ds-q1

September 4, 2024

1. Discuss String Slicing and provide examples.

String slicing is a method of systematically extracting parts of a string based on specified indices. It allows you to create a substring from a given string by defining start and end points in the optional section. This approach is particularly useful for data manipulation, information processing, and analysis.

String slicing is accomplished using a syntax that specifies a range of indices within the string.

Syntax:

substring = original_string[start:stop:step]

1. start: The index at which the slice begins (inclusive).
2. stop: The index at which the slice ends (exclusive).
3. step: The interval between indices (optional).

Examples

1. Extracting a Substring

1 New Section

[ ]:
```
text = "Hello, World!"
substring = text[0:5]  # 'Hello'
print(substring)
```

```
Hello
```

Here, text[0:5] retrieves the characters from index 0 to 4 (not including 5).

2. Omitting the End Index

[ ]:
```
substring = text[7:]  # 'World!'
print(substring)
```

```
World!
```

If you omit the end index, slicing continues to the end of the string.

3. Omitting the Start and End Indices

[ ]:
```
substring = text[:]  # 'Hello, World!'
print(substring)
```

```
Hello, World!
```

When both indices are omitted, it returns a copy of the entire string.

4. Using Negative Indices

[ ]:
```
substring = text[-6:]  # 'World!'
print(substring)
```

```
World!
```

Negative indices count backward from the end of the string. -6 starts counting from the end, yielding 'World!'.

5. With a Step Value

[ ]:
```
substring = text[::2]  # 'Hlo ol!'
print(substring)
```

```
Hlo ol!
```

The step value of 2 means that every second character is included in the slice.

ds-q2

September 4, 2024

2. Explain the key features of list in Python

Python lists are a versatile and powerful data structure with several key features:

1. **Ordered**: The items in a list have a defined order, which remains consistent unless explicitly changed.
2. **Mutable**: You can modify a list after its creation by adding, removing, or changing items.
3. **Dynamic**: Lists can grow and shrink in size as needed, allowing for flexible data management.
4. **Heterogeneous**: A single list can contain items of different data types, such as integers, strings, and even other lists.
5. **Indexed**: Each item in a list has a specific index, starting from 0, which allows for easy access and manipulation of individual elements.
6. **Allow Duplicates**: Lists can contain multiple items with the same value.
7. **List Methods**: Python lists come with a variety of built-in methods for common operations. Examples .append(item): Adds an item to the end of the list.

.remove(item): Removes the first occurrence of an item.

.pop(index): Removes and returns the item at the specified index.

.sort(): Sorts the list in place.

.reverse(): Reverses the order of elements in the list.

8. **Nested Lists**:Lists can contain other lists, allowing for multi-dimensional data structures.

Example matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] print(matrix[1][2]) # Output: 6 (second row, third column)

ds-q3

September 4, 2024

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

Accessing Elements

We can access elements in a list using their index. Lists are starting with zero-indexed

[1]:
```
my_list = [10, 20, 30, 40, 50]

# Accessing the first element
print(my_list[0])  # Output: 10

# Accessing the last element
print(my_list[-1])  # Output: 50

# Accessing a range of elements (slicing)
print(my_list[1:3])  # Output: [20, 30]
```

```
10
50
[20, 30]
```

Modifying Elements

We can modify elements by assigning a new value to a specific index.

[2]:
```
my_list = [10, 20, 30, 40, 50]

# Modifying the second element
my_list[1] = 25
print(my_list)  # Output: [10, 25, 30, 40, 50]

# Modifying a range of elements
my_list[2:4] = [35, 45]
print(my_list)  # Output: [10, 25, 35, 45, 50]
```

```
[10, 25, 30, 40, 50]
[10, 25, 35, 45, 50]
```

Deleting Elements

You can delete elements using the del statement, remove() method, or pop() method.

[3]:
```python
my_list = [10, 20, 30, 40, 50]

# Deleting the third element
del my_list[2]
print(my_list)  # Output: [10, 20, 40, 50]

# Removing an element by value
my_list.remove(20)
print(my_list)  # Output: [10, 40, 50]

# Removing the last element
my_list.pop()
print(my_list)  # Output: [10, 40]

# Removing an element by index and returning it
removed_element = my_list.pop(0)
print(removed_element)  # Output: 10
print(my_list)  # Output: [40]
```

```
[10, 20, 40, 50]
[10, 40, 50]
[10, 40]
10
[40]
```

ds-q4

September 4, 2024

4. Compare and contrast tuples and lists with examples

In Python tuples and lists are both used to store collections of items, but they have key differences in their characteristics and uses. Here's a comparison of Python tuples and lists, along with examples to illustrate their similarities and differences:

Similarities

Ordered: Both tuples and lists maintain the order of elements.

Indexed: Elements in both can be accessed using indices.

Heterogeneous: Both can store elements of different data types.

Differences

1. **Syntax**

Lists: Defined with square brackets [ ].

Tuples: Defined with parentheses ( ).

[6]:
```
#List Syntax
fruits = ['apple', 'banana', 'cherry']
print(fruits)
```

```
['apple', 'banana', 'cherry']
```

[7]:
```
##Tuples Syntax
coordinates = (10, 20, 30)
print(coordinates)
```

```
(10, 20, 30)
```

2. **Mutability**

Lists: Mutable. You can modify, add, or remove elements after the list is created.

Tuples: Immutable. Once a tuple is created, you cannot change, add, or remove its elements.

[8]:
```
#List are Mutable
fruits = ['apple', 'banana', 'cherry']
fruits[1] = 'blueberry'  # Modify an element
fruits.append('date')    # Add a new element
print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'date']
```

```
['apple', 'blueberry', 'cherry', 'date']
```

[10]:
```
#Tuple are Immutable
numbers = (1, 2, 3)
#numbers[1] = 4  # This will raise a TypeError
# numbers.append(4)  # This will raise an AttributeError
```

3.**Performance**

Lists: Generally slower for operations that involve modifying the data due to their mutable nature.

Tuples: Faster and have a smaller memory footprint due to their immutability. This can be beneficial for performance in certain contexts.

[11]:
```
import timeit

# Timing list operations
list_time = timeit.timeit('a = [1, 2, 3]; a.append(4); a[1] = 5',␣
 ↪number=1000000)
print(f"List operation time: {list_time}")

# Timing tuple operations
tuple_time = timeit.timeit('a = (1, 2, 3); a + (4,)', number=1000000)
print(f"Tuple operation time: {tuple_time}")
```

```
List operation time: 0.1350315629999841
Tuple operation time: 0.07008489999998346
```

4. **Usage**

Lists: Use when you need a collection that can be modified, such as when elements need to be added, removed, or changed.

Tuples: Use when you need a collection that should remain constant. Useful for fixed data structures like coordinates or configuration data.

[13]:
```
#List Use Case
tasks = ['write code', 'review code', 'commit code']
tasks.append('deploy code')  # Task list can change
print(tasks)
```

```
['write code', 'review code', 'commit code', 'deploy code']
```

[14]:
```
#Tuple Use Case
point = (10, 20)  # Represents a fixed coordinate
```

5. **Methods**

Lists: Have a variety of methods such as .append(), .remove(), .pop(), .extend(), .sort(), etc.

Tuples: Have fewer methods due to their immutability. The most common methods are .count() and .index().

[18]:
```
#List Methods
fruits = ['apple', 'banana']
fruits.append('cherry')  # Adds an element
fruits.remove('banana')  # Removes an element
print(fruits)
```

```
['apple', 'cherry']
```

[16]:
```
#Tuple Methods
numbers = (1, 2, 3, 2, 2)
print(numbers.count(2))  # Output: 3
print(numbers.index(3))  # Output: 2
```

```
3
2
```

ds-q5

September 4, 2024

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

Sets in Python are a built-in data structure that represents an unordered collection of unique elements. They offer several key features and use cases. Here's a detailed look at the key features of sets along with examples:

Key Features of Sets

1. **Unordered Collection** : Sets do not maintain the order of elements. The order in which elements are stored and retrieved is not guaranteed.

[2]:
```
fruits = {'apple', 'banana', 'cherry'}
# Order of elements might vary when printed
print(fruits)
```

```
{'apple', 'cherry', 'banana'}
```

2. **Unique Elements**: Sets automatically remove duplicate elements. Each element in a set must be unique.

[3]:
```
numbers = {1, 2, 3, 2, 4, 1}
print(numbers)  # Output: {1, 2, 3, 4}
```

```
{1, 2, 3, 4}
```

3. **Mutable**: Sets are mutable; you can add or remove elements after creation.

[4]:
```
fruits = {'apple', 'banana'}
fruits.add('cherry')  # Add a new element
fruits.remove('banana')  # Remove an existing element
print(fruits)  # Output: {'apple', 'cherry'}
```

```
{'apple', 'cherry'}
```

4. **No Duplicate Elements**: If you attempt to add a duplicate element, it will not be added.

[5]:
```
colors = {'red', 'green', 'blue'}
colors.add('red')  # 'red' is already in the set, so no change
print(colors)  # Output: {'red', 'green', 'blue'}
```

```
{'green', 'blue', 'red'}
```

5. **Set Operations**: Sets support various mathematical set operations such as union, intersection, difference, and symmetric difference.

Examples:

[7]:
```
# Union: Combines elements from both sets.
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # or set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

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

[8]:
```
#Intersection: Returns elements that are common to both sets.
intersection_set = set1 & set2  # or set1.intersection(set2)
print(intersection_set)  # Output: {3}
```

```
{3}
```

[9]:
```
#Difference: Returns elements that are in the first set but not in the second␣
 ↪set.
difference_set = set1 - set2  # or set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

```
{1, 2}
```

[10]:
```
#Symmetric Difference: Returns elements that are in either set, but not in both.
symmetric_difference_set = set1 ^ set2  # or set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

```
{1, 2, 4, 5}
```

6.**Set Comprehensions** : Similar to list comprehensions, you can create sets using set comprehensions.

[11]:
```
squares = {x**2 for x in range(6)}
print(squares)  # Output: {0, 1, 4, 9, 16, 25}
```

```
{0, 1, 4, 9, 16, 25}
```

7. **Immutable Sets (frozenset)** : frozenset is a built-in type for immutable sets. Unlike regular sets, frozenset cannot be modified after creation.

[12]:
```
immutable_set = frozenset([1, 2, 3, 4])
# immutable_set.add(5)  # This will raise an AttributeError
```

8. **No Indexing** : Sets do not support indexing, slicing, or other sequence-like behavior because they are unordered.

[13]:
```
my_set = {'a', 'b', 'c'}
# my_set[0]  # This will raise a TypeError
```

1 Use Cases

Removing Duplicates: Sets are useful for removing duplicate items from a list.

[14]:
```
my_list = [1, 2, 2, 3, 4, 4, 5]
unique_set = set(my_list)
print(unique_set)  # Output: {1, 2, 3, 4, 5}
```

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

Membership Testing: Sets provide a fast way to check if an element is in a collection.

[15]:
```
my_set = {1, 2, 3, 4, 5}
print(3 in my_set)  # Output: True
print(6 in my_set)  # Output: False
```

```
True
False
```

ds-q6

September 4, 2024

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

1 Use Cases of Tuples

1. **Fixed Collections:** Tuples are ideal for storing data that should not change throughout the program. For example, coordinates (x, y), RGB color values, or configuration settings.

[4]:
```
coordinates = (10.0, 20.0)
color = (255, 0, 0)  # Red
location = (40.7128, -74.0060)  # Fixed geographical coordinates
```

```
(255, 0, 0)
```

2. **Returning Multiple Values:** Functions can return multiple values using tuples.

[6]:
```
def get_min_max(numbers):
    return min(numbers), max(numbers)

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

```
(1, 5)
```

3. **Dictionary Keys**: Tuples can be used as keys in dictionaries because they are immutable.

[7]:
```
student_grades = {("John", "Doe"): "A", ("Jane", "Smith"): "B"}
print(student_grades[("John", "Doe")])  # Output: A
```

```
A
```

4. **Data Integrity**: When you want to ensure that the data remains unchanged, tuples are a good choice.

[8]:
```
days_of_week = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",␣
↪"Saturday", "Sunday")
```

2 Use Cases of Sets

1. **Removing Duplicates**: Sets automatically remove duplicate items, making them useful for filtering unique elements from a list.

[9]:
```
my_list = [1, 2, 2, 3, 4, 4, 5]
unique_set = set(my_list)
print(unique_set)  # Output: {1, 2, 3, 4, 5}
```

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

2.**Membership Testing**: Sets provide a fast way to check if an element is in a collection.

[10]:
```
my_set = {1, 2, 3, 4, 5}
print(3 in my_set)  # Output: True
print(6 in my_set)  # Output: False
```

```
True
False
```

3. **Set Operations**: Sets support mathematical operations like union, intersection, difference, and symmetric difference, which are useful in various applications such as data analysis.

[11]:
```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
print(set_a | set_b)  # Output: {1, 2, 3, 4, 5}

# Intersection
print(set_a & set_b)  # Output: {3}

# Difference
print(set_a - set_b)  # Output: {1, 2}

# Symmetric Difference
print(set_a ^ set_b)  # Output: {1, 2, 4, 5}
```

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

4. **Unique Collections**: When you need to maintain a collection of unique items, sets are the perfect choice.

[12]:
```
unique_words = {"apple", "banana", "cherry"}
```

ds-q7

September 4, 2024

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

Dictionaries are mutable, we can add, modify, and delete items after the dictionary has been created. Here's a detailed explanation of how to perform these operations, along with examples:

1 Adding Items to a Dictionary

1. **Add a New Key-Value Pair**

To add a new key-value pair to a dictionary, you can assign a value to a new key. If the key does not already exist, it will be added to the dictionary.

[1]:
```
# Initial dictionary
person = {'name': 'Alice', 'age': 30}

# Adding a new key-value pair
person['city'] = 'New York'

print(person)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

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

2. **Add Multiple Key-Value Pairs Using the update() Method**

The update() method can be used to add multiple key-value pairs to a dictionary. It can take another dictionary or an iterable of key-value pairs as its argument.

[2]:
```
# Initial dictionary
person = {'name': 'Alice', 'age': 30}

# Adding multiple key-value pairs
person.update({'city': 'New York', 'job': 'Engineer'})

print(person)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York', 'job': 'Engineer'}
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York', 'job': 'Engineer'}
```

2 Modifying Items in a Dictionary

1. **Modify the Value of an Existing Key**

To modify the value associated with an existing key, simply assign a new value to that key.

[3]:
```
# Initial dictionary
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Modifying the value of an existing key
person['age'] = 31

print(person)
# Output: {'name': 'Alice', 'age': 31, 'city': 'New York'}
```

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

2.**Modify Multiple Values Using update()**

You can also use the update() method to change multiple values at once.

[4]:
```
# Initial dictionary
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Modifying multiple values
person.update({'age': 31, 'city': 'Los Angeles'})

print(person)
# Output: {'name': 'Alice', 'age': 31, 'city': 'Los Angeles'}
```

```
{'name': 'Alice', 'age': 31, 'city': 'Los Angeles'}
```

3 Deleting Items from a Dictionary

1.**Delete an Item Using the del Statement**

The del statement can be used to delete a specific key-value pair from the dictionary by specifying the key.

[5]:
```
# Initial dictionary
person = {'name': 'Alice', 'age': 31, 'city': 'Los Angeles'}

# Deleting a specific key-value pair
del person['city']

print(person)
# Output: {'name': 'Alice', 'age': 31}
```

```
{'name': 'Alice', 'age': 31}
```

2. **Remove an Item Using the pop() Method**

The pop() method removes a key-value pair by key and returns the value. If the key is not found, it raises a KeyError unless a default value is provided.

[6]:
```python
# Initial dictionary
person = {'name': 'Alice', 'age': 31, 'city': 'Los Angeles'}

# Removing and retrieving a value
city = person.pop('city')

print(city)
# Output: 'Los Angeles'
print(person)
# Output: {'name': 'Alice', 'age': 31}
```

```
Los Angeles
{'name': 'Alice', 'age': 31}
```

[7]:
```python
# Removing a key with a default value
job = person.pop('job', 'Not Found')

print(job)
# Output: 'Not Found'
```

```
Not Found
```

3. **Remove All Items Using the clear() Method**

The clear() method removes all items from the dictionary, leaving it empty.

[8]:
```python
# Initial dictionary
person = {'name': 'Alice', 'age': 31, 'city': 'Los Angeles'}

# Clearing all items
person.clear()

print(person)
# Output: {}
```

```
{}
```

ds-q8

September 4, 2024

Discuss the importance of dictionary keys being Immutable and provide example

Dictionary keys are required to be immutable. This immutability is crucial for several reasons related to the dictionary's performance and functionality. Here's a discussion on the importance of dictionary keys being immutable, along with examples to illustrate these points:

Importance of Dictionary Keys Being Immutable

1. **Hashing and Performance**

Dictionaries in Python use a hashing mechanism to quickly access values based on their keys. For this hashing to work correctly, the keys must be immutable. If a key could change after being added to the dictionary, it would disrupt the hashing process and lead to inconsistencies in retrieving the value.

[1]:
```
# Example with a tuple (immutable key)
my_dict = {('a', 'b'): 1}
print(my_dict[('a', 'b')])  # Output: 1

# Example with a list (mutable key - will raise an error)
try:
    my_dict = {[1, 2]: 1}
except TypeError as e:
    print(e)  # Output: unhashable type: 'list'
```

```
1
unhashable type: 'list'
```

2. **Consistency**

Immutable keys ensure that the dictionary's structure remains consistent. Since keys cannot be changed, the dictionary remains stable and reliable. This consistency is crucial for maintaining accurate mappings between keys and values.

[2]:
```
# Immutable key (tuple) maintains dictionary integrity
immutable_key = (1, 2)
my_dict = {immutable_key: 'value'}
print(my_dict[immutable_key])  # Output: 'value'

# Mutable key (list) would break dictionary integrity
try:
```

```
    mutable_key = [1, 2]
    my_dict = {mutable_key: 'value'}
except TypeError as e:
    print(e)  # Output: unhashable type: 'list'
```

```
value
unhashable type: 'list'
```

3. **Hashing Mechanism**

The hash function relies on the immutability of keys to produce a consistent hash value. If keys were mutable, the hash value could change, making it impossible to locate or update the value associated with that key.

[3]:
```
# Immutable key (string) ensures consistent hashing
my_dict = {'key': 100}
print(hash('key'))  # Output: consistent hash value

# Mutable key (list) cannot be hashed
try:
    hash([1, 2])
except TypeError as e:
    print(e)  # Output: unhashable type: 'list'
```

```
-1072515371879265403
unhashable type: 'list'
```

4. **Practical Use Cases**

Immutable keys allow dictionaries to be used in various practical applications, including as elements in sets or as keys in other dictionaries, where immutability is required.

[4]:
```
# Immutable keys can be used in sets
my_set = {('a', 1), ('b', 2)}
print(my_set)  # Output: {('a', 1), ('b', 2)}

# Mutable keys (lists) cannot be used in sets
try:
    my_set = {[1, 2], [3, 4]}
except TypeError as e:
    print(e)  # Output: unhashable type: 'list'
```

```
{('b', 2), ('a', 1)}
unhashable type: 'list'
```