Tuples

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

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,);
```

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

Properties:

1.Ordered: Tuples maintain the order of elements as they are defined.

2.Immutable: Once a tuple is created, its elements cannot be changed or modified. This makes tuples suitable for storing data that should not be altered accidentally.

3.Heterogeneous Data Types: Tuples can contain elements of different data types (integers, floats, strings, etc.).

4.Indexing and Slicing: Like lists, tuples support indexing to access individual elements and slicing to access multiple elements at once.

5.Iterability: Tuples are iterable, meaning you can iterate over them using loops or comprehensions.

6.Nesting: Tuples can be nested within each other to create more complex data structures.

Advantages:

1.Immutable: Once created, tuples cannot be modified. This immutability ensures data integrity and prevents accidental changes to critical data.

2.Performance: Tuples are generally faster than lists for accessing elements. Since tuples are immutable, Python can optimize memory usage and access operations more efficiently.

3.Sequence Operations: Tuples support all sequence operations like indexing, slicing, and iteration, making them versatile for data access and manipulation.

4.Valid Dictionary Keys: Tuples can be used as keys in dictionaries because they are immutable and hashable, whereas lists cannot be used as dictionary keys.

5.Function Arguments: Tuples are commonly used to pass data to and return data from functions. They can hold multiple values and maintain their integrity.

Disadvantages:

1.Immutable: While immutability is an advantage for data integrity, it can be a limitation when you need to modify elements in-place. Lists are more suitable if you require dynamic changes to your data.

2.Limited Methods: Tuples have fewer built-in methods compared to lists. For instance, you cannot append or remove elements from a tuple since it is immutable.

3.Less Flexible: Due to their immutability, tuples are less flexible in terms of data manipulation compared to lists. Operations like sorting or adding/removing elements are not directly supported.

4.Less Common: Tuples are less commonly used compared to lists in Python programming, especially for general-purpose data storage where dynamic changes are anticipated.

When to Use Tuples:

- Use tuples when you have data that should not change, such as constant values, configuration settings, or fixed collections of related values.
- Use tuples for sequence-like data where immutability is an advantage, such as representing coordinates, dates, or records that shouldn't be altered.
- Use tuples as dictionary keys when you need a key that won't change.

When to Avoid Tuples:

- Avoid tuples when you need to modify your data frequently or perform operations like sorting or appending elements.
- Avoid tuples if you need a data structure that can grow or shrink dynamically, as lists are more suitable for such scenarios.

Time Complexity:

1. **Access (Indexing and Slicing)**:
   - Indexing and slicing operations in tuples have a time complexity of O(1). This is because tuples are implemented as contiguous arrays in memory, so accessing an element by index or slicing a portion of a tuple is a direct operation.
2. **Iteration**:
   - Iterating over a tuple with a `for` loop or using comprehensions has a time complexity of O(n), where n is the number of elements in the tuple. This is because it requires visiting each element once.

Space Complexity:

1. **Storage**:
   - Tuples in Python store their elements in a contiguous block of memory. The space complexity for tuples is O(n), where n is the number of elements in the tuple. This space complexity includes the space required to store each element.
2. **Additional Overhead**:
   - Tuples also have additional overhead associated with storing references to objects, especially when tuples contain objects of varying sizes or types. This overhead is typically small but can contribute to space complexity considerations in memory-constrained environments.

Summary:

- **Time Complexity**: O(1) for indexing and slicing, O(n) for iteration.
- **Space Complexity**: O(n) where n is the number of elements in the tuple.

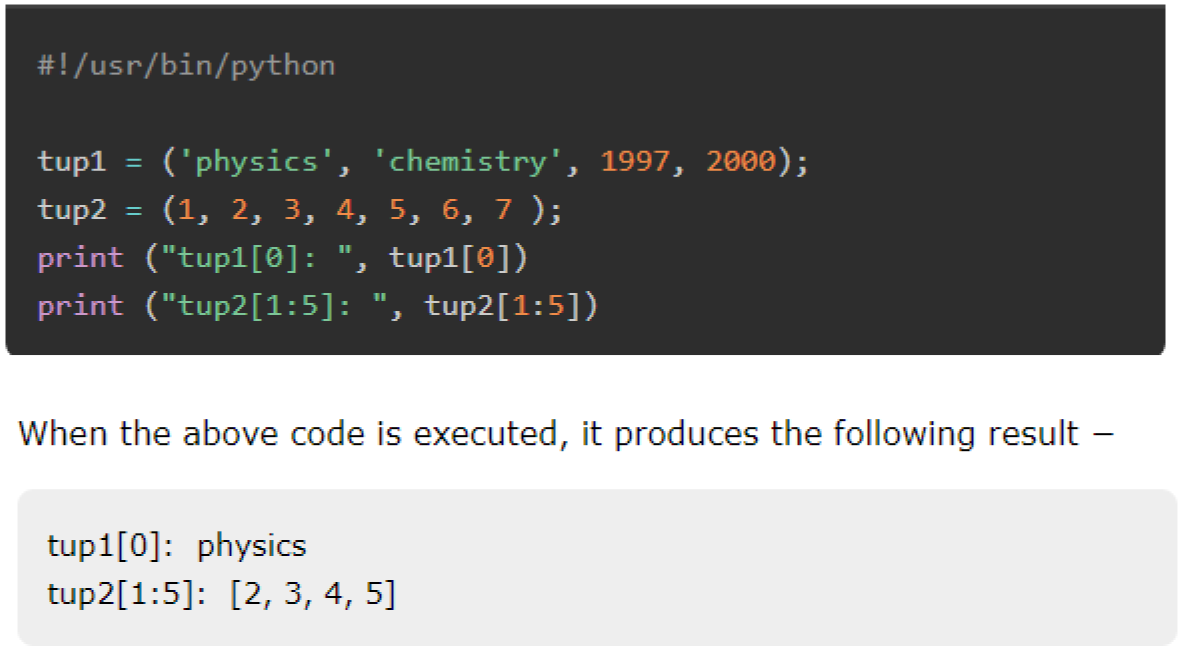
These complexities make tuples efficient for accessing and storing fixed collections of data, especially when immutability and order preservation are required.

Basic Operations performed on tuple:

Accessing Values in Tuples

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

For example



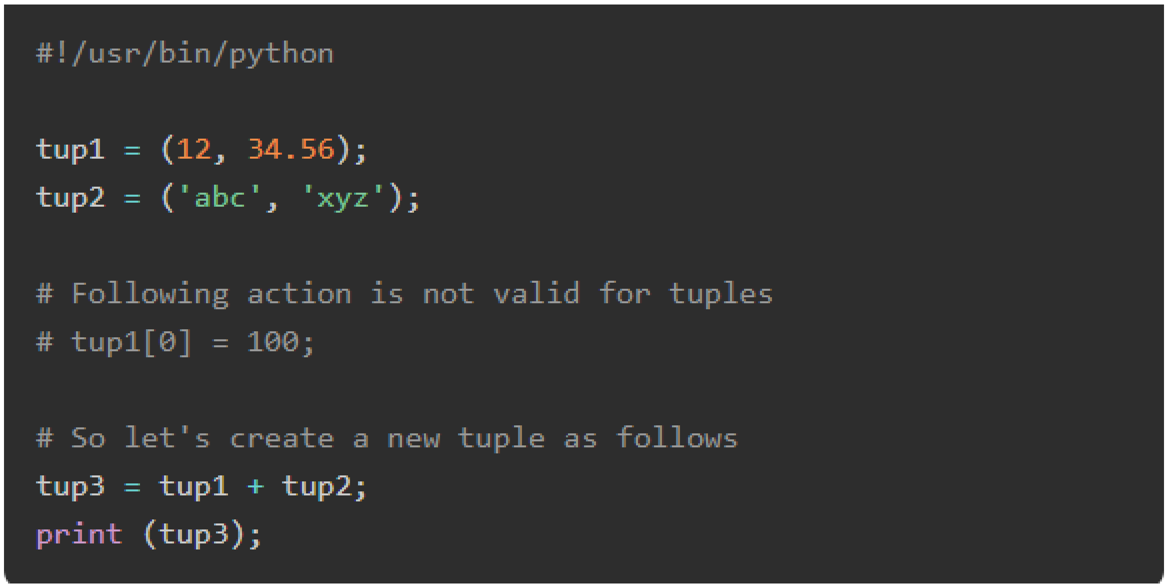
```
#!/usr/bin/python

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 −

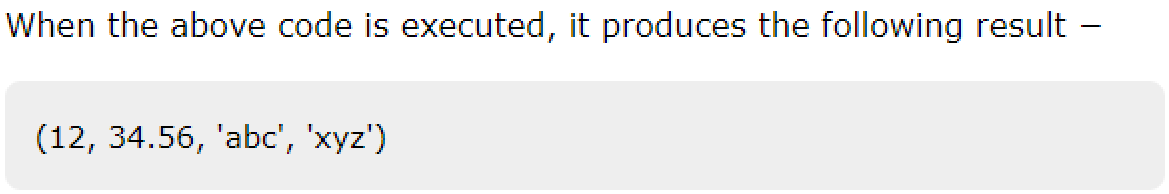
```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples

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



```
#!/usr/bin/python

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')
```

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

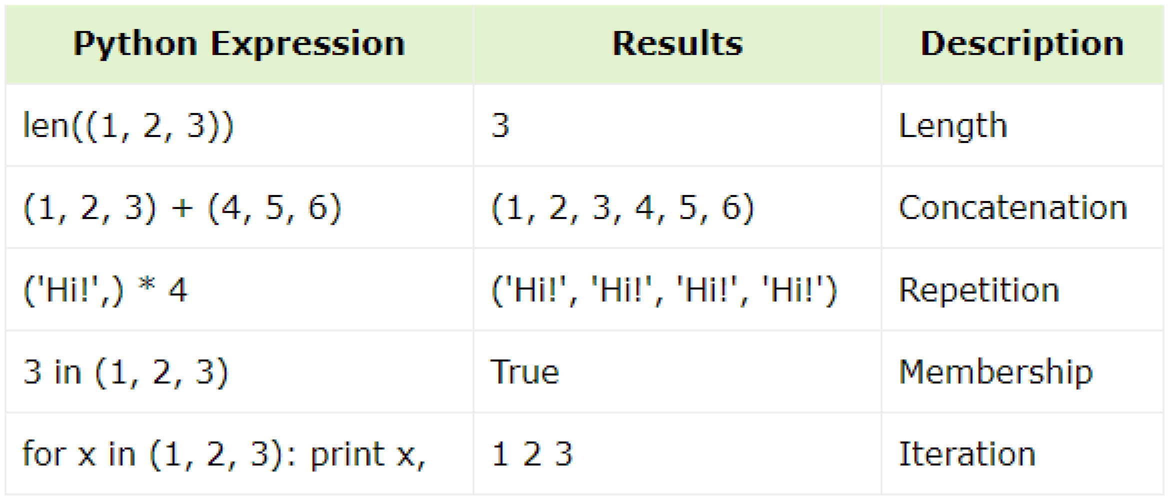
```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);
print (tup);
del tup;
print ("After deleting tup : ");
print (tup);
```

- **Note** – an exception raised, this is because after **del tup** tuple does not exist anymore.

```
('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
```

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.



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,	1 2 3	Iteration