Tuples and Sets

Data Collections

- Python has four built-in data types to store collection of data such as Lists, Tuples, Sets and Dictionaries, all with different qualities and usage.
- Tuples are used to store multiple items in a single variable.
- A tuple is a collection of data items which is ordered and unchangeable.
- Tuples are Immutable means once the tuple is created its values cannot be changed.
- Tuples are created by using round brackets or by using tuple() constructor
- Syntax:
  tuple_name = (item_1, item_2, item_3, ……., item_n)
- Tuples items are ordered, Unchangeable.
- Tuples allows duplicate values
- Example:
  # creating tuple to contain multiple values of different types
  Mytuple=(“python”,23,4.5)

  print(type(Mytuple)) # output will be <class, ‘tuple’>

- Empty tuple can be created using ()
  Ex: tuple1=() # Creates empty tuple
- We can create tuples without using () as shown in below example
  tuple2="abc",1,2,3.5
  print(tuple2) # Output: ('abc', 1, 2, 3.5)
- To create a tuple with single value the following syntax has to be followed
  Tuple3=(“laxmi”,) # Here comma is must to specify it as tuple otherwise
  # it treated as string
- It is also possible to use the tuple() constructor to make a tuple.
- Example
  thistuple = tuple(("apple", "banana", "cherry"))
  # the double round-brackets is must because tuple() takes only one argument
  print(thistuple)

Accessing Tuples:

Since tuple items are ordered, they can be accessed by their index. The first item has index 0 and the consecutive items have index 1, 2, 3 so on. It is also possible to use negative indices to access tuple items. The index -1 corresponds to the last element, -2 corresponds to the second last element and so on. Below is an example showing this.

```
tpl = ("apple", "banana", "mango", "orange")
print(tpl[0])
print(tpl[1])
print(tpl[-1])
print(tpl[-2])
```

The above code will output:

apple

banana

orange

mango

- Each item in the tuple can be accessed individually through indexing.
- Each item in the tuple is numbered from 0 i.e, the first item is numbered as 0, second item as 1 and so on.
- Square brackets [ ] are used by tuples to access individual items
- The syntax for accessing an item in a tuple is,

  tuple_name[index]

  where index should always be an integer value and indicates the item to be selected
- In addition to positive index numbers, you can also access tuple items using a negative index number, by counting backwards from the end of the tuple, starting at −1.
- Negative indexing is useful if you have a large number of items in the tuple and you want to locate an item towards the end of a tuple.
- Example:

“v”	“i”	“b”	“g”	“y”	“o”	“r”
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

The above table shows how tuple items are numbered with positive indexing and negative indexing.

- Example: colors=(“v”,”i”,”b”,”g”,”y”,”o”,”r”)
  print(colors[4]) # prints the item “y”
  print(colors[-4]) # prints the item “g”

Slicing :

- Slicing of tuples is allowed in Python wherein a part of the tuple can be extracted by specifying an index range along with the colon (:) operator, which itself results as tuple type.
- Syntax:
  tuple_name[start:stop[:step]]
  #Colon is used to specify range value
- where both start and stop are integer values (positive or negative values).
- Tuple slicing returns a part of the tuple from the start index value to stop index value, which includes the start index value but excludes the stop index value.
- The step specifies the increment value to slice by and it is optional.
- Example program:
  #Accessing tuple elements using slicing

```
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[:])
```

Built-In Functions Used on Tuples:

There are many built-in functions for which a tuple can be passed as an argument.

1) len() :The len() function returns the numbers of items in a tuple.
2) sum() : The sum() function returns the sum of numbers in the tuple.
3) sorted() :The sorted() function returns a sorted copy of the tuple as a list while leaving the original tuple untouched.
4) max() : Returns the maximum item of the tuple.
5) min() : Returns the minimum item of the tuple.

Example:

```
years = (1987, 1985, 1981, 1996)
print(len(years)) # prints 4
print(sum(years)) # prints the sum of items of tuples i.e, 7949
sorted_years = sorted(years)
print(sorted_years)  # [1981, 1985, 1987, 1996]
```

Basic operation on Tuples:

- The + operator to concatenate tuples together and the * operator to repeat a sequence of tuple items.

  Example:

```
tuple_1 = (2, 0, 1, 4)
tuple_2 = (2, 0, 1, 9)
print(tuple_1 + tuple_2) #prints (2, 0, 1, 4, 2, 0, 1, 9)
print(tuple_1 * 3) # (2, 0, 1, 4, 2, 0, 1, 4, 2, 0, 1, 4)
```

- Two tuples can be compared by using the comparison operators like <,<=,>,>=,==,!=. Tuples are compared position by position. Result of comparison is either True or False.
  Example :
  tuple_1 = (9, 8, 7).
  tuple_2 = (9, 1, 1)
  print(tuple_1 > tuple_2) # True
  print(tuple_1 != tuple_2) # True
  print(tuple1_==tuple_2) # Returns False as two tuples are not identical
- Presence of an item in a tuple can be tested using in and not in membership operators. It returns a Boolean True or False.
  For example,
  tuple_items = (1, 9, 8, 8)

  print(1 in tuple_items) # Returns True

  print(25 in tuple_item) # Returns False

Tuple Methods:

1) tuple_name.count(item)
   The count() method counts the number of times the item has occurred in the tuple and returns it.

2) tuple_name.index(item) The index() method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then ValueError is thrown by this methodTuple Methods

   Example:
   my_tuple = ('a', 'p', 'p', 'l', 'e',)
   print(my_tuple.count('p')) # Output: 2
   print(my_tuple.index('l')) # Output: 3

   Tuple Packing and Unpacking
   The statement t = 12345, 54321, 'hello!' is an example of tuple packing.
   t = 12345, 54321, 'hello!'
   print(t) # (12345, 54321, 'hello!')

The values 12345, 54321 and 'hello!' are packed together into a tuple.

Tuple Unpacking: Extracting the values of tuple back into variables is called tuple unpacking.

For example,

```
x, y, z = t
print(x,y,z) # prints the output as : 12345,54321,hello
```

Traversing of Tuples

You can iterate through each item in tuples using for loop.

Example:

```
for i in ('laxmi','vedant','Gautami'):
   print("hello" +" " + i)
```

Output :

```
hello laxmi
hello vedant
hello Gautami
```

Populating Tuples with Items

You can populate tuples with items using += operator.

**Program to Populate Tuple with User-Entered Items

```
tuple_items = ()
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
        user_input = int(input("Enter a number: "))
        tuple_items += (user_input,)
print(f"Items added to tuple are {tuple_items}")
```

OUTPUT

```
Enter the total number of items: 4
 Enter a number: 4
Enter a number: 3
Enter a number: 2
 Enter a number: 1
Items added to tuple are (4, 3, 2, 1)
```

Write Python Program to Swap Two Numbers Without Using Intermediate/Temporary Variables. Prompt the User for Input

```
a = int(input("Enter a value for first number "))
b = int(input("Enter a value for second number "))
print("Before Swapping")
print(f"Value for first number {a}")
print(f"Value for second number {b}")
b, a = a, b
print("After Swapping")
print(f"Value for first number {a}")
print(f"Value for second number {b}")
```

Output:

```
Enter a value for first number 34
Enter a value for second number 23
Before Swapping
Value for first number 34
Value for second number 23
After Swapping
Value for first number 23
Value for second number 34
```

SETS

- A set is a collection which is unordered, unchangeable (i.e. it is not possible to edit set elements) and disallows duplicate members. However, unlike tuples, it is possible to add elements to or remove elements from a set.
  Sets are created as follows using {}
  s = {1,2,3}
  Or
  s = {"apple", "banana", "orange"}
  Or even
  s = {"apple", "banana", 3, 4}
- If duplicates are included while defining a set, they are considered only once.
  Consider the below example:
  s1 = {2,3,2,4,3,1,2,3}

s2 = {1,2,3,4}
print(s1==s2)
The above code will give the output: True
We could also have sets inside a set as follows:
s = {{1,2},{2,3}}
Or
s = {{1,2},2}

- The empty set can be created using the set() constructor.
  Ex: s=set() # Created an empty set with no elements
  S1={} # Creates an empty dictionary not a set
  The statement type({}) will output: <class 'dict'>
  But the statement type(set()) will output: <class 'set'>
- A set could also be created from tuples, lists, dictionaries and strings using the set() constructor. Here's an example:
  t = (1,2,3) # tuple
  l = [1,2,3] # list
  d = {"brand": "Ford", "model": "Mustang"} # dictionary
  s = "hello"
  print(set(t))
  print(set(l))
  print(set(d))
  print(set(s))
  The above code will output:
  {3, 2, 1}
  {2, 3, 1}
  {'model', 'brand'}
  {'h', 'l', 'o', 'e'}
- The length of a set can be obtained using len() function. An example is shown below:
  s = {1,2,3,4}
  print(len(s))
  The above code will output: 4
- We can check if a specified item is in a set using the in operator. Below is an example:
  s = {"apple", "orange", "banana"}
  print("orange" in s)
  The above code will output: True

Accessing Set Items

- Since sets are unordered, their elements won't have any indices. Hence it is not possible to access set elements using an index like in tuples.
- However it is possible to loop through the elements of a set using for loop. An example is shown below:
  for city in {"Bangalore", "Mysore", "Chennai"}
  print(city)
  The above code will output:
  Bangalore
  Chennai
  Mysore
  Note: Again due to the lack of a defined order for a set, every time the code is executed, the cities can be printed in a different order.

Adding items to set:

- Items can be added to a set using the inbuilt method add(). Below is an example:
  s = {"apple", "orange", "banana"}
  s.add("mango")
  print(s)
  The above code will give the output: {'apple', 'mango', 'banana', 'orange'}
- We can add more than one item in the set using update method. The update() accepts only iterable argument.
  For Ex:
  S={“January”,”March”}
  S.update((“February”, “April”, “May”)) # 3 items get added into set S

Removing items from set:

- Items can be removed from a set using the inbuilt method **remove().**
- **remove()** gives an KeyError if specified item to be removed is not present in the set.
- Below is an example:
  s = {"apple", "orange", "banana"}
  s.remove("banana")

print(s)
The above code will give the output: {'apple', 'orange'}

- Items can also be removed using **discard().** It is same as remove() but doesnot throw keyerror if the item is not in set.
- pop() method can be used to remove an item from the set. Normally pop() removes last item but as set is unordered we can't determine the element to be popped.
- clear() method removes all the items from the set.

More Set Methods

- The union of two sets can be taken using the inbuilt function union() or by | operator. Below is an example:
  x = {"apple", "orange", "melon"}
  y = {"melon", "mango", "kiwi"}
  z = x.union(y) or z=x|y
  print(z)
  The above code will give the output: {'apple', 'orange', 'melon', 'mango', 'kiwi'}
- The intersection of two sets can be taken using the inbuilt method intersection() or by & operator
  Below is an example:
  x = {"apple", "orange", "banana", "mango"}
  y = {"banana", "mango", "strawberry"}
  z = x.intersection(y) or z=x&y
  print(z)
  The above code will give the output: {'banana', 'mango'}
- The difference between two sets can be obtained using the inbuilt method difference() or by using – operator.

  Below is an example:
  x = {"apple", "orange", "cherry", "banana"}
  y = {"cherry", "banana", "melon"}
  z = x.difference(y) z=x-y # z contains only the items present in set x.
  print(z)
  The above code will give the output: {'apple', 'orange'}
- The symmetric difference of two sets is calculated by using built-in method symmetric_difference or by ^ operator.

z=x.symmetric_difference(y) or z=x^y # it removes the common items present in both the sets

print(z) # {“apple”,”orange”,”melon”}

Various Set Methods

Set Methods	Syntax	Description
add()	set_name.add(*item*)	The *add()* method adds an *item* to the set *set_name*.
clear()	set_name.clear()	The *clear()* method removes all the items from the set *set_name*.
difference()	set_name.difference(**others*)	The *difference()* method returns a new set with items in the set *set_name* that are not in the *others* sets.
discard()	set_name.discard(item)	The *discard()* method removes an *item* from the set *set_name* if it is present.
intersection()	set_name.intersection(**others*)	The *intersection()* method returns a new set with items common to the set *set_name* and all *others* sets.
isdisjoint()	set_name.isdisjoint(*other*)	The *isdisjoint()* method returns True if the set *set_name* has no items in common with *other* set. Sets are disjoint if and only if their intersection is the empty set.
issubset()	set_name.issubset(*other*)	The *issubset()* method returns True if every item in the set *set_name* is in *other* set.
issuperset()	set_name.issuperset(*other*)	The *issuperset()* method returns True if every element in *other* set is in the set *set_name*.
pop()	set_name.pop()	The method *pop()* removes and returns an arbitrary item from the set set_name. It raises KeyError if the set is empty.
remove()	set_name.remove(*item*)	The method *remove()* removes an *item* from the set *set_name*. It raises KeyError if the *item* is not contained in the set.
symmetric_ difference()	set_name. symmetric_difference(other)	The method *symmetric_difference()* returns a new set with items in either the set or other but not both.
union()	set_name.union(**others*)	The method *union()* returns a new set with items from the set *set_name* and all *others* sets.
update()	set_name.update(**others*)	Update the set *set_name* by adding items from all *others* sets.

Frozenset:

- A frozenset is basically the same as a set, except that it is immutable.
- Once a frozenset is created, then its items cannot be changed. Since they are immutable, they can be used as members in other sets and as dictionary keys.
- The frozensets have the same functions as normal sets except none of the functions that change the contents (update, remove, pop, etc.) are available.
- Example:
  fs = frozenset(("g", "o", "o", "d"))
  print(fs) # output: frozenset({'o', 'd', 'g})