Python Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses `()`, separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

script.py IPython Shell

```
1    # Empty tuple
2    my_tuple = ()
3    print(my_tuple)  # Output: ()
4
5    # Tuple having integers
6    my_tuple = (1, 2, 3)
7    print(my_tuple)  # Output: (1, 2, 3)
8
9    # tuple with mixed datatypes
10   my_tuple = (1, "Hello", 3.4)
11   print(my_tuple)  # Output: (1, "Hello", 3.4)
12
13   # nested tuple
14   my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
15
16   # Output: ("mouse", [8, 4, 6], (1, 2, 3))
17   print(my_tuple)
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
1    my_tuple = 3, 4.6, "dog"
2    print(my_tuple)    # Output: 3, 4.6, "dog"
3
4    # tuple unpacking is also possible
5    a, b, c = my_tuple
6
7    print(a)       # 3
8    print(b)       # 4.6
9    print(c)       # dog
```

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple))  # <class 'str'>

# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))  # <class 'tuple'>

# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple))  # <class 'tuple'>
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator `[]` to access an item in a tuple where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an element outside of tuple (for example, 6, 7,...) will raise an `IndexError`.

The index must be an integer; so we cannot use float or other types. This will result in `TypeError`.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])        # 's'
print(n_tuple[1][1])        # 4
```

Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

Slicing

We can access a range of items in a tuple by using the slicing operator - colon ":".

```
my_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
my_tuple = (4, 2, 3, [6, 5])


# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9

# However, item of mutable element can be changed
my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p','r','o','g','r','a','m','i','z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
```

We can use + operator to combine two tuples. This is also called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the `*` operator.

Both `+` and `*` operations result in a new tuple.

Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Python Tuple Method	
Method	Description
`count(x)`	Returns the number of items `x`
`index(x)`	Returns the index of the first item that is equal to `x`

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)

print(my_tuple.count('p'))  # Output: 2
print(my_tuple.index('l'))  # Output: 3
```

Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword `in`.

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
# Output: True
print('g' not in my_tuple)
```

Iterating Through a Tuple

Using a `for` loop we can iterate through each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John','Kate'):
    print("Hello",name)
```

Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuples are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Nesting List and tuple

We can store list inside tuple or tuple inside the list up to any number of level.

Lets see an example of how can we store the tuple inside the list.

```
Employees = [(101, "Ayush", 22), (102, "john", 29), (103, "james", 45), (104, "Ben", 34)]
print("----Printing list----");
for i in Employees:
   print(i)
Employees[0] = (110, "David",22)
print();
print("----Printing list after modification----");
for i in Employees:
   print(i)
```

Output

```
----Printing list----
(101, 'Ayush', 22)
(102, 'john', 29)
(103, 'james', 45)
(104, 'Ben', 34)

----Printing list after modification----

(110, 'David', 22)
(102, 'john', 29)
(103, 'james', 45)
(104, 'Ben', 34)
```