Tuples, Dictionaries, And Sets

Data Structures In Python

One can think of a data structure as a means of organizing and storing data such that it can be accessed and modified with ease. We have already seen the most conventional data types like strings and integers. Now let us take a deeper dive into the more unconventional data structures— **Tuples, Dictionaries, and Sets**.

Tuples

- A tuple is an ordered collection of elements/entries.
- Objects in a Tuple are immutable i.e. they cannot be altered once created.
- In a tuple, elements are placed within parentheses, separated by commas.
- The elements in a tuple can be of different data types like integer, list, etc.

How To Create A Tuple?

- A variable is simply assigned to a set of comma-separated values within closed parentheses.

```
>>> a=(1,2)
>>> b=("eat","dinner","man","boy")
>>> print(a)
(1,2)
>>> print(type(a))
<class 'tuple'>
```

- "()" parentheses are optional. If we assign a variable to a set of comma-separated values without parentheses, then by default, Python interprets it as a Tuple; This way of creating a Tuple is called **Tuple Packing**. It is always a good practice to use parentheses. This is shown in the given code snippet:

```
>>> c= 1,2,3,4,5
>>> print(c)
(1,2,3,4,5)
>>> print(type(c))
<class 'tuple'>
```

Note: If we assign a set of comma separated elements (or objects) to a set of comma separated variables, then these variables are assigned the corresponding values. **For Example:**

```
>>> e, f= "boy","man"
>>> print(e)
boy
>>> print(f)
man
```

Creating an Empty Tuple

In order to create an empty Tuple, simply assign a closed parentheses , which doesn't contain any element within, to a variable. This is shown below:

```
a=()
b=()
#Here 'a' and 'b' are empty tuples.
```

Creating A Tuple With A Single Element

- Creating a tuple with a single element is slightly complicated. If you try to create such a tuple by putting a single element within the parentheses, then it would not be a tuple, it would lead to the assignment of a single element to a variable.

```
>>> a=("Hamburger")
>>> print(type(a))
<class 'str'>
```

Here **'a'**, has a type **'string'**.

- If you try to create a tuple using the keyword `tuple`, you will get:

```
>>> tuple("Hamburger")
('H', 'a', 'm', 'b', 'u', 'r', 'g', 'e', 'r')
```

- To create such a tuple, along with a single element inside parentheses, we need a trailing comma. This would tell the system that it is in fact, a tuple.

```
>>> a=("Hamburger",)
>>> print(type(a))
<class 'tuple>
#Here a is indeed a tuple. Thus, the trailing comma is important in such an assignment.
```

Difference Between A Tuple and A List

Tuple	List
A Tuple is immutable	A list is mutable
We cannot change the elements of a Tuple once it is created	We can change the elements of a list once it is created
Enclosed within parentheses “( )”	Enclosed within square brackets “[ ]”

Accessing Elements of a Tuple

Indexing in a Tuple

- Indexing in a Tuple starts from index **0**.
- The highest index is the `NumberOfElementsInTheTuple-1`**.**
- So a tuple having **10** elements would have indexing from **0-9**.
- This system is just like the one followed in Lists.

Negative Indexing

- Python language allows negative indexing for its tuple elements.
- The last element in the tuple has an index **-1**, the second last element has index **-2**, and so on.

Let us define a Tuple:

```
myTemp=(1,15,13,18,19,23,4,5,2,3) #Number of elements=10
```

Element	1	15	13	18	19	23	4	5	2	3
Index	0	1	2	3	4	5	6	7	8	9
Negative indexing	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> print(myTemp[0])
1
>>> print(myTemp[8])
2
>>> print(myTemp[-1])
3
```

If any index out of this range is tried to be accessed, then it would give the following error.

```
>>> print(myTemp[10])
IndexError: tuple index out of range
>>> print(myTemp[20])
IndexError: tuple index out of range
>>> print(myTemp[-100])
IndexError: tuple index out of range
```

Slicing of a Tuple

- We can slice a Tuple and then access any required range of elements. Slicing is done by the means of a slicing operator which is ":".
- The basic format of slicing is:

```
<Name of Tuple>[start index: end index]
```

- This format by default considers the end index to be `endIndex-1`

```
>>> print(myTemp[1:4])
(15,13,18)
>>> print(myTemp[7:])
(5,2,3)
```

What Changes Can Be Made To A Tuple?

A tuple is immutable. This means that the objects or elements in a tuple cannot be changed once a tuple has been initialized. This is contrary to the case of lists - as a list is mutable, and hence the elements can be changed there.

- Python enables us to reassign the variables to different tuples.
- **For example**, the variable `myTemp` which has been assigned a tuple. To this variable some other tuple can be reassigned.

```
>>> myTemp = ([6,7],"Amity", 2, "boy")
```

- However, you cannot change any single element in the tuple.

```
>>> myTemp[2] = "Amity"
TypeError: 'tuple' object does not support item assignment
```

- Though, you can change items from any mutable object in the tuple.

```
>>> myTemp[0][1] = "Amity"
>>> print(myTemp)
([6,"Amity"],"Amity", 2, "boy")
```

- If you want to delete a tuple entirely, such an operation is possible.
- The keyword used is `del`

```
>>> del(myTemp)
>>> print(myTemp)
NameError: name 'myTemp' is not defined
# The name error shows that the tuple 'myTemp' has been deleted
```

Tuples Functions

- **We can use loops to iterate through a tuple:** For example, if we want to print all the elements in the tuple. We can run the following loop.

```
myTemp=(1,2,3)
for t in myTemp:
      print(t)
```

Output:

```
1
2
3
```

- **Checking whether the tuple contains a particular element:** The keyword used is `in`. We can easily get a boolean output using this keyword. It returns `True` if the tuple contains the given element, else the output is `False`.

```
>>> 10 in myTemp
False
>>> 4 in myTemp
True
```

- **Finding the length of a tuple:** We can easily find the number of elements that any tuple contains. The keyword used is `len`.

```
>>> print(len(myTemp))
6
```

- **Concatenation:** We can add elements from two different tuples and form a single tuple out of them. This process is concatenation and is similar to data types like string and list. We can use the **" + "** operator to combine two tuples.

```
a = (1,2,3,4)
b = (5,6,7,8)
d = a+b
print(d)
Out[]: (1,2,3,4,5,6,7,8)
```

We can also combine two tuples into another tuple in order to form a nested tuple. This is done as follows:

```
a = (1,2,3,4)
b = (5,6,7,8)
d = (a, b)
print(d)
--> ((1,2,3,4),(5,6,7,8))
# Here d is a nested tuple which is formed from 2 different tuples.
```

- **Repetition of a Tuple:** We can repeat the elements from a given tuple into a different tuple. The operator used is **" * "**, the multiplication operator. In other words, by using this operator, we can repeat the elements of a tuple as many times as we want.

```
>>> a = (1,2,3,4)
>>> print(a*4)
(1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4)
# The same tuple elements are repeated 4 times in the form of another
tuple.
```

- **Finding minimum or maximum element in a tuple:** To find the maximum element in a tuple, the keyword used is `max` and for finding the minimum element, the keyword used is `min`. These keywords return the maximum and the minimum elements of the tuple, respectively.

```
>>> a = (1,2,3,4)
>>> print(min(a))
1
>>> print(max(a))
4
```

Note: We can find the minimum and maximum of only those tuples, which have comparable entries. We cannot compare two different data types like a string and a tuple. Such comparisons would throw an error.

For example:

```
>>> s= (1,2,"string",4)
>>> print(min(s))
TypeError: '<' not supported between instances of 'string' and 'int'
```

```
>>> e= (1,2,2.6,4)
>>> print(min(e))
1
# Even though the data types are different , however since a float can be compared to a floating point value, hence this operation does not give an error.
```

- **Converting a list to a tuple:** We can typecast a list into a tuple. The keyword used is `tuple`. This is done as follows:

```
>>> myList= [1,2,3,4]
>>> myTuple= tuple(myList)
>>> print(myTuple)
(1,2,3,4)
```

Using Tuples For Variable Length Input And Output

Variable Length Inputs

There are some situations where we need to give a variable number of inputs to some functions. The use of tuples in such situations has proved to be highly efficient.

Task 1: Giving a variable number of inputs and printing them:

```
def printNum(a, b,*more):
      print(a)
      print(b)
      print(more)
printNum(1,2,3,4,5,5)
```

Output

```
1
2
(3,4,5,5)
```

- We use `*more` as the third parameter.
- The first two arguments are taken as the first two parameters and hence are printed individually. However, all the arguments after them, are taken as a single tuple and hence are printed in the form of a tuple.

Task 2: Finding the sum of a variable number of inputs:

Consider an example in which we have to calculate the sum of a variable number of inputs. In such a situation we cannot practically have multiple parameters in the function. This can be done as follows:

```
def printNum(a, b,*more):
      sum=a+b
      for t in more: #Traverse the tuple *more
            sum=sum+t #Add all elements in *more
      return sum
printNum(1,2,3,4,5,5)
Out[]: 20
```

Variable Length Outputs

- Following the conventional ways, we can return only a single value from any function. However, with the help of tuples, we can overcome this disadvantage.
- Tuples help us in returning multiple values from a single function.
- This can be done by returning comma-separated-values, from any function.
- On being returned, these comma-separated values act as a tuple.
- We can access the various entries from this returned tuple. This can be shown as:

```
def sum_diff(a, b):
      return a+b, a-b #Return the sum and difference together
print(sum_diff(1,2))
Out[]: (3,-1)
```

Dictionaries

- A Dictionary is a Python's implementation of a data structure that is more generally known as an associative array.
- A dictionary is an unordered collection of key-value pairs.
- Indexing in a dictionary is done using these **"keys".**
- Each pair maps the key to its value.
- Literals of the type dictionary are enclosed within curly brackets.
- Within these brackets, each entry is written as a key followed by a colon " : ", which is further followed by a value. This is the value that corresponds to the given key.
- These keys must be unique and cannot be any immutable data type. **Eg-** string, integer, tuples, etc. They are always mutable.
- The values need not be unique. They can be repetitive and can be of any data type (Both mutable and immutable)
- Dictionaries are mutable, which means that the key-value pairs can be changed.

What does a dictionary look like?

This is the general representation of a dictionary. Multiple key-value pairs are enclosed within curly brackets, separated by commas.

```
myDictionary = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Creating A Dictionary

Way 1- The Basic Approach

Simply put pairs of key-value within curly brackets. Keys are separated from their corresponding values by a colon. Different key-value pairs are separated from each other by commas. Assign this to a variable.

```
myDict= {"red":"boy", 6: 4, "name":"boy"}
months= {1:"January", 2:"February", 3: "March"}
```

Way 2- Type Casting

This way involves type casting a list into a dictionary. To typecast, there are a few mandates to be followed by the given list.

- The list must contain only tuples.
- These tuples must be of length **2**.
- The first element in these tuples must be the key and the second element is the corresponding value.

This type casting is done using the keyword **`dict`.**

```
>>> myList= [("a",1),("b",2),("c",2)]
>>> myDictionary= dict(myList)
>>> print(myDictionary)
{"a":1,"b":2,"c":2}
```

Way 3- Using inbuilt method `.copy()`

We can copy a dictionary using the method **`.copy()`**. This would create a shallow copy of the given dictionary. Thus the existing dictionary is copied to a new variable. This method is useful when we wish to duplicate an already existing dictionary.

```
>>> myDict= {"a":1,"b":2,"c":2}
>>> myDict2= myDict.copy()
>>> print(myDict2)
{"a":1,"b":2,"c":2}
```

Way 4- Using inbuilt method `.fromkeys()`

This method is particularly useful if we want to create a dictionary with variable keys and all the keys must have the same value. The values corresponding to all the keys is exactly the same. This is done as follows:

```
>>> d1= dict.fromkeys(["abc",1,"two"])
>>> print(d1)
{"abc":None ,1: None, "two": None}
# All the values are initialized to None if we provide only one argument
i.e. a list of all keys.
```

```
# We can initialise all the values to a custom value too. This is done by
providing the second argument as the desired value.
>>> d2= dict.fromkeys(["abc",1,"two"],6)
>>> print(d2)
["abc":6 ,1:6, "two":6]
```

How to access elements in a dictionary?

We already know that the indexing in a dictionary is done with the keys from the various key-value pairs present within. Thus to access any value we need to use its index i.e. it's **key**.

Similar to the list and tuples, this can be done by the square bracket operator **[ ]**.

```
foo= {"a":1,"b":2,"c":3}
print(foo["a"])
--> 1
print(foo["c"])
--> 3
```

If we want the value corresponding to any key , we can even use an inbuilt dictionary method called **`get`**.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a"))
1
>>> print(foo.get("c"))
3
```

A very unique feature about this method is that , incase the desired **key** is not present in the dictionary , it won't throw an error or an exception. It would simple return **`None`**.

We can make use of this feature in another way. Say we want the method to do the following action: If the key is present in the dictionary then return the value corresponding to the key. In case the key is not present, return a custom desired value (say 0).

This can be done as follows:

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a",0))
1
>>> print(foo.get("d",0))
0
```

Accessing all the available keys in the dictionary:

This can be done using the method **`.keys()`**. This method returns all the different keys present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.keys()
dict_keys(["a","b","c"])
```

Accessing all the available values in the dictionary:

This can be done using the method **`.values()`**. This method returns all the different values present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.values()
dict_values([1,2,3])
```

Accessing all the available items in the dictionary:

This can be done using the method `.items()`. This method returns all the different items (key-value pairs) present in the dictionary in the form of a list of tuples, with the first element of the tuple as the key and the second element as the value corresponding to this key .

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.items()
dict_items([("a",1),("b",2),("c",3)])
```

Checking if the dictionary contains a given key:

The keyword used is `in`. We can easily get a boolean output using this keyword. It returns True if the dictionary contains the given **key**, else the output is False. This checks the presence of the keys and not the presence of the values.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> "a" in foo
True
>>> 1 in foo
False
```

Iterating over a Dictionary:

In order to traverse through the dictionary, we can use a simple for loop. The loop will go through the keys of the dictionary one by one and do the required action.

```
bar= {2:1,3:2,4:3}
for t in bar:
      print(t)
Out[]:
2
3
4
# Here t is the key in the dictionary and hence when we print t in all
iterations then all the keys are printed.
```

```
for t in bar:
      print(t, bar[t])
Out[]:
2 1
3 2
4 3
# Here along with the keys, the values which are bar[t] are printed. In
this loop, the values are accessed using the keys.
```

Adding Elements In a Dictionary

Since a dictionary is mutable, we can add or delete entries from the dictionary. This is particularly useful if we have to maintain a dynamic data structure. To assign a value corresponding to a given key *(This includes over-writing the value present in the key or adding a new key-value pair)*, we can use the square bracket operators to simply assign the value to a key.

If we want to update the value of an already existing key in the dictionary then we can simply assign the new value to the given key. This is done as follows:

```
>>> bar= {2:1,3:2,4:3}
>>> bar[3]=4
# This operation updates the value of the key 3 to a new value i.e. 4.
>>> print(bar)
{2:1,3:4,4:3}
```

Now if we want to add a new key-value pair to our dictionary, then we can make a similar assignment. If we have to add a key-value pair as `"man":"boy"`, then we can make the assignment as:

```
>>> bar["man"]="boy"
>>> print(bar)
{2:1,3:2,4:3,"man":"boy"}
```

Adding or concatenation of two dictionaries:

If we have 2 dictionaries and we want to merge the contents of both the dictionaries and form a single dictionary out of it . It is done as follows:

```
a= {1:2,2:3,3:4}
b= {7:2,10:3,6:4}
a.update(b)
print(a)
--> {1:2,2:3,3:4,7:2,10:3,6:4}
```

In this process, the second dictionary is unchanged and the contents of the second dictionary are copied into the first dictionary. The uncommon keys from the second dictionary are added to the first with their corresponding values. However, if these dictionaries have any common key, then the value of the common key present in the first dictionary is updated to the new value from the second.

Deleting an entry:

In order to delete an entry corresponding to any key in a dictionary , we can simple pop the key from the dictionary. The method used here is `.pop()` . This method removes the key-value pair corresponding to any particular key and then returns the value of the removed key-value pair.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.pop(2)
(3,23,3)
```

Deleting all the entries from the dictionary:

If we want to clear all the key-value pairs from the given dictionary and thus convert it into an empty dictionary we can use the `.clear()` method.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.clear()
>>> print(c)
{}
```

Deleting the entire dictionary:

We can even delete the entire dictionary from the memory by using the `del` keyword. This would remove the presence of the dictionary. This is similar to tuples and lists.

Problem statement: Print all words with frequency k.

Approach to be followed:

First, we convert the given string of words into a list containing all the words individually. Some of these words are repetitive and to find all the words with a specific frequency, we convert this list into a dictionary with all the unique words as keys and their frequencies or the number of times they occur as their values.

To convert the string to a list, we use the `.split()` function. This gives us a list of words. Now, we run a loop through this list and keep making changes to the frequency in the dictionary. If the word in the current iteration already exists in the dictionary as a key, then we simply increase the value(or frequency) by 1. If the key does not exist, we create a new key-value pair with value as 1.

Now we have a dictionary with unique keys and their respective frequencies. Now we run another loop to print the keys with the frequency '**k**'.

Given below is a function that serves this purpose.

```
def printKFreqWords(string, k):
      # Converting the input string to a list
      myList= string.split()
      # Initialise an empty dictionary
      dict= {}
      # Iterate through the list in order to find frequency
      for i in myList:
            dict[i]=dict[i]+1
      else:
            dict[i]=1
      # Loop for printing the keys with frequency as k
      for t in dict:
            if dict[t]==k:
                  print(t)
```

Sets

Mathematically a set is a collection of items *(not in any particular order)*. A Python set is similar to this mathematical definition with below additional conditions.

- The elements in a set cannot be repeated i.e. an element can occur only once.
- The elements in the set are immutable*(cannot be modified)* but the set as a whole is mutable.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference, and complement, etc. We can create a set, access its elements, and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the `set()` function or placing all the elements within a pair of curly braces, separated by commas.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Note: The order of the elements has changed in the result.

Accessing Values in a Set

We cannot access individual values in a set as there is no specific order of elements in a set. We can only access all the elements together as shown. We can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
for d in Days:
      print(d)
```

When the above code is executed, it produces the following result.

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using the `add()` method. Again as discussed there is no specific index attached to the newly added element.

```
>>> Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
>>> Days.add("Sun")
>>> print(Days)
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using the `discard()` method. Again as discussed there is no specific index attached to the newly added element.

```
>>> Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
>>> Days.discard("Sun")
>>> print(Days)
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element `"Wed"` is present in both the sets. The union operator is **'|'**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB #Union of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element `"Wed"` is present in both the sets. The intersection operator is **"&"**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB #Intersection of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element `"Wed"` is present in both the sets so it will not be found in the result set. The operator used is **"-"**.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB #Difference of Sets
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one `"wed"`.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB #Check Subset
SupersetRes = DaysB >= DaysA #Check Superset
print(SubsetRes)
print(SupersetRes)
```

When the above code is executed, it produces the following result.

```
True
True
```