TUPLE

TUPLES - IMMUTABLE SEQUENCES

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

- A tuple is an ordered sequence of zero or more object references.

- Tuples support the same slicing and striding syntax as slicing and striding strings. This makes it easy to extract items from a tuple.

- Like strings, tuples are immutable, so we *cannot replace or delete any of their items*. If we want to be able to modify an ordered sequence, we simply use a list instead of a tuple; or if we already have a tuple but want to modify it, we can convert it to a list using the list() conversion function and then apply the changes to the resultant list.

- The tuple data type can be called as a function, tuple()—with no arguments it returns an empty tuple, with a tuple argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a tuple. It does not accept more than one argument.

- Tuples can also be created without using the tuple() function. An empty tuple is created using empty parentheses, (), and a tuple of one or more items can be created by using commas. Sometimes tuples must be enclosed in parentheses to avoid syntactic ambiguity. For example, to pass the tuple 1, 2, 3 to a function, we would write function((1, 2, 3)).

- Strings are indexed in the same way, but whereas strings have a character at every position, tuples have an object reference at each position.

- Tuples provide just two methods, t.count(x), which returns the number of times object x occurs in tuple t, and t.index(x), which returns the index position of the leftmost occurrence of object x in tuple t—or raises a ValueError exception if there is no x in the tuple. (These methods are also available for lists.)

- In addition, tuples can be used with the operators + (concatenation), * (replication), and [] (slice), and with-in and not in to test for membership.

- The += and *= augmented assignment operators can be used even though tuples are immutable—behind the scenes *Python creates a new tuple to hold the result* and sets the left-hand object reference to refer to it; the same technique is used when these operators are applied to strings.

- Tuples can be compared using the standard comparison operators (<, <=, ==, !=, >=, >), with the comparisons being applied item by item (and recursively for nested items such as tuples inside tuples).

- The elements of a tuple have a defined order, just like a list. Tuple indices are zero-based, just like a list, so the first element of a non-empty tuple is always a_tuple[0].

- Negative indices count from the end of the tuple, just like a list. Slicing works too, just like a list. When you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

TUPLE

CREATING

Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also. For example

```
>>> tup1 = ('physics', 'chemistry', 1997, 2000)
>>> tup2 = (1, 2, 3, 4, 5 )
>>> tup3 = "a", "b", "c", "d"
```

The empty tuple is written as two parentheses containing nothing.

```
>>> tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value.

```
>>> tup1 = (50,)
```

```
>>> 42
42
>>> 42,
(42,)
>>> (42,)
(42,)
```

The last two examples produce tuples of length one, while the first is not a tuple at all. The comma is crucial.

Simply adding parentheses won’t help:

(42) is exactly the same as 42.

One lonely comma, however, can change the value of an expression completely:

```
>>> 3*(40+2)
126
>>> 3*(40+2,)
(42, 42, 42)
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

ACCESSING VALUES IN TUPLES

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

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
```

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

```
tup1[0]: physics
tup2[1:5]: [2, 3, 4, 5]
```

UPDATING TUPLES

Tuples are immutable, which means you *cannot update or change the values of tuple elements*. You are able to take portions of the existing tuples to create new tuples as the following example demonstrates.

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')
# Following action is not valid for tuples
# tup1[0] = 100;
# So let's create a new tuple as follows
tup3 = tup1 + tup2
print (tup3)
```

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

```
(12, 34.56, 'abc', 'xyz')
```

The following code will throw error because we attempted to update a Tuple. A Tuple is read-only. But update is possible with lists.

```
>>> tuple = ( 'abc', 44 , 3.14, 'xyz', 33.33 )
>>> list = [ 'abc', 44 , 3.14, 'xyz', 33.33 ]
>>> list[2] = 88                       # Valid syntax with list
>>> tuple[2] = 88                      # Invalid syntax with tuple
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

DELETE TUPLE ELEMENTS

Removing *individual tuple elements is not possible*. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
To explicitly remove an entire tuple, just use the del statement. For example-

```
tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del tup;

print "After deleting tup : "
print tup
```

This produces the following result.
Note: An exception is raised. This is because after del tup, tuple does not exist any more.

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
File "test.py", line 9, in <module>
print tup;
NameError: name 'tup' is not defined
```

Also, note that the rule about tuple *immutability* applies *only to the top level* of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual:

```
>>>
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
T[1] = 'spam'
TypeError: 'tuple' object does not support item assignment
>>>
>>> T[1][0] = 'spam'
>>> T
(1, ['spam', 3], 4)
>>>
```

TUPLE

BASIC TUPLES OPERATIONS

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the *result is a new tuple*, not a string. In fact, tuples respond to all of the general sequence operations we used on strings.

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

Indexing, Slicing, and Matrixes

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input-
T=('C++', 'Java', 'Python')

Python Expression	Results	Description
T[2]	'Python'	Offsets start at zero
T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

Let's look at a few slicing examples, starting with extracting one item, and a slice of items:

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:]                    # same as: hair[1:]
('brown', 'blonde', 'red')
>>>
```

These work the same for strings, lists, and any other sequence type.

```
>>> hair[:2], "gray", hair[2:]
(('black', 'brown'), 'gray', ('blonde', 'red'))
```

Here we tried to create a new 5-tuple, but ended up with a 3-tuple that contains two 2-tuples. This happened because we used the comma operator with three items (a tuple, a string, and a tuple).

To get a single tuple with all the items we must concatenate tuples:

```
>>>
>>> hair[:2] + ("gray",) + hair[2:]
('black', 'brown', 'gray', 'blonde', 'red')
>>>
```

There is no obligation to follow this coding style; some programmers prefer to always use parentheses—which is the same as the tuple representational form, whereas others use them only if they are strictly necessary.

```
>>>
>>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
('green', 'blue')
>>>
```

Here we have nested two tuples inside another tuple. Nested collections to any level of depth can be created like this without formality.

The slice operator [] can be applied to a slice, with as many used as necessary. For example:

```
>>>
>>> things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
>>> things[2][1][1][2]
'z'
>>>
```

Let’s look at this piece by piece:

- things[2] which gives us the third item in the tuple (since the first item has index 0), which is itself a tuple, ("pea", (5, "Xyz"), "queue").
- The expression things[2][1] gives us the second item in the things[2] tuple, which is again a tuple, (5, "Xyz").
- And things[2][1][1] gives us the second item in this tuple, which is the string "Xyz".
- Finally, things[2][1][1][2] gives us the third item (character) in the string, that is, "z".

BUILT-IN TUPLE FUNCTIONS

Python includes the following tuple functions-

Function	Description
cmp(tuple1, tuple2)	No longer available in Python 3
len(tuple)	Gives the total length of the tuple
max(tuple)	Returns item from the tuple with max value
min(tuple)	Returns item from the tuple with min value
tuple(seq)	Converts a list into tuple

Tuple len() Method

Description

The **len()** method returns the number of elements in the tuple.

Syntax

Following is the syntax for len() method: len(tuple)

Parameters

tuple - This is a tuple for which number of elements to be counted.

Return Value

This method returns the number of elements in the tuple.

Example

The following example shows the usage of len() method.

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

When we run above program, it produces following result-

```
First tuple length : 3
Second tuple length : 2
```

Tuple max() Method

Description

The **max()** method returns the elements from the tuple with maximum value.

Syntax

Following is the syntax for max() method: max(tuple)

Parameters

tuple - This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("Max value element : ", max(tuple1))
print ("Max value element : ", max(tuple2))
```

When we run the above program, it produces the following result-

```
Max value element : phy
Max value element : 700
```

Tuple min() Method

Description

The **min()** method returns the elements from the tuple with minimum value.

Syntax

Following is the syntax for min() method: min(tuple)

Parameters

tuple - This is a tuple from which min valued element is to be returned.

Return Value

This method returns the elements from the tuple with minimum value.

Example

The following example shows the usage of min() method.

```
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("min value element : ", min(tuple1))
print ("min value element : ", min(tuple2))
```

When we run the above program, it produces the following result

```
Min value element : bio
min value element : 200
```

Tuple tuple() Method

Description

The **tuple()** method converts a list of items into tuples.

Syntax

Following is the syntax for tuple() method: tuple(seq)

Parameters

seq - This is a tuple to be converted into tuple.

Return Value

This method returns the tuple.

Example

The following example shows the usage of tuple() method.

```
list1= ['maths', 'che', 'phy', 'bio']
tuple1=tuple(list1)
print ("tuple elements : ", tuple1)
```

When we run the above program, it produces the following result

```
Tuple elements : ('maths', 'che', 'phy', 'bio')
```

TUPLE

Methods and attributes found in list or tuple (methods implemented by object are omitted for brevity).

METHOD	List	Tuple	DESCRIPTION
s.__add__(s2)	●	●	s + s2 — concatenation
s.__iadd__(s2)	●		s += s2 — in-place concatenation
s.append(e)	●		append one element after last
s.clear()	●		delete all items
s.__contains__(e)	●	●	e in s
s.copy()	●		shallow copy of the list
s.count(e)	●	●	count occurrences of an element
s.__delitem__(p)	●		remove item at position p
s.extend(it)	●		append items from iterable it
s.__getitem__(p)	●	●	s[p] — get item at position
s.__getnewargs__()	●		support for optimized serialization with pickle
s.index(e)	●	●	find position of first occurrence of e
s.insert(p, e)	●		insert element e before the item at position p
s.__iter__()	●	●	get iterator
s.__len__()	●	●	len(s) — number of items
s.__mul__(n)	●	●	s * n — repeated concatenation
s.__imul__(n)	●		s *= n — in-place repeated concatenation
s.__rmul__(n)	●	●	n * s — reversed repeated concatenation
s.pop(≪p≫)	●		remove and return last item or item at optional position p
s.remove(e)	●		remove first occurrence of element e by value
s.reverse()	●		reverse the order of the items in-place
s.__reversed__()	●		get iterator to scan items from last to first
s.__setitem__(p, e)	●		s[p] = e — put e in position p, overwriting existing item
s.sort(≪key≫, ≪reverse≫)	●		sort items in place with optional keyword arguments key and reverse

In practical terms, they have no methods that would allow you to change them. Lists have methods like append(), extend(), insert(), remove(), and pop(). Tuples have none of these methods. You can slice a tuple (because that creates a new tuple), and you can check whether a tuple contains a particular value (because that doesn't change the tuple), and… that's about it.

EXERCISE:

1. Open a Python Shell window.

You see the familiar Python prompt.

2. Type MyTuple = (“Red”, “Blue”, “Green”) and press Enter.

Python creates a tuple containing three strings.

3. Type MyTuple and press Enter.

You see the content of MyTuple, which is three strings. Notice that the entries use single quotes, even though you used double quotes to create the tuple. In addition, notice that a tuple uses parentheses rather than square brackets, as lists do.

4. Type dir(MyTuple) and press Enter.

Python presents a list of functions that you can use with tuples,. Notice that the list of functions appears significantly smaller than the list of functions provided with lists. The count() and index() functions are present.

However, appearances can be deceiving. For example, you can add new items using the __add__() function. When working with Python objects, look at all the entries before you make a decision as to functionality.

5. Type MyTuple = MyTuple.__add__((“Purple”,)) and press Enter.

This code adds a new tuple to MyTuple and places the result in a new copy of MyTuple. The old copy of MyTuple is destroyed after the call.

The __add__() function accepts only tuples as input. This means that you must enclose the addition in parentheses. In addition, when creating a tuple with a single entry, you must add a comma after the entry. This is an odd Python rule that you need to keep in mind or you’ll see an error message similar to this one:

```
TypeError: can only concatenate tuple (not "str") to tuple
```

6. Type MyTuple and press Enter.

The addition to MyTuple appears at the end of the list. Notice that it appears at the same level as the other entries.

7. Type MyTuple = MyTuple.__add__((“Yellow”, (“Orange”, “Black”))) and press Enter.

This step adds three entries: Yellow, Orange, and Black. However, Orange and Black are added as a tuple within the main tuple, which creates a hierarchy. These two entries are actually treated as a single entry within the main tuple.

You can replace the __add__() function with the concatenation operator. For example, if you want to add Magenta to the front of the tuple list, you type MyTuple = ("Magenta",) + MyTuple.

8. Type MyTuple[4] and press Enter.

Python displays a single member of MyTuple, ‘Orange’. Tuples use indexes to access individual members, just as lists do. You can also specify a range when needed. Anything you can do with a list index you can also do with a tuple index.

9. Type MyTuple[5] and press Enter.

You see a tuple that contains’ Orange’ and ‘Black’. Of course, you might not want to use both members in tuple form.

Tuples do contain hierarchies on a regular basis. You can detect when an index has returned another tuple, rather than a value, by testing for type. For example, in this case, you could detect that the sixth item (index 5) contains a tuple by typing type(MyTuple[5]) == tuple. The output would be True in this case.

10. Type MyTuple[5][0] and press Enter.

At this point, you see Orange as output. The indexes always appear in order of their level in the hierarchy.

Using a combination of indexes and the __add__() function (or the concatenation operator, +), you can create flexible applications that rely on tuples. For example, you can remove an element from a tuple by making it equal to a range of values.

If you wanted to remove the tuple containing Orange and Black, you type MyTuple = MyTuple[0:5].

TUPLE

ASSIGNING MULTIPLE VALUES AT ONCE

Here's a cool programming shortcut: in Python, you can use a tuple to assign multiple values at once.

```
>>>
>>> v = ('a', 2, True)
>>> (x, y, z) = v
>>> x
'a'
>>> y
2
>>> z
True
>>>
```

v is a tuple of three elements, and (x, y, z) is a tuple of three variables. Assigning one to the other assigns each of the values of v to each of the variables, in order. Suppose you want to assign names to a range of values. You can use the built-in range() function with multi-variable assignment to quickly assign consecutive values.

```
>>>
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY) = range(7)
>>> MONDAY
0
>>> TUESDAY
1
>>> SUNDAY
6
>>>
```

- The built-in range() function constructs a sequence of integers. (Technically, the range() function returns an iterator, not a list or a tuple). MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are the variables you're defining.

- 2. Now each variable has its value: MONDAY is 0, TUESDAY is 1, and so forth. You can also use multi-variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a single tuple, or it can assign the values to individual variables.

PACK – UNPACK :

Remember, the **for** loop assigns **items** in the **sequence** object to the **target**, and assignment works the same everywhere:

```
>>>
>>> T = [(1, 2), (3, 4), (5, 6)]

>>> for (a, b) in T:                    # Tuple assignment at work
        print(a, b)
1 2
3 4
5 6
>>>
```

Here, the first time through the loop is like writing (a,b) = (1,2), the second time is like writing (a,b) = (3,4), and so on. The net effect is to automatically **unpack** the current tuple on each iteration.

It's important to note that tuple assignment in **for** loops isn't a special case; ***any assignment target*** works syntactically after the word **for**. We can always assign manually within the loop to unpack:

```
>>>
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
        a, b = both             # Manual assignment equivalent
        print(a, b)             # 2.X: prints with enclosing tuple "()"
1 2
3 4
5 6
>>>
```

Tuples in the loop header save us an extra step when iterating through sequences of sequences.

```
>>>
>>> ((a, b), c) = ((1, 2), 3)                # Nested sequences work too
>>> a, b, c
(1, 2, 3)
>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]:
        print(a, b, c)
1 2 3
4 5 6
>>>
```

Any nested sequence structure may be unpacked this way.

```
>>>
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]:
        print(a, b, c)
1 2 3
X Y 6
>>>
```

Consider the tuple assignment form introduced in the prior section. ***A tuple of values is assigned to a tuple of names on each iteration***, exactly like a simple assignment statement:

```
>>> a, b, c = (1, 2, 3)                        # Tuple assignment
>>> a, b, c
(1, 2, 3)
>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:          # Used in for loop
        print(a, b, c)
1 2 3
4 5 6
 >>>
```

Because a sequence can be assigned to a more general set of names with a **starred name** to collect multiple items, we can use the same syntax to extract parts of nested sequences in the **for** loop:

```
>>> a, *b, c = (1, 2, 3, 4)                    # Extended seq assignment
>>> a, b, c
(1, [2, 3], 4)
>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
        print(a, b, c)
1 [2, 3] 4
5 [6, 7] 8
>>>
```

Manual slicing in 2.X

```
>>>
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:          # Manual slicing in 2.X
        a, b, c = all[0], all[1:3], all[3]
        print(a, b, c)
1 (2, 3) 4
5 (6, 7) 8
>>>
```