ASSIGNMENT

DATA STRUCTURES

1.String Slicing : String slicing in python is used to access a part of the string using specified ranges.

Syntax : string[start : end : step]

Here,

Start – starting index of the substring you want to access

End – ending index of the substring you want to access

Step – it tells how many characters to skip

Ex:

S = “Hello World!”

Print(S[0:5]) #Hello

Print(S[2:6:2]) #lo

#we can do this with negative indexing too

Print(S[-5:-1]) #orld

Print(S[-7:-1:3]) # r

This is how string slicing works in python.

---

2. **List :** Lists in python are an ordered collection of items/elements and is one of the most used data structure.

Key features of list are,

List items are ordered : This means the order in which the list items are specified will remain same and first element is at index 0 , second element is at index 1 and so on.

Mutable: Lists are mutable. It means that the list elements can be changed after once they are created. Lists support item insertion, deletion , modification.

Heterogenous : Lists can store various kinds of data like integers, float , string,

Complex numbers , Boolean and so on even other lists too.

Dynamic sizing : list can grow or shrink when we add or delete elements.

Slicing and Indexing : list support slicing using which we can access only a part of the list.

Iterable : lists are iterable, that means we can traverse through the list items gone by one.

List Comprehension : lists support list comprehension which is a very concise to perform computation tasks on list.

---

3. **Accessing list Items:** List items can be accessed using list indexing and list slicing.

Indexing: accessing list items based on indices of the list.

- Lists has both positive and negative indexing
- Positive indexing starts from 0 to len(list)-1
- Negative starts from -1 in the reverse oreder

Ex: l = [1,2,3,4,5,6]

```
Print(l[0]) #returns first element 1

Print(l[-1]) #returns last element 6

Print(l[3]) #returns 4th element
```

Slicing: slicing can be done using colon ‘:’ operator.

Ex: l = [1,23,45,67,78]

Print(l[1:3])

Print(l[-3:-1])

Print(l[0:4:2])

Modifying list elements:

You can change the value of an element at a specific index,

```
my_list = [10, 20, 30, 40]
my_list[2] = 100
print(my_list)
```

You can replace multiple elements using slicing,

```
my_list = [10, 20, 30, 40, 50]
my_list[1:4] = [200, 300, 400]
print(my_list)  # Output: [10, 200, 300, 400, 50]
```

list comprehension:

list comprehension can be used to create new list by performing modifications using list comprehension.

```
my_list = [1, 2, 3, 4]
new_list = [x * 2 for x in my_list]  # Multiply each element by 2
print(new_list)  # Output: [2, 4, 6, 8]
```

Removing list elements:

Remove():

```
my_list = [1, 2, 3, 2]
my_list.remove(2)  # Removes the first occurrence of 2
print(my_list)     # Output: [1, 3, 2]
```

pop():

```
my_list = [1, 2, 3]
popped_value = my_list.pop(1)  # Removes the element at index 1
print(my_list)              # Output: [1, 3]
print(popped_value)         # Output: 2
```

del :

```
my_list = [10, 20, 30, 40]
del my_list[2]  # Delete the element at index 2
print(my_list)  # Output: [10, 20, 40]
```

clear:

```
my_list = [1, 2, 3, 4]
my_list.clear()  # Removes all elements
print(my_list)   # Output: []
```

---

4. Lists and tuples are both data structures in Python used to store collections of elements, but they have some important differences. Here's a comparison of their features:

1. Mutability

- **List**: Mutable (we can modify, add, or remove elements).

```
my_list = [1, 2, 3]
my_list[1] = 20  # Modifying element at index 1
print(my_list)   # Output: [1, 20, 3]
```

- **Tuple**: Immutable (once created, elements cannot be changed, added, or removed).

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

```
my_tuple[1] = 20
# This will raise an error: TypeError
```

2. Syntax

- **List**: Defined using square brackets [ ].

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

- **Tuple**: Defined using parentheses ( ), or just commas for a single tuple.

  ```
  my_tuple = (1, 2, 3)
  single_element_tuple = (1,)
  # Need a comma for single-element tuple
  ```

3. Performance

- **List**: Slower than tuples in operations like iteration and accessing elements because of their mutable nature.
- **Tuple**: Faster because they are immutable and have less overhead.
  - If performance is a concern, and you don't need to modify the collection, tuples are more efficient.

4. Use Cases

- **List**: Used when you need a dynamic collection, where elements can be added, removed, or changed.
  - Ex: Storing a list of users where modifications are expected.
- **Tuple**: Used for static or read-only collections of items, especially when you want to ensure the integrity of data.
  - Ex: Storing geographic coordinates (latitude, longitude) or data that shouldn’t be changed.

5. Methods

- **List**: Has a wide range of built-in methods for manipulation like append(), insert(), pop(), remove(), etc.

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

```
my_list.append(4)  # Adds 4 at the end
print(my_list)     #  [1, 2, 3, 4]
```

- **Tuple**: Limited methods because it's immutable. It has only two methods: count() and index().

```
my_tuple = (1, 2, 3, 2)
print(my_tuple.count(2))  # Output: 2
print(my_tuple.index(3))  # Output: 2
```

6. Memory Usage

- **List**: Takes more memory because of its dynamic nature.
- **Tuple**: Takes less memory because it's immutable, and Python optimizes storage for tuples.

7. Nested Structures

- **List**: Can hold other lists (or any other object), making it possible to create complex data structures.

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

- **Tuple**: Can also hold other tuples or lists, but the tuple itself remains immutable.

```
my_tuple = (1, [2, 3], 4)
# The list inside the tuple can still be modified
my_tuple[1][0] = 20  # This is allowed
```

8. Usage as Dictionary Keys

- **List**: Cannot be used as dictionary keys because they are mutable and hence not hashable.

```
#my_dict = {[1, 2]: "value"}  # Raises TypeError: unhashable type: 'list'
```

- **Tuple**: Can be used as dictionary keys because they are immutable and hashable.

```
my_dict = {(1, 2): "value"}
print(my_dict[(1, 2)])  # Output: "value"
```

9. Copying Behavior

- **List**: Copying a list creates a reference to the original list (shallow copy). Changes to the copy affect the original unless you perform a deep copy.

```
list1 = [1, 2, 3]
list2 = list1  # Shallow copy
list2[0] = 10
print(list1)   # Output: [10, 2, 3]
```

- **Tuple**: Since tuples are immutable, copying a tuple is more straightforward. The concept of shallow or deep copy doesn't apply.

```
tuple1 = (1, 2, 3)
tuple2 = tuple1
```

---

5. Key Features of sets are,

- Unordered Collection : There is no specific order for elements in a set. It means that the elements are stored in a random manner and it is not possible to access them using indexing.

```
Ex :    my_set = {23,12,45,67,32,74}
        print(my_set)
        #output : {32,67,23,74,12,45}
```

- Unique Elements : Sets doesn't allow duplicate elements and always contain unique elements. Though we try to put a duplicate value set doesn't accept the value, it discards those duplicated values.

```
Ex: my_set = {12,23,12,45,67,32,74}
    print(my_set)
 #output : {32,67,23,74,12,45}
```

- Mutable : sets are mutable, it means we can add or remove elements according to our comfort after the set has been created.

  Ex : my_set = {12,43,56,85}

  my_set.add(67)

  print(my_set) #{67,43,12,85,56}

  my_set.remove(56) #{67,43,12,85}

  print(my_set)

- Immutable Set variant : Python has a variant called frozenset, which is immutable.

  Ex: my_set = frozenset({12,43,65,31,86})

  my_set.add(10)

  print(my_set) #generates Attribute error

- Uses hashing to store the elements which makes it faster to access the elements.

  Ex : hash() function is used to calculate hash value

- Efficient membership test : Testing for an element if it is present in the set or not is faster than testing for element in lists and tuples as sets use hash tables.

  Ex : my_set = {34 , 43, 32,67,53,78}

  Print(34 in my_set) #True

---

6. **use case of tuples in python :**

- Tuples are immutable , so tuples would be very helpful when we store data that can not be changed.

  For example, data like mail id's, mobile no's , aadhar no's and so on

When we group related data and ensure that the data cannot be modified then also tuples would be best suited.

- Returning multiple values from a function.
- Tuples can be used as dictionary keys
- Used to store heterogeneous data.

Use case of sets in python:

- To remove duplicate values and return unique elements.
- When the elements order is not mandatory.
- Membership testing.
- To perform set operations like Union , Intersection , Difference.

---

7. As we know dictionary stores values in the form of key-value pairs,

Accessing Dictionary Elements, is done using key values.

ex : dic = {1 : 'one' , 2 : 'two' , 3:'three'}

print(dic[1])

Also we can use .keys() -> to get the list of keys

.values() -> to get the list of values

.items() -> to get dict items as key-value pairs

.get() -> to get the corresponding value

Modifying Dictionary Elements,

Modifying elements in a Python dictionary involves updating the value of an existing key or adding new key-value pairs.

--Modifying existing key value through direct assignment

--Adding a new key-value pair

--Update() -> to change multiple values at a time

--setdefault – to add or return default value

Ex : dic = { 1 : ‘one’ , 2:’two’,3:’three’}

Print(dic)

dic[2] = ‘Twice’

dic[4] = ‘four’

dic.update({1:’once’,3:’thrice’,4:’fourth’})

dic.setdefault(5,”five”)

Print(dic)

Deleting the items in a dictionary,

Dictionary items can be deleted using various methods depending on various methods, depending on whether you want to delete a specific key-value pair.

--using del statement

--using pop() method ->removes the item with specified key value

--using popitem() method ->removes last element

--clear() ->removes all items

Ex : my_dict = { 1 : ‘one’ , 2:’two’,3:’three,4:’four’,5:’five}

Del my_dict[1]

My_dict.pop(2)

```
My_dict.popitem()  #deletes last item

My_dict.clear()  #dictionary becomes empty {}
```

---

8. Dictionary keys must be **immutable** because they are hashed to allow fast lookups, comparisons, and membership tests. If the keys were mutable, their hash values could change over time, making the dictionary unreliable and inefficient. Immutable keys ensure that the hash value remains constant throughout the key's existence in the dictionary, allowing the dictionary to correctly locate, store, and retrieve values.

Dictionary Keys must be immutable because,

1.Hashing and Efficiency

2.consistency in key comparisions

3.Reliability

Immutables in python are,

Integers , strings , tuples …

Mutables are,

Lists , sets

Ex : Using immutable keys,

```
My_dict = {“name”:”alice”,1:”first”,(1,2) : “tuple_key”}

Print(my_dict[“name”])

Print(my_dict[1])

Print(my_dict[(1,2)])
```

Note : Trying to use lists,sets as keys will raise a ‘TypeError’ because mutables cannot be hashed.
