**Operations On Tuple**

Indexing

-- We can use the index operator [] to access an object in a tuple, where the index starts at 0.

-- The indices of a tuple with five items will range from 0 to 4.

-- An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record.

-- An index above four will be out of range in this scenario.

-- Because the index in Python must be an integer, we cannot provide an index of a floating data type or any other type. If we provide a floating index, the result will be TypeError.

# *Python program to show how to access tuple elements*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Collection")

print(tuple\_[0])        # *o/p: Python*

print(tuple\_[1])        # *o/p: Tuple*

# *trying to access element index more than the length of a tuple*

*try*:

    print(tuple\_[5])

*except* Exception *as* e:

    print(e)            # *o/p: tuple index out of range*

# *trying to access elements through the index of floating data type*

*try*:

    print(tuple\_[1.0])

*except* Exception *as* e:

    print(e)            # *o/p: tuple indices must be integers or slices, not float*

# *Creating a nested tuple*

nested\_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))

# *Accessing the index of a nested tuple*

print(nested\_tuple[0][3])   # *o/p: 1*

print(nested\_tuple[1][1])   # *o/p: 6*

-- Python's sequence objects support negative indexing.

-- The last thing of the assortment is addressed by - 1, the second last thing by - 2, etc.

# *Python program to show how negative indexing works in Python tuples*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Collection")

# *Printing elements using negative indices*

print("Element at -1 index: ", tuple\_[-1])

# *o/p: Element at -1 index:  Collection*

print("Elements between -4 and -1 are: ", tuple\_[-4:-1])

# *o/p: Elements between -4 and -1 are: ('Python', 'Tuple', 'Ordered')*

Slicing

-- Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues.

-- Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.

-- To gain access to various tuple elements, we can use the slicing operator colon (:).

# *Python program to show how slicing works in Python tuples*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")

# *Using slicing to access elements of the tuple*

print("Elements between indices 1 and 3: ", tuple\_[1:3])

# *o/p: Elements between indices 1 and 3:  ('Tuple', 'Ordered')*

# *Using negative indexing in slicing*

print("Elements between indices 0 and -4: ", tuple\_[:-4])

# *o/p: Elements between indices 0 and -4:  ('Python', 'Tuple')*

# *Printing the entire tuple by using the default start and end values.*

print("Entire tuple: ", tuple\_[:])

# *o/p: Entire tuple:  ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection', 'Objects')*

Deleting A Tuple

-- A tuple's parts can't be modified, as was recently said. We are unable to eliminate or remove tuple components as a result.

-- However, the keyword del can completely delete a tuple.

# *Python program to show how to delete elements of a Python tuple*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")

# *Deleting a particular element of the tuple*

*try*:

*del* tuple\_[3]

    print(tuple\_)

*except* Exception *as* e:

    print(e)            # *o/p: 'tuple' object does not support item deletion*

# *Deleting the variable from the global space of the program*

*del* tuple\_

# *Trying accessing the tuple after deleting it*

*try*:

    print(tuple\_)

*except* Exception *as* e:

    print(e)            # *o/p: name 'tuple\_' is not defined*

-- Repeating the tuple:

# *Python program to show repetition in tuples*

tuple\_ = ('Python', "Tuples")

print("Original tuple is: ", tuple\_)        # *o/p: Original tuple is:  ('Python', 'Tuples')*

# *Repeating the tuple elements*

tuple\_ = tuple\_ \* 3

print("New tuple is: ", tuple\_)

# *o/p: New tuple is:  ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')*

Count() Method

-- The times the predetermined component happens in the Tuple is returned by the count () capability of the Tuple.

# *Creating tuples*

T1 = (0, 1, 5, 6, 7, 2, 2, 4, 2, 3, 2, 3, 1, 3, 2)

T2 = ('python', 'java', 'python', 'Tpoint', 'python', 'java')

# *counting the appearance of 3*

res = T1.count(2)

print('Count of 2 in T1 is:', res)      # *o/p: Count of 2 in T1 is: 5*

# *counting the appearance of java*

res = T2.count('java')

print('Count of Java in T2 is:', res)   # *o/p: Count of java in T2 is: 2*

Index() Method

-- While working with tuples many times we need to access elements at a certain index but for that, we need to know where exactly is that element, and here comes the use of the index() function.

-- The Index() method returns the first occurrence of the given element from the tuple.

my\_tuple = (4, 2, 5, 6, 7, 5)

print(my\_tuple.index(5))        # *o/p: 2*

# *Creating tuples*

Tuple = ( 1, 3, 4, 2, 5, 6 )

# *getting the index of 3*

res = Tuple.index(3)

print('Index of 3 is', res)     # *o/p: Value of Index of 3 is 1*

-- Here we are finding the index of a particular element in a tuple with multiple occurrences but here it only returns the first occurrence of that element.

# *Creating tuples*

Tuple = ( 3, 3, 5, 7, 3, 3 )

# *getting the index of 3*

res = Tuple.index(3)

print('Index of 3 is', res)     # *o/p: Index of 3 is 0*

-- Here we are finding the index of a particular element in a tuple in the given range.

# *alphabets tuple*

alphabets = ('G', 'e', 'e', 'k', 's', 'f', 'o',

             'r', 'G', 'e', 'e', 'k', 's')

# *scans 'G' from index 4 to 10 and*

# *returns its index*

index = alphabets.index('G', 4, 10)

print('Index of G in alphabets from index 4 to 10:', index)

# *o/p: Index of G in alphabets from index 4 to 10: 8*

-- Here we are finding the index of a particular element in a tuple that does not exist.

t = ('G', 'F', 'G')

# *accessing element not present in the tuple*

print(t.index('i'))

# *o/p: ValueError: tuple.index(x): x not in tuple*

-- Utilizing the watchword, we can decide whether a thing is available in the given Tuple.

# *Python program to show how to perform membership test for tuples*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")

# *In operator*

print('Tuple' *in* tuple\_)            # *o/p: True*

print('Items' *in* tuple\_)            # *o/p: False*

# *Not in operator*

print('Immutable' *not* *in* tuple\_)    # *o/p: False*

print('Items' *not* *in* tuple\_)        # *o/p: True*

-- A for loop can be used to iterate through each tuple element.

# *Python program to show how to iterate over tuple elements*

# *Creating a tuple*

tuple\_ = ("Python", "Tuple", "Ordered", "Immutable")

# *Iterating over tuple elements using a for loop*

*for* item in tuple\_:

    print(item)

Edit Objects Inside Tuple

-- Sometimes, while working with tuples, being immutable can offer a lot of confusion regarding its working.

-- One of the questions that can pop into the mind is, Are objects inside tuples mutable? The answer to this is **Yes**.

**Method #1: Using Access methods**

-- This is one of the ways in which edit inside objects of tuples can be performed.

-- This occurs similarly to any other container and in place using list access method.

# *Python3 code to demonstrate working of Edit objects inside tuple Using Access Methods*

# *initializing tuple*

test\_tuple = (1, [5, 6, 4], 9, 10)

# *printing original tuple*

print("The original tuple : " + str(test\_tuple))

# *o/p: The original tuple : (1, [5, 6, 4], 9, 10)*

# *Edit objects inside tuple Using Access Methods*

test\_tuple[1][2] = 14

# *printing result*

print("The modified tuple : " + str(test\_tuple))

# *o/p: The modified tuple : (1, [5, 6, 14], 9, 10)*

**Method #2 : Using pop() + index()**

-- The combination of the above functionalities can also be used to solve this problem. In this, we perform the task of removal using pop() and add an element at a particular index using index().

# *Python3 code to demonstrate working of Edit objects inside tuple Using pop() + index()*

# *initializing tuple*

test\_tuple = (1, [5, 6, 4], 9, 10)

# *printing original tuple*

print("The original tuple : " + str(test\_tuple))

# *o/p: The original tuple : (1, [5, 6, 4], 9, 10)*

# *Edit objects inside tuple Using pop() + index()*

test\_tuple[1].pop(2)

test\_tuple[1].insert(2, 14)

# *printing result*

print("The modified tuple : " + str(test\_tuple))

# *o/p: The modified tuple : (1, [5, 6, 14], 9, 10)*

#### Method #3 : Using the list

-- Convert the tuple to a list, edit the list, and convert it back to a tuple.

-- Edit the list by accessing the desired index and assigning a new value to it.

-- Convert the edited list back to a tuple using the tuple() function.

# *Initial tuple*

my\_tuple = (1, 2, 3, 4, 5)

# *Convert the tuple to a list*

my\_list = list(my\_tuple)

# *Edit the list*

my\_list[2] = 10

# *Convert the list back to a tuple*

my\_tuple = tuple(my\_list)

# *Print the updated tuple*

print(my\_tuple)

# *o/p: (1, 2, 10, 4, 5)*

### Method #4: Using lambda function

-- Define a lambda function to modify the given tuple by replacing the element at the specified index with a new value.

-- Call the lambda function with the original tuple, index of the element to be replaced, and the new value.

-- Define a lambda function modify\_tuple that takes three arguments – the original tuple, the index of the element to be replaced, and the new value.

-- Use slicing to create a new tuple that has the same elements as the original tuple up to the specified index, followed by the new value, and then the remaining elements of the original tuple.

-- Return the new tuple.

# *Using lambda function to modify the tuple*

modify\_tuple = lambda tup, index, new\_val: tup[:index] + (new\_val,) + tup[index+1:]

# *Example usage*

tup = (1, [5, 6, 4], 9, 10)

new\_tup = modify\_tuple(tup, 1, [5, 6, 14])

print(new\_tup)

# *o/p: (1, [5, 6, 14], 9, 10)*

### ****Method #5: Using the slicing****

# *Initial tuple*

my\_tuple = (1, 2, 3, 4, 5)

# *Convert the tuple to a list*

my\_list = list(my\_tuple)

# *Edit the list using slicing*

my\_list[2:3] = [10]

# *Convert the list back to a tuple*

my\_tuple = tuple(my\_list)

# *Print the updated tuple*

print(my\_tuple)     # *o/p: (1, 2, 10, 4, 5)*

**METHOD 6: By creating a new tuple**

**--** The approach modifies an object inside a tuple by creating a new tuple with the desired modification.

-- It uses indexing and concatenation to replace the specified element in the tuple.

-- The modified tuple is then printed as the output.

# *Given input*

tpl = (1, [5, 6, 4], 9, 10)

# *Create a new tuple with the modified object*

modified\_tpl = tpl[:1] + ([tpl[1][0], tpl[1][1], 14] + tpl[1][3:],) + tpl[2:]

# *Output*

print("The modified tuple:", modified\_tpl)

# *o/p: The modified tuple: (1, [5, 6, 14], 9, 10)*

Unpack Tuple Items

-- The term "unpacking" refers to the process of parsing tuple items in individual variables.

-- In Python, the parentheses are the default delimiters for a literal representation of sequence object.

t1 = (1 ,2)

t1 = 1 ,2

type (t1)       # *o/p: <class 'tuple'>*

-- To store tuple items in individual variables, use multiple variables on the left of assignment operator, as shown in the following example –

tup1 = (10,20,30)

x, y, z = tup1

print ("x: ", x, "y: ", "z: ",z)    # *o/p: x: 10 y: 20 z: 30*

-- If the number of variables is more or less than the length of tuple, Python raises a ValueError.

tup1 = (10,20,30)

x, y = tup1             # *o/p: ValueError: too many values to unpack (expected 2)*

x, y, p, q = tup1       # *o/p: ValueError: not enough values to unpack (expected 4, got 3)*

-- In such a case, the "\*" symbol is used for unpacking. Prefix "\*" to "y", as shown below –

tup1 = (10,20,30)

x, \*y = tup1

print ("x: ", "y: ", y) # *o/p: x: y: [20, 30]*

-- Above, The first value in tuple is assigned to "x", and rest of items to "y" which becomes a list.

-- In this example, the tuple contains 6 values and variables to be unpacked are 3. We prefix "\*" to the second variable.

-- Here, values are unpacked in "x" and "z" first, and then the rest of values are assigned to "y" as a list.

tup1 = (10,20,30, 40, 50, 60)

x, \*y, z = tup1

print ("x: ",x, "y: ", y, "z: ", z)     # *o/p: x: 10 y: [20, 30, 40, 50] z: 60*

-- What if we add "\*" to the first variable?

-- Here again, the tuple is unpacked in such a way that individual variables take up the value first, leaving the remaining values to the list "x".

tup1 = (10,20,30, 40, 50, 60)

\*x, y, z = tup1

print ("x: ",x, "y: ", y, "z: ", z)     # *o/p: x: [10, 20, 30, 40] y: 50 z: 60*