Python lists, sets, and tuples

A **collection** in Python is a single variable used to stare multiple values

List = [ ] ordered and changeable. Duplicates OK

Set = { } unordered and immutable, but Add/Remove OK. NO duplicates

Tuple = ( ) ordered and unchangeable. Duplicates OK. FASTER

In other programming languages, a **list** is the same as an "**array**". Lists in Python are versatile and can be used to store and manipulate collections of items.

Accessing Elements - You can access elements in a list using indexing. Indexing starts at 0 for the first element:

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

first_element = my_list[0]  # Access the first element
second_element = my_list[1]  # Access the second element

print(my_list)
print(first_element)
print(second_element)
```

Modifying Lists - Lists are mutable, meaning you can change their elements:

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

my_list[0] = 10  # Change the value of the first element
my_list.append(6)  # Add an element to the end of the list

print(my_list)
```

Slicing Lists - You can create sublists (slices) by specifying a range of indices:

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

subset = my_list[1:4]  # Extract elements from index 1 to 3
(not including 4)

print(subset)
```

Iterating Over Lists - You can use a **for** loop to iterate over the elements of a list:

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

for item in my_list:
    print(item)
```

List Comprehensions - List comprehensions provide a concise way to create lists:

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

squared_numbers = [x**2 for x in my_list]

print(squared_numbers)
```

Built-in Functions - Python provides built-in functions for working with lists, such as **len()**, **sum()**, **min()**, and **max()**:

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

length = len(my_list)
total = sum(my_list)
minimum_value = min(my_list)
maximum_value = max(my_list)

print(length, total, minimum_value, maximum_value)
```

Removing Elements - You can remove elements from a list using methods like **remove()**, **pop()**, or **del**:

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

my_list.remove(3)  # Remove the first occurrence of the
specified value
popped_value = my_list.pop(2)  # Remove and return the element
at index 2
del my_list[1]  # Remove the element at index 1

print(my_list)
```

Checking Membership - Check if an element is in a list using the **in** keyword:

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

is_present = 5 in my_list  # Check if 5 is in the list

print(is_present)
```

Sorting - You can sort a list using the **sort()** method or the **sorted()** function:

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

my_list.sort()  # Sort the list in-place
sorted_list = sorted(my_list)  # Create a new sorted list

print(sorted_list)
```

Lists work with any data type – example of a list of string values:

```
fruits = ["kiwi", "banana", "cherry", "orange", "apple"]

#print(dir(fruits)) # <-- scroll to end to see methods
#print(help(fruits)) # <-- displays a list of options
#print(len(fruits)) # <-- shows number of list items
#print("banana" in fruits) # <-- returns True or False

#fruits[0] = "pineapple" # <-- replaces position 0
#fruits.append("pineapple") # <-- adds to the end
#fruits.remove("apple") # <-- removes item 'apple'
#fruits.insert(0, "pineapple") #<-- at start
#fruits.sort() # <-- sorts list alphabetically
#fruits.reverse() # <-- reverses the order of the list
#fruits.clear() # <-- removes everything from list

#print(fruits)

#print(fruits.index("apple")) # <-- returns location #
#print(fruits.count("banana"))
```

Sets

a set is an unordered collection of unique elements. Sets are defined by placing comma-separated values inside curly braces **{ }**. Main characteristics of sets are:

1. **Uniqueness:** A set cannot contain duplicate elements. If you try to add an element that is already present, it won't be added again.
2. **Unordered:** The elements in a set have no specific order. Unlike lists or tuples, you cannot access elements in a set using indices.
3. **Mutable:** Sets are mutable, meaning you can add or remove elements after the set is created.

Sets in Python support various operations such as union, intersection, difference, and more. Here's a brief overview of sets and their usage:

Iteration over a set, checking membership and set comprehensions, dir and help are the same as for Lists.

Adding and Removing Elements

```
my_set = {1, 2, 3, 4, 5}

my_set.add(6)        # Add an element to the set
my_set.remove(3)     # Remove an element from the set
                     # (raises KeyError if not present)
my_set.discard(3)    # Remove an element from the set
                     # (no error if not present)

print(my_set)
```

Set Operations

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

print(set1.union(set2))          # Union of two sets
print(set1.intersection(set2))   # Intersection of two sets
print(set1.difference(set2))     # Set difference
```

Because sets are unordered, we cannot use the **index operator** to look in a specific location.

```
my_set = {1, 2, 3, 4, 5}

print(my_set[0]) # <-- this will fail
```

Tuples

A tuple is an ordered and immutable collection of elements. Tuples are similar to lists, they are FASTER and they have a few key differences:

1. **Immutability:** Once a tuple is created, its elements cannot be changed, added, or removed. This makes tuples suitable for situations where you want to ensure that the data remains constant.
2. **Ordered:** Like lists, tuples maintain the order of elements. Each element in a tuple is assigned a specific index, and you can access elements using these indices.
3. **Syntax:** Tuples are defined by enclosing elements in parentheses **()**.

Here's an overview of tuples and their usage:

Accessing Elements

```
my_tuple = (1, 2, 3, "four", 5.0)

print(my_tuple[0])  # Access the first element
print(my_tuple[3])  # Access the second element
```

Immutability

```
my_tuple = (1, 2, 3, "four", 5.0)

# Attempting to modify a tuple will result in an error
my_tuple[0] = 10  # Raises TypeError
```

Tuple Unpacking – create 5 variables (a-e) from a single tuple of 5 items

```
my_tuple = (1, 2, 3, "four", 5.0)

# Unpack the elements into separate variables
a, b, c, d, e = my_tuple

print(d)
```

Iterating over a Tuple

```
my_tuple = (1, 2, 3, "four", 5.0)

for element in my_tuple:
    print(element)
```

Concatenation and Repetition

```
tuple1 = (1, 2, 3)
tuple2 = ("a", "b", "c")
combined_tuple = tuple1 + tuple2  # Concatenation
repeated_tuple = tuple1 * 2       # Repetition

print(combined_tuple)
print(repeated_tuple)
```

Checking Membership:

```
my_tuple = (1, 2, 3, "four", 5.0)

is_present = 5 in my_tuple  # Check if 5 is in the tuple

print(is_present)
```