```
#1. Discuss string slicing and provide examoles.

#Ans:-String slicing is a technique used to extract a subset of characters from a string.

#Examples

#1. Simple Slicing


my_string = "Hello, World!"
print(my_string[0:5])  # Output: "Hello"


#2. Omitting Start Index


my_string = "Hello, World!"
print(my_string[:5])  # Output: "Hello"


#3. Omitting Stop Index


my_string = "Hello, World!"
print(my_string[7:])  # Output: "World!"


#4. Negative Indices


my_string = "Hello, World!"
print(my_string[-6:])  # Output: "World!"


#5. Step Parameter


my_string = "Hello, World!"
print(my_string[::2])  # Output: "Hlo ol!"


#6. Reversing a String


my_string = "Hello, World!"
print(my_string[::-1])  # Output: "!dlroW ,olleH"
```

```
Hello
Hello
World!
World!
Hlo ol!
!dlroW ,olleH
```

```
#2. Explain the key features of list in python.

#Ans:- Lists are a fundamental data structure in Python.

#Key Features:
```

```
#1. Ordered collection: Elements are stored in a specific order.

#2. Mutable: Lists can be modified after creation.

#3. Indexed: Elements are accessed using their index.

#4. Dynamic size: Lists can grow or shrink dynamically.

#List Methods:

#1. append()
#2. extend()
#3. insert()
#4. remove()
#5. pop()
#6. index()
#7. count()
#8. sort()
#9. reverse()




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

#Ans:- ere's how to access, modify, and delete elements in a list:

#Accessing Elements

#1. Indexing: Use square brackets [] with the index.


my_list = [1, 2, 3, 4, 5]
print(my_list[0])  # Output: 1


#1. Negative Indexing: Use - to start from the end.


my_list = [1, 2, 3, 4, 5]
print(my_list[-1])  # Output: 5


#1. Slicing: Use start:stop or start:stop:step.


my_list = [1, 2, 3, 4, 5]
print(my_list[1:3])  # Output: [2, 3]


#Modifying Elements

#1. Assign new value: Use indexing.


my_list = [1, 2, 3, 4, 5]
my_list[0] = 10
print(my_list)  # Output: [10, 2, 3, 4, 5]


#1. Modify slice: Use slicing.
```

```
my_list = [1, 2, 3, 4, 5]
my_list[1:3] = [20, 30]
print(my_list)  # Output: [1, 20, 30, 4, 5]


#1. Append: Use append().


my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list)  # Output: [1, 2, 3, 4, 5, 6]


#1. Insert: Use insert().


my_list = [1, 2, 3, 4, 5]
my_list.insert(2, 10)
print(my_list)  # Output: [1, 2, 10, 3, 4, 5]


#Deleting Elements

#1. Remove by value: Use remove().


my_list = [1, 2, 3, 4, 5]
my_list.remove(3)
print(my_list)  # Output: [1, 2, 4, 5]


#1. Remove by index: Use pop().


my_list = [1, 2, 3, 4, 5]
my_list.pop(2)
print(my_list)  # Output: [1, 2, 4, 5]


#1. Delete slice: Use del.


my_list = [1, 2, 3, 4, 5]
del my_list[1:3]
print(my_list)  # Output: [1, 4, 5]
```

```
1
5
[2, 3]
[10, 2, 3, 4, 5]
[1, 20, 30, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 3, 4, 5]
[1, 2, 4, 5]
[1, 2, 4, 5]
[1, 4, 5]
```

```
#4. Compare and contrast tuple and lists with examples.


#Ans:- Tuples and lists are both data structures in Python.

Similarities:
```

#1. Ordered collection: Both store elements in a specific order.

#2. Indexed: Elements are accessed using their index.

#3. Dynamic size: Both can grow or shrink dynamically.

Differences:

#1. Immutability: Tuples are immutable (cannot be changed), while lists are mutable.

#2. Syntax: Tuples use parentheses (), while lists use square brackets [].

#3. Performance: Tuples are faster and more memory-efficient.

#Tuple Examples:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[0])  # Output: 1
print(my_tuple[1:3])  # Output: (2, 3)
```

#List Examples:

```
my_list = [1, 2, 3, 4, 5]
my_list[0] = 10  # Modify element
print(my_list)  # Output: [10, 2, 3, 4, 5]
my_list.append(6)  # Add element
print(my_list)  # Output: [10, 2, 3, 4, 5, 6]
```

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

#Ans:- Sets are an unordered collection of unique elements.

#Key Features:

#1. Unordered: Elements have no specific order.

#2. Unique: No duplicate elements allowed.

#3. Mutable: Sets can be modified.

#Set Operations:

#1. Union: Combines elements from two sets.

#2. Intersection: Returns common elements.

#3. Difference: Returns elements in one set but not the other.

#Set Methods:

#1. add(): Adds a single element.

#2. update(): Adds multiple elements.

#3. remove(): Removes an element.

```
#4. discard(): Removes an element if present.

#Examples:


# Create sets
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# Union
print(set1 | set2)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}

# Intersection
print(set1 & set2)  # Output: {4, 5}

# Difference
print(set1 - set2)  # Output: {1, 2, 3}

# Add element
set1.add(9)
print(set1)  # Output: {1, 2, 3, 4, 5, 9}

# Remove element
set1.remove(9)
print(set1)  # Output: {1, 2, 3, 4, 5}


#Use Cases:

#1. Removing duplicates from a list.


my_list = [1, 2, 2, 3, 4, 4, 5]
my_set = set(my_list)
print(my_set)  # Output: {1, 2, 3, 4, 5}


#1. Finding common elements between lists.


list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
set1 = set(list1)
set2 = set(list2)
print(set1 & set2)  # Output: {4, 5}
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5}
{1, 2, 3}
{1, 2, 3, 4, 5, 9}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{4, 5}
```

```
#6. Discuss the use cases of tuples and sets in python programing.

#Ans:- Tuples and sets are essential data structures in Python.

#Tuple Use Cases:

#1. Constant data: Store data that shouldn't be changed.

#2. High-performance applications: Tuples are faster than lists.
```

```
#3. Function arguments: Use tuples to pass multiple arguments.

#4. Data integrity: Ensure data consistency with immutable tuples.

#5. Dictionary keys: Tuples can be used as dictionary keys.

#Set Use Cases:

#1. Removing duplicates: Convert a list to a set to remove duplicates.

#2. Fast membership testing: Check if an element exists in a set.

#3. Intersection, union, and difference operations.

#4. Data validation: Ensure unique values with sets.

#5. Database query optimization.
```

```
#7. Describe how to add, modify, and delete item in a dictionary with examples.


#Ans:- Dictionaries are mutable data structures that store key-value pairs.

#Adding Items:

#1. Direct Assignment: dict[key] = value

my_dict = {"name": "John", "age": 30}
my_dict["city"] = "New York"
print(my_dict)  # Output: {"name": "John", "age": 30, "city": "New York"}

#1. Direct Assignment: dict[key] = new_value

my_dict = {"name": "John", "age": 30}
my_dict["age"] = 31
print(my_dict)  # Output: {"name": "John", "age": 31}

#Deleting Items:

#1. del Statement:

my_dict = {"name": "John", "age": 30}
del my_dict["age"]
print(my_dict)  # Output: {"name": "John"}
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
{'name': 'John', 'age': 31}
{'name': 'John'}
```

```
#8. Discus the importence of dictionary keys being immutable and provide examples.

#Ans:- In Python, dictionary keys must be immutable.

#Why Immutability Matters:

#1. Hashing: Dictionary keys are hashed to optimize lookup efficiency. Immutable keys ensure co

#2. Uniqueness: Immutable keys prevent accidental modifications, ensuring unique keys.

#3. Performance: Immutable keys reduce overhead from updating hash values.
```

```
#Examples of Immutable Keys:

#1. Strings:


my_dict = {"name": "John", "age": 30}


#1. Integers:


my_dict = {1: "John", 2: "Jane"}


#1. Tuples:


my_dict = {(1, 2): "John", (3, 4): "Jane"}

#Thank you
```

Start coding or generate with AI.