

✓ Name: Shivanshu Singh Parihar

Batch : September 2024

Assignment Date: 08 Oct 2024

Assignment : Python - Data Structure

Assignment -2

1. Discuss string slicing and provide examples?

Ans: String slicing in Python allows you to extract a portion of a string by specifying a start and end index. It uses the syntax string [start:end], where:

- start: The index where the slice begins (inclusive).
- end: The index where the slice ends (exclusive).
- step: The interval between each index in the slice (optional).

```
phrase = "Python Programming"
substring = phrase[7:18]
print(substring)
```

➞ Programming

```
my_string = "my name is shivanshu"
sliced1 = my_string[3:7]
sliced2 = my_string[11:]
sliced3 = my_string[-5:-1]
sliced4 = my_string[::3]
reversed_string = my_string[::-1]
```

```
print(sliced1,
      sliced2,
      sliced3,
      sliced4,
      reversed_string)
```

➞ name shivanshu ansh mneshah uhsnavihs si eman ym

2. Explain the key features of lists in Python.

Ans: Lists in Python are versatile and widely used data structures. Lists in Python are ordered, mutable and can contain items of different types at the same time. Each item in a Python list also has an associated index, where the first item's index is 0 and ascends accordingly for each following item. Here are some key features of Lists are :

- **Ordered:** Lists maintain the order of elements, meaning the position of each element is preserved. You can access elements by their index.

- **Mutable:** Lists are mutable, allowing you to modify them after creation. You can change, add, or remove elements.
- **Heterogeneous:** Lists can contain elements of different data types, including integers, floats, strings, and even other lists.
- **Dynamic Size:** Lists can grow and shrink in size as you add or remove elements. There's no need to declare a fixed size when creating a list.
- **Versatile Methods:** Python provides a variety of built-in methods for lists, such as `.append()`, `.remove()`, `.pop()`, `.sort()`, and `.reverse()`, making it easy to manipulate the list.
- **Nested Lists:** Lists can contain other lists as elements, allowing the creation of complex data structures like matrices.
- **Support for Slicing:** You can use slicing to access a range of elements in a list, just like with strings.

```
shivanshu_list = [1, "hello", 3.14, [2, 3]]
shivanshu_list.append(5)
print(shivanshu_list)
print(shivanshu_list[1])
print(shivanshu_list[3][1])
```

```
⇒ [1, 'hello', 3.14, [2, 3], 5]
   hello
   3
```

3. Describe how to access, modify, and delete elements in a list with examples.

Ans: Accessing, changing, and removing items in a list in Python is easy.

- **Accessing Elements :** We can get an item from a list by its position (index). In Python we use zero-based indexing, which means the first item is at index 0.

```
Accessing_Elements = [10, 20, 30, 40, 50]
first_value = Accessing_Elements[0]
next_value = Accessing_Elements[2]

print(first_value, next_value)
```

```
⇒ 10 30
```

- **Modifying Elements :** We can change an item in a list by assigning a new value to its index.

```
Modifying_Elements = [10, 20, 30, 40, 50]
Modifying_Elements[1] = 25    #we use zero indexing, that means [1] element is 20.
print(Modifying_Elements)
```

```
⇒ [10, 25, 30, 40, 50]
```

- **Deleting Elements :** We can remove or delete items from a list in a different way, that is by using `del`, `.remove()`, `.pop()`

- Using `del`: This will removes an item at a specific index.

```
Deleting_Elements = [10, 20, 30, 40, 50]
del Deleting_Elements[1]
print(Deleting_Elements)
```

⇒ [10, 30, 40, 50]

- Using `.remove()`: This will removes the first occurrence of a specific value.

```
Deleting_Elements = [10, 20, 30, 40, 50]
Deleting_Elements.remove(40)
print(Deleting_Elements)
```

⇒ [10, 20, 30, 50]

- Using `.pop()`: This will removes the last item and returns it.

```
Deleting_Elements = [10, 20, 30, 40, 50]
last_item = Deleting_Elements.pop()
print(last_item)
print(Deleting_Elements)
```

⇒ 50
[10, 20, 30, 40]

4. Compare and contrast tuples and lists with examples.

Ans: Tuples and lists are both used to store collections of items in Python, but they have some important differences.

- **Mutability**
- Lists are Mutable, means we can change it after it created.
- Tuples are Immutable, means once we create a tuple, we cannot make changes.

```
my_list = [1, 2, 3]
my_list[1] = 5
my_list.append(4)
print(my_list)
```

⇒ [1, 5, 3, 4]

```
my_tuple = (1, 2, 3)
my_tuple[1] = 5                                # This is not allowed in tuple
print(my_tuple)
```




```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-15-128d8802e909> in <cell line: 2>()  
      1 my_tuple = (1, 2, 3)  
----> 2 my_tuple[1] = 5                    # This is not allowed in tuple  
      3 print(my_tuple)  
  
TypeError: 'tuple' object does not support item assignment
```



- **Syntax**
- Lists are defined using square brackets [].
- Tuples are defined using parentheses ().


```
my_list = [1, 2, 3]  
my_tuple = (1, 2, 3)  
print("List is", my_list, "and", "Tuple is", my_tuple)
```

 List is [1, 2, 3] and Tuple is (1, 2, 3)

- **Performance**
- In Lists generally it have a bit more overhead because they are mutable, making them slightly it is slower for certain operations.
- Tuples are typically faster than lists because of their immutability, which can lead to performance improvements in some of the situations.
- **Use Cases**
- Lists are used when we need a collection of items that may change after created. For Example I'm going to market and I have list of items. But after mother told me to add some more items.
- Tuples are useful for fixed collections of items and can be used as keys in dictionaries due to their immutability.

we can simply understand by example of Mutability.

```
my_list = ["Mango", "Bis-kit", "milk"]  
my_list[1] = "Chana-moong"  
my_list.append("Almond")  
print(my_list)
```

 ['Mango', 'Chana-moong', 'milk', 'Almond']

```
my_tuple = ("Mango", "Bis-kit", "milk")  
my_tuple[1] = "Chana-moong"                # This is not allowed in tuple  
print(my_tuple)
```



```

-----
TypeError                                Traceback (most recent call last)
<ipython-input-30-fb3ffd12d6c6> in <cell line: 2>()
      1 my_tuple = ("Mango", "Bis-kit", "milk")
----> 2 my_tuple[1] = "Chana-moong"           # This is not allowed in tuple
      3 print(my_tuple)

TypeError: 'tuple' object does not support item assignment

```



- **Built-in methods**
- Lists have many built-in methods for adding, removing, and modifying elements, such as `.append()`, `.remove()`, and `.pop()` etc.
- Tuples have fewer methods because they can't be changed due to immutability. We mainly use `.count()` and `.index()`.

```

my_list = [10, 20, 30]
my_list[1] = 25 # Modifying the list
my_list.append(40) # Adding to the list
print(my_list) # Output: [10, 25, 30, 40]

```



```
[10, 25, 30, 40]
```

```
my_tuple = ("Shivanshu", "Alice", "Bob", "Shivanshu", "Eve")
```

```
count_shivanshu = my_tuple.count("Shivanshu")
print(f"'Shivanshu' occurs {count_shivanshu} times in the tuple.")
```

```
index_shivanshu = my_tuple.index("Shivanshu")
print(f"The first occurrence of 'Shivanshu' is at index {index_shivanshu}.")
```



```
'Shivanshu' occurs 2 times in the tuple.
The first occurrence of 'Shivanshu' is at index 0.
```

5. Describe the key features of sets and provide examples of their use.

Ans: In Python, a set is a built-in data type that represents an unordered collection of unique elements. Sets are fundamental mathematical concepts that have several key features:

- **Unordered Collection:** Sets do not have a specific order. For example, the set {1,2,3} is the same as {3,2,1}.

```

set1 = {1, 2, 3}
set2 = {3, 2, 1}
print(set1 == set2) # Demonstrating that they are equal despite order

```



```
True
```

- **Uniqueness:** Elements in a set are unique. For instance, {1,2,2,3} is equivalent to {1,2,3}.

```

# Creating a set with duplicate elements
set_with_duplicates = {1, 2, 2, 3}

```

```
# Displaying the set, duplicates are removed
print(set_with_duplicates)
```

```
⇒ {1, 2, 3}
```

- **Membership:** We can determine if an element is a member of a set. For example, in the set $\{a,b,c\}$, the statement $a \in \{a,b,c\}$ is true.

```
my_set = {'a', 'b', 'c'}
```

```
# Checking membership
print('a' in my_set)
print('d' in my_set)
```

```
⇒ True
False
```

- **Subset:** A set can contain subsets, which are sets formed from elements of the original set. For instance, if $A=\{1,2,3\}$, then $\{1,2\}$ is a subset of A.

```
A = {1, 2, 3}
B = {1, 2}
```

```
# Checking if B is a subset of A
is_subset = B.issubset(A)
print(is_subset)
```

```
⇒ True
```

- **Union and Intersection:** Sets can be combined using operations like union and intersection. For example:
 - Union: $A \cup B$ combines all unique elements from sets A and B.
 - Intersection: $A \cap B$ includes only the elements common to both sets.

```
A = {1, 2, 3}
B = {2, 3, 4}
```

```
# Union
union_set = A.union(B)
print("Union:", union_set)
```

```
# Intersection
intersection_set = A.intersection(B)
print("Intersection:", intersection_set)
```

```
⇒ Union: {1, 2, 3, 4}
Intersection: {2, 3}
```

- **Empty Set:** The empty set, denoted \emptyset , contains no elements and is a subset of every set.

```
empty_set = set()
# Checking if it's a subset of any set
print(empty_set.issubset(A))
```

➡ True

6. Discuss the use cases of tuples and sets in Python programming.

Ans: Tuples and sets in Python are both versatile data structures, but they serve different purposes and have unique characteristics. Here are the some use cases for Tuples and sets are:

- **Tuples:**
- **Immutable Collections:** Tuples are immutable, meaning their contents cannot be changed after creation. This makes them ideal for storing data that should remain constant throughout the program. For example we have Addhar Card Number, Epfo Id, Unique code etc.

```
def get_employee_details():
    aadhar_number = "1234-5678-9101"
    employment_id = "EMP12345"
    unique_code = "XYZ2024"
    return (aadhar_number, employment_id, unique_code)
employee_details = get_employee_details()
print("Employee Details:")
print(f"Aadhar Number: {employee_details[0]}")
print(f"Employment ID: {employee_details[1]}")
print(f"Unique Code: {employee_details[2]}")
```

➡ Employee Details:
Aadhar Number: 1234-5678-9101
Employment ID: EMP12345
Unique Code: XYZ2024

- **Grouping Related Data:** Tuples are useful when we need to group a fixed collection of data together, such as coordinates, database records, or key-value pairs.

```
coordinates = (40.7128, -74.0060)
print(coordinates)
```

➡ (40.7128, -74.006)

- **Fixed Data Structure:** When we need a collection of data that won't change, such as a record in a database, tuples provide a fixed-size structure.

```
person_info = ("John Doe", 28, "Engineer")
```

```
# Accessing tuple elements
name = person_info[0]
age = person_info[1]
profession = person_info[2]
```

```
print("Name:", name)
print("Age:", age)
```

```
print("Profession:", profession)
```

```
➦ Name: John Doe  
Age: 28  
Profession: Engineer
```

- **Faster Access:** Tuples, being immutable, are faster to access than lists. If performance is a concern and the data does not require modification, using tuples is beneficial.

```
import time  
  
start = time.time()  
list_example = [1, 2, 3, 4, 5]  
end = time.time()  
print("List creation time:", end - start)
```

```
➦ List creation time: 9.822845458984375e-05
```

- **Return Multiple Values:** Tuples are often used to return multiple values from functions. This allows for clean, unpacked assignment in one line.

```
def get_dimensions():  
    return (1920, 1080)  
  
width, height = get_dimensions()  
print(f"Width: {width}, Height: {height}")
```

```
➦ Width: 1920, Height: 1080
```

- **Heterogeneous Data:** Tuples can hold different data types. It has a variety of data in it.

```
person = ("Shivanshu", 30, 5.7)  
print(person)
```

```
➦ ('Shivanshu', 30, 5.7)
```

- **Dictionary Keys:** Tuples can be used as keys in dictionaries because they are hashable, unlike lists. This is useful for creating composite keys.

```
location_data = {  
    (10, 20): "Point A",  
    (15, 25): "Point B"  
}  
print(location_data[(10, 20)])
```

```
➦ Point A
```

- **Sets**

- **Unique Elements:** Sets automatically handle duplicates, making them ideal for collections where uniqueness is a requirement. e.g. storing unique user IDs.

```
numbers = {1, 2, 2, 3, 4}
print(numbers)
```

```
➡ {1, 2, 3, 4}
```

- **Membership Testing:**

Sets provide average time complexity for membership tests. If we need to check whether an item exists in a collection frequently, using a set is much more efficient than a list.

```
fruits = {'apple', 'banana', 'orange'}
if 'banana' in fruits:
    print("Banana is in the set.")
else:
    print("Banana is not in the set.")
```

```
➡ Banana is in the set.
```

- **Removing Duplicates:** When converting a list with potential duplicates into a set, you can easily remove duplicates.

```
my_list = ["Shivanshu", "Ajay", "Guruji", 3, "Shivanshu", "Data", "Analytics"]
unique_items = set(my_list)
print(unique_items)
```

```
➡ {3, 'Shivanshu', 'Data', 'Ajay', 'Analytics', 'Guruji'}
```

- **Data Analysis:** In data analysis tasks, sets can be used to quickly find common or differing elements between datasets, making them helpful for operations like finding common customers between two businesses.

```
data1 = {'apple', 'banana', 'cherry'}
data2 = {'banana', 'dragonfruit', 'cherry'}
```

```
# Finding common elements
common = data1.intersection(data2)
print("Common items:", common)
```

```
➡ Common items: {'cherry', 'banana'}
```

- **Removing Unwanted Items:** We can use set operations to filter out unwanted items from a collection. For instance, using set difference to exclude certain elements from a list.

```
all_items = {'apple', 'banana', 'cherry', 'date'}
unwanted = {'banana', 'date'}
```

```
# Removing unwanted items
```

```
filtered_items = all_items.difference(unwanted)
print("Filtered items:", filtered_items)
```

```
➦ Filtered items: {'cherry', 'apple'}
```

7. Describe how to add, modify, and delete items in a dictionary with examples.

Ans: Dictionaries in Python are versatile data structures that store key-value pairs. Here's how to add, modify, and delete items in a dictionary.

- Adding Items: Assign a value to a new key.
- Modifying Items: Assign a new value to an existing key.
- Deleting Items: Use `del` or `pop()` to remove a key-value pair.

Adding Items : You can add new key-value pairs to a dictionary simply by assigning a value to a new key.

```
my_dict = {'name': 'Shivanshu', 'age': 24}
```

```
# Adding a new key-value pair
my_dict['city'] = 'Raipur'
print(my_dict)
```

```
➦ {'name': 'Shivanshu', 'age': 24, 'city': 'Raipur'}
```

Modifying Items: To modify an existing value, you can assign a new value to an existing key.

```
my_dict = {'name': 'Shivanshu', 'age': 24, 'City': 'Raipur'}
my_dict['age'] = 23
print(my_dict)
```

```
➦ {'name': 'Shivanshu', 'age': 23, 'City': 'Raipur'}
```

Deleting Items: You can delete items from a dictionary using the `del` statement or the `pop()` method.

```
my_dict = {'name': 'Shivanshu', 'age': 24, 'City': 'Raipur'}
# Deleting a key-value pair using del
del my_dict['City']
print(my_dict)
```

```
# Using pop to remove 'age' and get its value
age = my_dict.pop('age')
print(my_dict)
print("Removed age:", age)
```

```
# Trying to pop a non-existent key with a default value
unknown = my_dict.pop('city', 'Not Found')
print(unknown)
```

```
➦ {'name': 'Shivanshu', 'age': 24}
{'name': 'Shivanshu'}
Removed age: 24
Not Found
```

8. Discuss the importance of dictionary keys being immutable and provide examples.

Ans: The requirement for dictionary keys to be immutable is vital for ensuring hashability, data integrity, consistency, preventing key conflicts, and optimizing performance. By enforcing this rule, Python dictionaries can operate efficiently and reliably, making them a robust choice for data storage and retrieval. In Python, dictionary keys must be immutable types. This requirement is crucial for several reasons:

- Importance of Immutable Keys

Hashability:: Dictionary keys are hashed to enable fast access to values. Immutable types, such as strings, numbers, and tuples, have a fixed hash value, which means that their hash does not change during their lifetime. This stability is essential for maintaining the integrity of the dictionary.

```
my_dict = {  
    'name': 'Shivanshu'  
}
```

```
# The hash value for 'name' will remain the same  
print(hash('name'))
```

```
➡ 9071918059470547895
```

Data Integrity: If mutable objects were allowed as keys, changes to the object could lead to inconsistent states within the dictionary. This could cause difficulties in retrieving values associated with those keys.

```
my_dict = {  
    (1, 2): 'Parihar'  
}
```

```
# Accessing the value with the same key  
print(my_dict[(1, 2)])
```

```
➡ Parihar
```

Consistency: Immutable keys maintain the consistency of the dictionary structure. Once a key-value pair is added, the key remains unchanged, allowing reliable access and manipulation.

```
coordinates = {  
    (0, 0): 'Shivanshu',  
    (1, 1): 'Singh',  
    (2, 2): 'Parihar'  
}
```

```
# Retrieving values consistently  
print(coordinates[(0, 0)])  
print(coordinates[(1, 1)])  
print(coordinates[(2, 2)])
```

```
➡ Shivanshu  
Singh  
Parihar
```

Preventing Key Conflicts: Using immutable keys prevents conflicts that could arise from changing key values. If mutable keys were permitted, updating a key might lead to a scenario where two different values are mapped to the same key.

```
try:
    my_dict = {
        [1, 2]:
    }
except TypeError as e:
    print(e)
```

⇒ unhashable type: 'list'

Performance Optimization: The immutability of keys allows Python to optimize operations like lookups and insertions. The hash values remain constant, allowing for faster retrieval.

```
my_dict = {
    'name': 'Shivanshu',
    'age': 24
}

# Fast lookup using immutable string keys
print(my_dict['name'])
```

⇒ Shivanshu