Tuples:

- Tuples are immutable, ordered collections of items, denoted by ()
- Each item in a tuple is called an element and is separated by a comma.

Functions in Tuple:

1. **index(element):** returns the index of the first occurrence of the given element in the tuple.
2. **count(element):** returns the number of occurrences of the given element in the tuple.
3. **len(tuple):** returns the number of elements in the tuple.
4. **min(tuple):** returns the smallest element in the tuple.
5. **max(tuple):** returns the largest element in the tuple.

```
# Creating a tuple
t = (1, 2, 3, 3, 4, 5)

# Index function
print("The index of 3 in the tuple is: ", t.index(3))
# Output: The index of 3 in the tuple is: 2

# Count function
print("The count of 3 in the tuple is: ", t.count(3))
# Output: The count of 3 in the tuple is: 2

# Length function
print("The length of the tuple is: ", len(t))
# Output: The length of the tuple is: 6

# Min function
print("The minimum element in the tuple is: ", min(t))
# Output: The minimum element in the tuple is: 1

# Max function
print("The maximum element in the tuple is: ", max(t))
# Output: The maximum element in the tuple is: 5

# Concatenation of two tuples
t1 = (6, 7)
t2 = t + t1
print("The concatenated tuple is: ", t2)
# Output: The concatenated tuple is: (1, 2, 3, 3, 4, 5, 6, 7)

# Changing elements (Convert tuple to list, change elements and convert list to tuple)
l = list(t2)
l[0] = 8
t2 = tuple(l)
print("The modified tuple is: ", t2)
# Output: The modified tuple is: (8, 2, 3, 3, 4, 5, 6, 7)
```

Adding items to a tuple:

- Tuples are immutable, meaning once you create a tuple you cannot change its elements.
- However, you can concatenate two tuples to form a new tuple, e.g. tuple1 + tuple2.

Changing elements:

- Tuples are immutable, meaning you cannot change its elements.
- If you need to change elements in a tuple, you'll have to convert the tuple to a list, change the elements and then convert the list back to a tuple.

The **del** statement is used in Python to delete a reference to an object. When you delete a reference, the object may not be immediately destroyed. Instead, the memory occupied by the object is released when there are no more references to the object.

```
t = (1, 2, 3, 4)
print("The original tuple: ", t)
# Output: The original tuple: (1, 2, 3, 4)
```

```
# Deleting the tuple
del t

# Accessing the tuple after deletion
# This will raise an error because the tuple has been deleted
print("The deleted tuple: ", t)
# Output: NameError: name 't' is not defined
```

Sets:

A set in Python is an unordered collection of unique elements.

Functions:

1. `add()`: Adds an element to the set.
2. `pop()`: Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.
3. `clear()`: Removes all elements from the set.

```
# Creating a set
s = {1, 2, 3}
print("The set: ", s)
# Output: The set: {1, 2, 3}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

s.add(4)
print("The set after adding an element: ", s)
# Output: The set after adding an element: {1, 2, 3, 4}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

x = s.pop()
print("The popped element: ", x)
# Output: The popped element: 1
print("The set after popping an element: ", s)
# Output: The set after popping an element: {2, 3}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

s.clear()
print("The set after clearing all elements: ", s)
# Output: The set after clearing all elements: set()
```

Adding elements to a set:

Sets can only contain immutable elements. Therefore, lists can't be added to a set directly, but tuple elements can be added to the set.

```
s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

# Adding a tuple to the set
s.add((4, 5))
print("The set after adding a tuple: ", s)
# Output: The set after adding a tuple: {1, 2, 3, (4, 5)}
```

Python Dictionaries

A dictionary in Python is an unordered collection of key-value pairs. It is also known as an associative array or hash map. You can use a dictionary to store values in a key-value format, where each key maps to a unique value.

```
# Creating a dictionary
d = {'key1': 'value1', 'key2': 'value2'}
```

```
print("The dictionary: ", d)
# Output: The dictionary: {'key1': 'value1', 'key2': 'value2'}
```

Dictionary Methods

Here are some of the common methods that can be performed on a dictionary:

1. `keys()`: Returns a view object that displays a list of all the keys in the dictionary.
2. `values()`: Returns a view object that displays a list of all the values in the dictionary.
3. `items()`: Returns a view object that displays a list of all the key-value pairs in the dictionary.
4. `update()`: Adds elements to the dictionary. If the key is already present, it updates the key-value.
5. The `update()` method is used to add elements to an existing dictionary or to merge two dictionaries.
6. The `pop()` method is used to remove an item from the dictionary.
7. The `get()` method is used to retrieve the value of a key from the dictionary.
8. The `fromkeys()` method is used to create a new dictionary where each key is the same and the values are set to None.
9. To create a deep copy of a dictionary, you can use the `copy` module in Python.

```
# Creating a dictionary
d = {'key1': 'value1', 'key2': 'value2'}
print("The dictionary: ", d)
# Output: The dictionary: {'key1': 'value1', 'key2': 'value2'}

d = {'key1': 'value1', 'key2': 'value2'}
print("The keys of the dictionary: ", d.keys())
# Output: The keys of the dictionary: dict_keys(['key1', 'key2'])

d = {'key1': 'value1', 'key2': 'value2'}
print("The values of the dictionary: ", d.values())
# Output: The values of the dictionary: dict_values(['value1', 'value2'])

d = {'key1': 'value1', 'key2': 'value2'}
print("The items of the dictionary: ", d.items())
# Output: The items of the dictionary: dict_items([('key1', 'value1'), ('key2', 'value2')])

d = {'key1': 'value1', 'key2': 'value2'}
print("The original dictionary: ", d)
# Output: The original dictionary: {'key1': 'value1', 'key2': 'value2'}

d.update({'key3': 'value3'})
print("The dictionary after updating an element: ", d)
# Output: The dictionary after updating an element: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

import copy

d = {'key1': 'value1', 'key2': 'value2'}
print("The original dictionary: ", d)
# Output: The original dictionary: {'key1': 'value1', 'key2': 'value2'}
d_copy = copy.deepcopy(d)
print("The deep copy of the dictionary: ", d_copy)
# Output: The deep copy of the dictionary: {'key1': 'value1', 'key2': 'value2'}

dict = {'key1': 'value1', 'key2': 'value2'}
dict2 = {'key3': 'value3', 'key4': 'value4'}
dict.update(dict2)
print("The updated dictionary: ", dict)
# Output: The updated dictionary: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3', 'key4': 'value4'}

dict = {'key1': 'value1', 'key2': 'value2'}
dict.pop('key1')
print("The dictionary after popping key1: ", dict)
# Output: The dictionary after popping key1: {'key2': 'value2'}

dict = {'key1': 'value1', 'key2': 'value2'}
print("The value of key1: ", dict.get('key1'))
# Output: The value of key1: value1

keys = ['key1', 'key2', 'key3']
dict = dict.fromkeys(keys, 'value')
print("The dictionary created from keys: ", dict) # Output: The dictionary created from keys: {'key1': 'value', 'key2': 'value', 'key3
```