannot change elements of a tuple in place.*
- *To create a tuple, put a number of comma-separated expressions in round brackets.*
- *The empty round brackets i.e., ( ) indicate an empty tuple.*
- *Tuples index their elements just like strings or lists, i.e., two way indexing.*
- *Tuples are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.*
- *Tuples are similar to strings in many ways like indexing, slicing and accessing individual elements and they are immutable just like strings are.*
- *Function len(L) returns the number of items (count) in the tuple L.*
- *Membership operator **in** tells if an element is present in the tuple or not and **not in** does the opposite.*
- *The + operator adds one tuple to the end of another. The * operator repeats a tuple.*
- *Tuple slice is an extracted part of tuple; tuple slice is a tuple in itself.*
- *T[start:stop] creates a tuple slice out of tuple T with elements falling between indexes start and stop, not including stop.*
- *Common tuple manipulation functions are : len( ), max( ), min( ), and tuple( ).*

Solved Problems

1. *How are tuples different from lists when both are sequences ?*

   Solution. The tuples and lists are different in following ways :

   - The tuples are immutable sequences while lists are mutable.
   - Lists can grow or shrink while tuples cannot.

2. *How can you say that a tuple is an ordered list of objects ?*

Solution. A tuple is an ordered list of objects. This is evidenced by the fact that the objects can be accessed through the use of an ordinal index and for a given index, same element is returned everytime.

3. *Following code is trying to create a tuple with a single item. But when we try to obtain the length of the tuple is, Python gives error. Why? What is the solution ?*

```
>>> t = (6)
>>> len(t)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    len(t)
TypeError: object of type 'int' has no len()
```

Solution. The syntax for a tuple with a single item requires the item to be followed by a comma as shown below :

```
t = ("a",)
```

Thus, above code is not creating a tuple in t but an integer, on which *len( )* cannot be applied. To create a tuple in t with single element, the code should be modified as :

```
>>> t = (6,)
>>> len(t)
```

4. *What is the length of the tuple shown below ?*

```
t = ((('a', 1), 'b', 'c'), 'd', 2), 'e', 3)
```

Solution. The length of this tuple is 3 because there are just three elements in the given tuple. Because a careful look at the given tuple yields that tuple *t* is made up of :

```
t1 = "a", 1
t2 = t1, "b", "c"
t3 = t2, "d", 2
t = ( t3,"e", 3)
```

5. *Can tuples be nested.?*

Solution. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

6. *Given a tuple namely cars storing car names as elements :*

```
('Toyota', 'Honda', 'GM', 'Ford', 'BMW', 'Volkswagon', 'Mercedes', 'Ferrai', 'Porshe')
```

Write a program to print names of the cars in the index range 2 to 6, both inclusive.

The output should also include the index in words as shown below :

```
One     Honda
Two     GM
:
```

Solution.

```
cars = ('Toyota','Honda','GM','Ford','BMW','Volkswagon','Mercedes','Ferrari','Porshe')
Number = ("Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine")
for x in range(2,7):
  print (Number[x], cars[x])
```

GLOSSARY

Tuple	An immutable sequence of elements.
Packing	Creating a tuple from individual values
Unpacking	Creating individual values from a tuple's elements.
Tuple Slice	A tuple created from another tuple containing the requested elements.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. Discuss the utility and significance of Tuples, briefly.
2. If a is (1, 2, 3)
   (*a*) what is the difference (if any) between a * 3 and (a, a, a) ?
   (*b*) is a * 3 equivalent to a + a + a ?
   (*c*) what is the meaning of a[1:1] ?
   (*d*) what is the difference between a[1:2] and a[1:1] ?
3. Does the slice operator always produce a new tuple ?
4. The syntax for a tuple with a single item is simply the element enclosed in a pair of matching parentheses as shown below :
   ```
   t = ("a")
   ```
   Is the above statement true? Why? Why not ?
5. Are the following two assignments same ? Why / why not ?
   (*a*)
   ```
   T1 = 3, 4, 5
   T2 = (3,4 ,5)
   ```
   (*b*)
   ```
   T3 = (3, 4, 5)
   T4 = ((3, 4, 5))
   ```
6. What would following statements print ? Given that we have tuple = ('t', 'p', 'l')
   (*a*) `print("tuple")` (*b*) `print(tuple("tuple"))` (*c*) `print(tuple)`
7. How is an empty tuple created ?
8. How is a tuple containing just one element created ?
9. How can you add an extra element to a tuple ?
10. When would you prefer tuples over lists ?
11. What is the difference between (30) and (30,) ?

Type B : Application Based Questions

1. Find the output generated by following code fragments :
   (*a*)
   ```
   plane = ("Passengers", "Luggage")
   plane[1] = "Snakes"
   ```
   (*b*) `(a, b, c) = (1, 2, 3)`
   (*c*) `(a, b, c, d) = (1, 2, 3)`
   (*d*) `a, b, c, d = (1, 2, 3)`
   (*e*) `a, b, c, d, e = (p, q, r, s, t) = t1`

(f) What will be the values and types of variables *a, b, c, d, e, f, p, q, r, s, t* after executing part *e* above ?

(g)
```
t2 = ('a')
type(t2)
```

(h)
```
t3 = ('a', )
type(t3)
```

(i)
```
T4 = (17)
type(T4)
```

(j)
```
T5 = (17,)
type(T4)
```

(k)
```
tuple = ('a', 'b', 'c', 'd', 'e')
tuple = ('A',) + tuple[1:]
print(tuple)
```

(l)
```
t2 = (4, 5, 6)
t3 = (6, 7)
t4 = t3 + t2
t5 = t2 + t3
print(t4)
print(t5)
```

(m)
```
t3 = (6, 7)
t4 = t3*3
t5 = t3 * (3)
print(t4)
print(t5)
```

(n)
```
1 = (3, 4)
t2 = ('3', '4')
print(t1 + t2 )
```

(o) What will be stored in variables a, b, c, d, e, f, g, h, after following statements ?

```
perc=(88,85,80,88,83,86)
a = perc[2:2]
b = perc[2:]
c = perc[:2]
d = perc[:-2]
e = perc[-2:]
f = perc[2:-2]
g = perc[-2:2]
h = perc[:]
```

2. What does each of the following expressions evaluate to? Suppose that T is the tuple

```
("These", ["are", "a", "few", "words"], "that", "we", "will", "use")
```

(a) `T[1][0::2]` (b) `"a" in T[1][0]` (c) `T[:1] + T[1]`

(d) `T[2::2]` (e) `T[2][2] in T[1]`

3. Carefully read the given code fragments and **figure out the errors** that the code may produce.

(a)
```
t = ('a', 'b', 'c', 'd', 'e')
print(t[5])
```

(b)
```
t = ('a', 'b', 'c', 'd', 'e')
t[0] = 'A'
```

(c)
```
t1 = (3)
t2 = (4, 5, 6)
t3 = t1 + t2
print(t3)
```

(d)
```
t2 = (4, 5, 6)
t3 = (6, 7)
print( t3 - t2 )
```

(e)
```
t3 = (6, 7)
t4 = t3*3
t5 = t3 * (3)
t6 = t3 * (3,)
print(t4)
print(t5)
print(t6)
```

(f)
```
t = ('a', 'b', 'c', 'd', 'e')
1, 2, 3, 4, 5, = t
```

(g)
```
t = ('a', 'b', 'c', 'd', 'e')
1n, 2n, 3n, 4n, 5n = t
```

(h)
```
t = ('a', 'b', 'c', 'd', 'e')
x, y, z, a, b = t
```

(i)
```
t = ('a', 'b', 'c', 'd', 'e')
a, b, c, d, e, f = t
```

4. What would be the output of following code if ntpl = ("Hello", "Nita", "How's", "life?")

```
(a, b, c, d) = ntpl
print ("a is:", a)
print ("b is:", b)
print ("c is:", c)
print ("d is:", d)
ntpl = (a, b, c, d)
print (ntpl[0][0]+ntpl[1][1], ntpl[1])
```

5. Predict the output.

```
tuple_a = 'a', 'b'
tuple_b = ('a', 'b')
print (tuple_a == tuple_b)
```

6. Predict the output.

```
tuple1 = ('Python') * 3
print(type(tuple1))
```

7. Find the error. Following code intends to create a tuple with three identical strings. But even after successfully executing following code (No error reported by Python), The len() returns a value different from 3. Why ?

```
tup1 = ('Mega') * 3
print(len(tup1))
```

8. What will the following code produce ?

```
Tup1 = (1,) * 3
Tup1 [0] = 2
print(Tup1)
```

9. What will be the output of the following code snippet ?

```
Tup1 = ((1, 2),) * 7
print(len(Tup1 [3:8]))
```

10. Predict the output

```
x = (1, (2, (3, (4,))))
print(len(x) )
print( x[1][0] )
print( 2 in x )
y = (1, (2, (3,), 4), 5)
print( len(y) )
print( len(y[1]) )
print( y[2] = 50 )
z = (2, (1, (2,), 1), 1)
print( z[z[z[0]]] )
```