1.4 Tuples

Tuples

1. Introduction to Sequence Data Types

A sequence is an ordered collection of elements. It is called ordered because the elements will be maintained in a specific order and the order will not change. Consider the following example.

```
>>> sample = “computer”
>>> my_list = list(sample)
>>> my_list
[‘c’, ‘o’, ‘m’, ‘p’, ‘u’, ‘t’, ‘e’, ‘r’]
>>> my_set = set(sample)
>>> my_set
{‘c’, ‘p’, ‘m’, ‘t’, ‘o’, ‘u’, ‘r’, ‘e’}
```

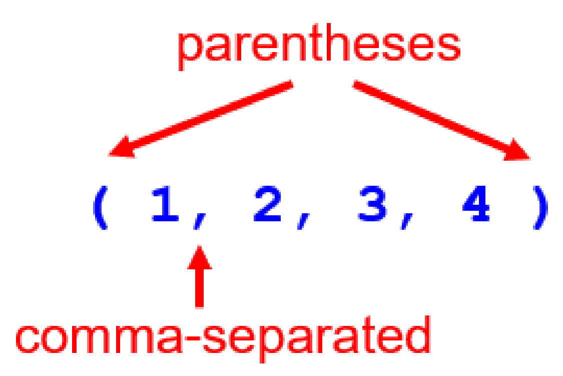
The variable ‘sample’ contains the string “computer”. Then we create a list with these letters. When we print the list, you can see that the list keeps the letters in the same order as the input. Next, we create a set, which is another data type in Python, with these same letters. When we print the set, you can see that the letters are in a completely different order. Therefore, list is categorized as an ordered data type and set as an unordered data type.

Three basic sequence data types in Python are list, tuple, and range. In our first course Python for Beginners, we learned about the list data structure. In this lesson, we will discuss the tuple data structure in detail, which is very similar to lists.

1. 1 Structure of a Tuple

Tuples contain comma-separated values between round brackets/parentheses. The only difference from lists is that the square brackets in lists are replaced by parentheses.

The following diagram illustrates a sample tuple in Python. The values are comma-separated and are within parentheses.



The parentheses are not mandatory when creating a tuple. Similar to lists, the values within a tuple need not be of the same data type.

2. Creating a Tuple

```
>>> tuple_1 = (‘a’, ‘b’, 20, 4.1)
>>> type(tuple_1)
<class ‘tuple’>
>>> tuple_2 = ‘a’, ‘b’, 20, 4.1    #no parentheses
>>> tuple_2
(‘a’, ‘b’, 20, 4.1)
>>> type(tuple_2)
<class ‘tuple’>
```

The variable ‘tuple_1’ contains a tuple with 4 elements. It is important to note that the first two elements ‘a’ and ‘b’ are strings, 20 is an integer and 4.1 is a floating-point number, which are all different data types.

The data type of a variable can be checked using the type function. If we call type(tuple_1), it outputs ‘tuple’ as the data type.

In the previous section (1.1) we mentioned that it is not mandatory to have the parentheses when creating tuples. The variable named ‘tuple_2’ contains the same set of elements as ‘tuple_1’ but without the parentheses. If we print ‘tuple_2’, you can see that the system has added the parentheses by itself. When we check the data type of ‘tuple_2’, it is indeed a tuple.

A tuple with a single element is a special case.

```
>>> tuple_3 = (‘a’)
>>> type(tuple_3)
<class ‘str’>
>>> tuple_4 = (‘a’,)      #equivalently tuple_4 = ‘a’,
>>> type(tuple_4)
<class ‘tuple’>
```

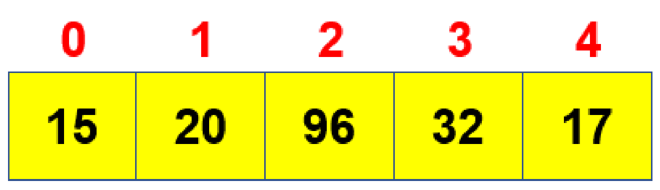
The variable ‘tuple_3’ contains one element which is string ‘a’ inside parentheses. If we check the data type, it turns out to be a string and not a tuple. To create a tuple with a single element, a comma should be included after the element as shown above. If you check the data type of the variable ‘tuple_4’, it is a tuple and not a string.

3. Accessing Values in a Tuple

Accessing values in tuples is similar to lists. Consider the tuple that contains 5 elements 15, 20, 96, 32 and 17.

```
>>> my_tuple = (15, 20, 96, 32, 17)
>>> my_tuple[0]
15
>>> my_tuple[4]
17
```

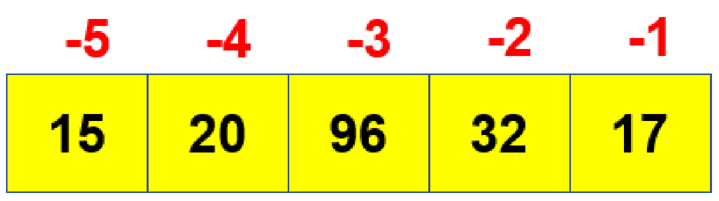
To create the tuple we code, ‘my_tuple = (15, 20, 96, 32, 17)’. The parentheses are optional. Similar to lists, tuples are also indexed starting from 0.



Hence, when we call my_tuple[0], it will output the first element in the tuple which is 15. Similarly, if we want to access 17 which is at index 4, we call my_tuple[4]. We get the expected output which is 17.

3.1 Negative Indices

Negative indices also work similar to lists. We have learnt in course 1 that in negative indexing, the last element is indexed -1 and the values are indexed from right to left as shown in the diagram below.



Look at the sample code below.

```
>>> my_tuple = (15, 20, 96, 32, 17)
>>> my_tuple[-1]
17
>>> my_tuple[-3]
96
```

When we call my_tuple[-1], we get the last element 17 as the output. Similarly, my_tuple[-3] will return the element at index -3 which is 96.

3.2 Slicing

Slicing means extracting a part of a sequence. Slicing is the same as what we learnt in the lesson on lists.

```
>>> my_tuple = (15, 20, 96, 32, 17)
>>> my_tuple[0:3]
(15, 20, 96)
>>> my_tuple[2:4]
(96, 32)
```

What will happen when we call my_tuple[0:3]. It will output the values from index 0 to index 2. Remember, if the range is from m to n, the values considered will be from index m to n-1. Therefore, the output will contain 15, 20 and 96. Similarly, if the range is from 2 to 4, the values at indices 2 and 3 will be outputted.

3.3 Nested Tuples

Nested tuples are similar to nested lists. Let us create a nested tuple first.

```
>>> my_tuple = ((1, 2), (‘a’, ‘b’))
>>> my_tuple[0][1]
2
>>> my_tuple[1]
(‘a’, ‘b’)
```

This variable ‘my_tuple’ contains two tuples inside another tuple. The first inner tuple contains the values 1 and 2 while the second inner tuple contains the values ‘a’ and ‘b’. Suppose we want to access the value 2. First, we need to access the inner tuple at index 0 and inside that inner tuple, 2 is located at index 1. Hence, we call my_tuple [0][1] where the 0 points to the inner tuple at index 0 and 1 points to the element at index 1 inside that inner tuple.

Similarly, if we want to access the inner tuple that contains the values ‘a’ and ‘b’, we know that it is located at index 1. Hence, calling my_tuple[1] will output the entire tuple at index 1 as shown above.

4. Tuple Packing and Unpacking

Tuple packing is nothing but assigning values into a tuple. In the following example, we are assigning the values ‘car’, ‘pen’ and ‘ice’ into a tuple. This is called tuple packing.

```
>>> my_tuple = (‘car’, ‘pen’, ‘ice’)
```

Tuple unpacking is extracting the content of a tuple into variables. Consider the following example.

```
>>> a, b, c = my_tuple
>>> b
‘pen’
```

We have three variables ‘a’, ‘b’ and ‘c’ on the left-hand side of the equal sign and the tuple ‘my_tuple’ on the right-hand side. The tuple on the right-hand side contains three values ‘car’, ‘pen’ and ‘ice’. The first line of code assigns the content of the tuple to the variables on the left-hand side. Hence the string ‘car’ will be assigned to variable ‘a’, the string ‘pen’ will be assigned to variable ‘b’ and the string ‘ice’ will be assigned to the variable ‘c’. To confirm let us print the variable ‘b’. It outputs the string ‘pen’ as expected.

One condition when using tuple unpacking is the number of variables on the left-hand side should be equal to the number of elements in the tuple. In the previous example, ‘my_tuple’ contains three elements. Hence, we have three variables on the left-hand side.

Exercise

Run the following code and analyze the output.

```
>>> my_tuple = (‘car’, ‘pen’, ‘ice’)
>>> a, b = my_tuple
```

5. Insertion, Deletion, and Update

Let us run the following code.

```
>>> my_tuple = (15, 20, 96, 32, 17)
>>> my_tuple[0] = 8
```

It will output the following error message.

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    my_tuple[0] = 8
TypeError: 'tuple' object does not support item assignment
```

The reason for this error is because tuples are immutable. It means once a tuple is created, it cannot be modified. Hence, a tuple does not support insert, delete, and update operations. Unlike tuples lists are mutable. We can insert, delete, and update values in a list.

6. Lists vs Tuples

Lists and tuples are very similar to each other. However, lists are more versatile than tuples. You can update the values in a list and expand or shrink a list. But you cannot do any of those with a tuple. On top of that lists have more built-in functions than tuples.

Since lists have all these advantages over tuples, how are tuples useful? Tuples are memory efficient than lists which means they consume less memory.

```
>>> import sys
>>> my_list = [1,2,3,4,5]
>>> my_tuple = (1,2,3,4,5)
>>> sys.getsizeof(my_list)
104
>>> sys.getsizeof(my_tuple)
88
```

Let us create a tuple and a list that contain the same set of elements and measure the amount of memory consumed by each of them. We will use a python function called getsizeof(). It will return the amount of memory consumed in bytes. To use this function, we need the sys module. Hence, sys module is imported. When we run sys.getsizeof(my_list), we get the output 104. Which means our list has consumed 104 bytes of memory. Let us do the same for our tuple as well. We can see that it has consumed only 88 bytes of memory which is 15% less compared to the list.

The reason why tuples consume less memory than tuples will not be discussed here since it is an advanced topic to discuss at this point. However, if you are interested, the following link will help you understand clearly why tuples consume less memory than lists.

Link: https://stackoverflow.com/questions/46664007/why-do-tuples-take-less-space-in-memory-than-lists

If the data need not to be changed in the future, use tuples since it will save a significant amount of memory. If the data needs to be modified in the future, you must use lists since tuples are immutable.

7. Tuple Operations

This section discusses some common tuple operations.

- **Length**

To find the size/length of a tuple, the function 'len' (stands for length) can be used. See the following example.

```
>>> len((1,2,3))
3
```

The length of the tuple is 3 as it contains 3 values.

- **Concatenation**

```
>>> a = (1,2,3)
>>> b = (4,5,6)
>>> a+b
(1, 2, 3, 4, 5, 6)
```

Suppose there are two tuples a and b. The contents of tuples a and b can be combined using the plus(+) operator. a+b will output a single tuple with contents from tuples a and b. In the output of the above example, values 1, 2, and 3 are from tuple a, and values 4, 5, and 6 are from tuple b.

- **Repetition**

The following code illustrates how repetition works.

```
>>> print((‘Hi’) * 4)
(‘Hi’, ‘Hi’, ‘Hi’, ‘Hi’)
```

Note that in the above example, multiplying by 4 does not create 4 separate tuples but a single tuple where the contents of the original tuple are multiplied 4 times.

- **Membership**

Membership checks whether a value is available in a tuple. See the following example.

```
>>> print(3 in (1,2,3))
True
```

The 'in' operator is used to check membership. Statement '3 in (1,2,3)' checks whether value 3 is available in the tuple. Since the value is available, it returns True. If the value is not available, it will return False.

- **Iteration**

Iteration means going through the tuple one element at a time.

```
>>> for x in (1,2,3):
        print(x)
```

```
1
2
3
```

The first element in the tuple, which is 1 in the example above, will be assigned to the variable x. Then the print(x) statement will be executed. After that, the second value in the tuple, which is 2, will be assigned to the variable x. The print(x) statement will be executed again. This pattern continues until the last element in the tuple.

--- End of the document---

◀ 1.3 Quiz

Jump to...