Lists, Tuples, Sets & Dictionary

Lists:

In Python, a list is a versatile and widely used data structure that represents an ordered, mutable sequence of elements. Lists are one of the built-in data types, and they can contain elements of different data types such as integer, string, character ,including other lists.

1. Creating Lists:

Lists are created by enclosing elements in square brackets [] and list() method. Elements within a list can be of different data types, and a single list can contain a mix of numbers, strings, or even other lists.

Ex:

```
my_list = [1, 2, 'hello', 3.14, [4, 5]]  (OR)   my_list = list(1,2,3,4,5)
```

2. Indexing:

Elements in a list are accessed using zero-based indexing.

Negative indexing can be used to access elements from the end of the list.

Ex:

```
my_list_1 = [15,67,89,16,50]

my_list_2 = [1,2,3,[1,2,3],5]

print(my_list_1[4])    # Output: 50

print(my_list_2[-2])   # Output: [1,2,3]
```

3. Slicing:

You can slice a part of the original list and create a sublist using slicing. The syntax is my_list[start:stop:step].

Ex:

```
my_list = [10,30,50,70,90]

sublist = my_list[1:4]   # Output: [10, 30, 50]
```

4. Lists Methods:

Lists come with a variety of built-in methods, such as append(), extend(), insert(), remove(), pop(), index(), count(), sort(), and reverse().

Ex:

```
my_list = [3, 1, 4, 1, 5, 9, 2]

my_list.sort()            # Sort the list in place

my_list.append(6)         # Append an element to the end

my_list.remove(4)         # Remove the first occurrence of 4
```

5. Built-in Functions:

Python provides built-in functions like len(), sum(), min(), copy() and max() that can be used with lists.

Ex:

```
my_list = [10, 20, 30, 40, 50]

length = len(my_list)   # Output: 5

total = sum(my_list)    # Output: 150

my_list_2 = my_list.copy()  #Copy the content of my_list to my_list_2
```

Tuples:

In Python, a tuple is a collection data type that is similar to a list but has some key differences. Tuples are immutable, meaning once they are created, their elements cannot be changed or modified which makes them unique compared to lists.

1. Creating Tuples:

Tuples are created by enclosing elements in parentheses ().

Elements within a tuple can be of different data types, and a single tuple can contain a mix of numbers, strings, or other tuples just like lists.

Ex:

```
my_tuple = (1, 2, 'hello', 3.14, (4, 5))
```

2. Indexing:

Elements in a tuple are accessed using zero-based indexing, similar to lists.

Negative indexing can be used to access elements from the end of the tuple.

Ex:

```
my_tuples_1 = [15,67,89,16,50]

my_tuples_2 = [1,2,3,[1,2,3],5]

print(my_tuples_1[4])    # Output: 50

print(my_tuples_2[-2])   # Output: [1,2,3]
```

3. Slicing:

You can slice a part of the original tuple and create a sub tuple using slicing similar to lists. The syntax is my_tuple[start:stop:step].

Ex:

```
my_tuple = [10,30,50,70,90]
subtuple = my_list[1:4]   # Output: [10, 30, 50]
```

4. Arithmetic Operators in Tuples:

Tuples support various operations like concatenation (+), repetition (*), and the “in” operator.

Ex:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
concatenated = tuple1 + tuple2   # Output: (1, 2, 3, 4, 5, 6)
repeated = tuple1 * 3          # Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

5. Immutable Nature:

One of the main differences between tuples and lists is that tuples are immutable.

Once a tuple is created, you cannot modify, add, or remove elements.

Ex:

```
my_tuple = (1, 2, 3)
# The following will raise an error
my_tuple[1] = 5   # TypeError: 'tuple' object does not support item assignment
```

6. Tuple Unpacking:

You can assign the elements of a tuple to multiple variables in a single line. Tuple Unpacking is one of the useful method in Python.

Ex:

```
coordinates = (4, 5)

x, y = coordinates  # x = 4 and y = 5
```

7. Tuples build-in Methods:

Tuples have fewer built-in methods compared to lists due to their immutability.

Common methods include count() and index().

Ex:

```
my_tuple = (1, 2, 2, 3, 3, 3)

count_of_2 = my_tuple.count(2)   # Output: 2

index_of_3 = my_tuple.index(3)   # Output: 3
```

Sets:

In Python, a set is an unordered and mutable collection of unique elements which means it only stores unique values and doesn't allow duplicates.

1. Creating Sets:

Sets are created by placing elements inside curly braces {}.

Duplicate elements are not allowed in a set.

Alternatively, the set() constructor can be used to create an empty set or convert other iterable types (lists, tuples) into a set.

Ex:

```
my_set = {2,4,6,8,10}

empty_set = set()
```

2. Unique Elements:

Sets only store unique elements; duplicate values are automatically removed.

Ex:

```
unique_set = {1, 2, 2, 3, 3, 3}
print(unique_set)   # Output: {1, 2, 3}
```

Note : Since Set is unordered, they do not support indexing and slicing.

3. Set Operations:

Sets support various mathematical operations such as union (|), intersection (&), difference (-), and symmetric difference (^).

Ex:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1 | set2         # Output: {1, 2, 3, 4, 5, 6}
intersection_set = set1 & set2  # Output: {3, 4}
difference_set = set1 - set2    # Output: {1, 2}
```

4. Set Build-in Methods:

Sets have various built-in methods for common set operations.

Some common methods include add(), remove(), discard(), pop(), clear(), union(), intersection(), difference(), and symmetric_difference().

Ex:

```
my_set = {1, 2, 3}
my_set.add(4)             # Adds 4 to the set
```

my_set.remove(2) # Removes 2 from the set (raises KeyError if not present)

my_set.discard(5) # Removes 5 from the set (no error if not present)

popped_element = my_set.pop() # Removes and returns an arbitrary element from the set

5. Frozen Sets:

Python also supports a data type called frozenset, which is an immutable version of a set.

Ex:

frozen_set = frozenset([3,4,5])

Dictionary:

A dictionary in Python is an unordered and mutable collection of key-value pairs. It is a versatile data type that allows you to store and retrieve values using keys.

1. Creating a Dictionary:

Dictionaries are created using curly braces {} and consist of key-value pairs separated by commas.

Keys and values are separated by a colon ':' .

Ex:

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

2. Accessing Values:

Values in a dictionary are accessed using their corresponding keys.

Square brackets [] are used to retrieve the value associated with a key.

Ex:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(my_dict['name'])   # Output: John
```

3. Dictionary Operations:

Dictionaries support various operations like adding, updating, and deleting key-value pairs.

Ex:

```
# Adding a new key-value pair
my_dict['gender'] = 'Male'

# Updating the value of an existing key
my_dict['age'] = 26

# Deleting a key-value pair
del my_dict['city']
```

4. Dictionary Build-In Methods:

Dictionaries have built-in methods for common operations such as keys(), values(), items(), get(), pop(), update(), and clear().

Ex:

```
keys = my_dict.keys()       # Returns a view of all keys
values = my_dict.values()   # Returns a view of all values
items = my_dict.items()     # Returns a view of key-value pairs
```

```
age = my_dict.get('age', 0)  # Returns the value for the given key, or a default value if the key is not present

removed_value = my_dict.pop('gender')  # Removes and returns the value for the given key


my_dict.update({'city': 'San Francisco', 'occupation': 'Engineer'})  # Updates the dictionary with key-value pairs from another dictionary

my_dict.clear()           # Removes all items from the dictionary
```

5. Iterating Through Dictionaries:

You can use “for” loops to iterate through keys, values, or items in a dictionary.

Ex:

```
for key in my_dict:
    print(key, my_dict[key])           #Return Keys of Dictionary


for value in my_dict.values():
    print(value)                       #Return Value of Keys in a Dictionary


for key, value in my_dict.items():
    print(key, value)                  #Return Keys and Value of a Dictionary
```