Python Tuple

In Python programming, a tuple 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.

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 tuple 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 key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Creating a Tuple

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. The parentheses are optional but is a good practice to write it.

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

```
# empty tuple
# Output: ()
my_tuple = ()
print(my_tuple)
# tuple having integers
# Output: (1, 2, 3)
my_tuple = (1, 2, 3)
print(my_tuple)
```

```
# tuple with mixed datatypes
# Output: (1, "Hello", 3.4)
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
# nested tuple
# Output: ("mouse", [8, 4, 6], (1, 2, 3))
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
# tuple can be created without parentheses
# also called tuple packing
```

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.

```
# only parentheses is not enough
# Output: <class 'str'>
my_tuple = ("hello")
print(type(my_tuple))
# need a comma at the end
# Output: <class 'tuple'>
my_tuple = ("hello",)
print(type(my_tuple))
# parentheses is optional
# Output: <class 'tuple'>
my_tuple = "hello",
print(type(my_tuple))
```

Accessing Elements in a Tuple

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 index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result into TypeError.

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

```
my_tuple = ('p','e','r','m','i','t')
# Output: 'p'
print(my_tuple[0])
# Output: 't'
print(my_tuple[5])
# index must be in range
# If you uncomment line 14,
# you will get an error.
# IndexError: list index out of range
#print(my_tuple[6])
# index must be an integer
# If you uncomment line 21,
# you will get an error.
# TypeError: list indices must be integers, not float
#my_tuple[2.0]
```

When you run the program, the output will be:

```
p

t

s

4
```

2. 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.

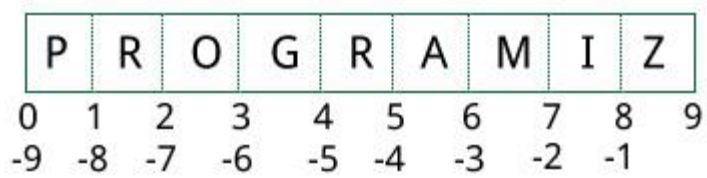
```
my_tuple = ('p','e','r','m','i','t')
# Output: 't'
print(my_tuple[-1])
# Output: 'p'
print(my_tuple[-6])
```

3. 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[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the 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])
# we cannot change an element
# If you uncomment line 8
# you will get an error:
# TypeError: 'tuple' object does not support item assignment
#my_tuple[1] = 9
# but item of mutable element can be changed
# Output: (4, 2, 3, [9, 5])
my_tuple[3][0] = 9
print(my_tuple)
# tuples can be reassigned
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
my_tuple = ('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 into a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))
# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword del.

```
my_tuple = ('p','r','o','g','r','a','m','i','z')
# can't delete items
# if you uncomment line 8,
# you will get an error:
# TypeError: 'tuple' object doesn't support item deletion
#del my_tuple[3]
# can delete entire tuple
# NameError: name 'my_tuple' is not defined
del my_tuple
my_tuple
```

Python 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)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)
# Count
# Output: 2
```

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

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)
```

2. Iterating Through a Tuple

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

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

Built-in Functions with Tuple

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `tuple()`etc. are commonly used with tuple to perform different tasks.

Built-in Functions with Tuple

Function	Description
all()	Return `True` if all elements of the tuple are true (or if the tuple is empty).
any()	Return `True` if any element of the tuple is true. If the tuple is empty, return `False`.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.