MARCH 2024

Sequence Data Types

Notes - PYTHON O LEVEL

Akshar Computer Centre | +0191-2585266 | doeaccaksharcomputercentre.com

Table of Contents

Note: All the topics have subtopics and examples to know how things really working.

List, Tuple & Dictionary

List	An ordered collection that can contain elements of different data types. Lists are mutable, meaning their elements can be changed after they are created. The order of elements is preserved. my_list = [1, 2, 3, 4, 5]
Tuple	Similar to lists, but tuples are immutable, meaning their elements cannot be changed after they are created. Tuples are often used for fixed collections of items. The order of elements is preserved. my_tuple = (1, 2, 3, 4, 5)
Set	An unordered collection of unique elements. Sets are mutable, meaning elements can be added or removed. Sets do not allow duplicate elements. my_set = {1,2,3,4,5}
Dictionary	A collection of key-value pairs. Dictionaries are mutable and unordered. They are indexed by keys, which must be immutable, and the values associated with those keys can be of any data type. Duplicate keys are not allowed, but values can be duplicated. my_dict = {'a': 1, 'b': 2, 'c': 3}

Structure	Mutable	Ordered	Indexable	Properties	Commands	Typical Use Cases
List	Yes	Yes	Yes	Sequential	append(), insert(), pop(), remove(), extend(), sort()	Storing collections of items, iterating over elements, modifying sequences of data
Tuple	No	Yes	Yes	Sequential, Immutable	tuple(), unpacking, accessing elements	Storing fixed collections of items, returning multiple values from a function
Set	Yes	No	No	Unordered, Unique Elements	add(), remove(), discard(), union(), intersection(), difference()	Removing duplicates, checking membership, set operations
Dictionary	Yes	No	Yes (keys)	Key-Value pairs	keys(), values(), items(), get(), pop(), update()	Associating unique keys with values, fast lookup by key

Slicing, Indexing & Concatenation

Indexing

- Indexing refers to accessing individual elements of a sequence by their position.
- In Python, indexing starts from 0 for the first element, -1 for the last element, -2 for the second to last, and so on.
- You can access an element at a specific index using square brackets [].

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

Slicing

- Slicing allows you to access a subset of elements from a sequence by specifying a range of indices.
- It uses the syntax [start:end:step], where start is the starting index (inclusive), end is the ending index (exclusive), and step is the increment.
- If start or end is omitted, it defaults to the beginning or end of the sequence, respectively. If step is omitted, it defaults to 1.

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

Note: In dictionaries, you cannot access values using indexing but using keys.
my_dict = {'a': 1, 'b': 2, 'c': 3}

Accessing values using keys
print(my_dict['b']) # Output: 2

Slicing, Indexing & Concatenation

And other operations on Sequence datatype

Concatenation

- Concatenation is the process of combining two or more sequences to create a new sequence.
- For lists and tuples, concatenation is performed using the + operator.
- For strings, concatenation is also done with the + operator.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result_list = list1 + list2
print(result_list)    # Output: [1, 2, 3, 4, 5, 6]

string1 = "Hello"
string2 = "World"
result_string = string1 + " " + string2
print(result_string)    # Output: "Hello World"
```

Repetition

Repeat a sequence a certain number of times.

```
my_list = [1, 2, 3]
repeated_list = my_list * 3
print(repeated_list)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

my_tuple = ('a', 'b')
repeated_tuple = my_tuple * 2
print(repeated_tuple)  # Output: ('a', 'b', 'a', 'b')
```

Slicing, Indexing & Concatenation

And other operations on Sequence datatype

Membership Test

Check if an element is present in the sequence.

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

my_tuple = ('a', 'b', 'c', 'd', 'e')
print('f' in my_tuple)  # Output: False
```

Length

Determine the number of elements in the sequence.

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

my_tuple = ('a', 'b', 'c', 'd', 'e')
print(len(my_tuple))  # Output: 5

my_string = "Hello, World!"
print(len(my_string))  # Output: 13
```

Slicing, Indexing & Concatenation

And other operations on Sequence datatype

Iterating over elements

Access each element in the sequence one by one.

```python
my_list = [1, 2, 3, 4, 5]
for element in my_list:
    print(element)
```

Counting Occurrences

Count the number of occurrences of a specific element in the sequence.

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

my_tuple = ('a', 'b', 'c', 'a', 'd', 'a')
print(my_tuple.count('a'))  # Output: 3
```

Index of an element

Find the index of the first occurrence of a specified element.

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

Concept of Mutability

Mutability

Mutability refers to the ability of an object to be changed after it is created. In Python, certain data types like lists and dictionaries are mutable, meaning their contents can be modified after they are created. On the other hand, immutable data types like tuples and strings cannot be changed once they are created.

Here's a breakdown:

- Mutable: Objects that can be altered after they are created.
- Immutable: Objects that cannot be changed after they are created.

When a data structure is mutable, you can modify its elements, add new elements, or remove existing elements without creating a new object. This can be advantageous when you need to perform operations that require modifying the structure frequently.

For example, with lists:

```
my_list = [1, 2, 3]
my_list[0] = 100
print(my_list)  # Output: [100, 2, 3]
```

The above code modifies the first element of the list **my_list** from **1** to **100**.

However, with immutable objects like tuples, attempting to modify them directly results in an error:

```
my_tuple = (1, 2, 3)
my_tuple[0] = 100  # This will raise a TypeError
```

Minimum, Maximum & Mean

Minimum

min(): The min() function in Python returns the smallest item in an iterable (such as a list, tuple, or set) or the smallest of two or more arguments.

Maximum

max(): The max() function in Python returns the largest item in an iterable or the largest of two or more arguments.

Mean

mean(): The mean() function is not directly available in the Python standard library, but it can be computed using the sum() and len() functions. It calculates the arithmetic mean (average) of a sequence of numbers.

```python
numbers = [5, 2, 8, 1, 9]

# Maximum
max_num = max(numbers)
print(max_num)  # Output: 9

# Minimum
min_num = min(numbers)
print(min_num)  # Output: 1

# Mean
mean = sum(numbers) / len(numbers)
print(mean)  # Output: 5.0
```

Linear search

Linear Search

Linear search, also known as sequential search, is a method for finding a target value within a list or array. It works by sequentially checking each element of the list until a match is found or all elements have been checked.

Here's how linear search works:

1. Start from the beginning of the list.
2. Compare each element with the target value.
3. If the current element matches the target value, return its index.
4. If the end of the list is reached without finding a match, return a message indicating that the target value is not in the list.

Here's a Python implementation of linear search:

```python
def linear_search(sequence, target):
    for index, element in enumerate(sequence):
        if element == target:
            return index
    return -1  # Element not found

# Example usage:
my_list = [4, 2, 7, 1, 9, 5]
target_element = 7
index = linear_search(my_list, target_element)
if index != -1:
    print(f"Element {target_element} found at index {index}.")
else:
    print(f"Element {target_element} not found in the list.")

#Output: Element 7 found at index 2.
```

Linear search

Let's break down each part of the function.

def linear_search(sequence, target):

- **def** is the keyword used to define a function in Python.
- **linear_search** is the name of the function.
- **(sequence, target)** are the parameters that the function accepts. **sequence** represents the sequence (e.g., list, tuple, string) to search within, and **target** represents the value to search for within the sequence.

for index, element in enumerate(sequence):

- **for** is a keyword used to start a loop in Python.
- **index** and **element** are variables used for iteration. **index** represents the index of the current element in the sequence, and **element** represents the actual element at that index.
- **enumerate(sequence)** is a built-in Python function that returns pairs of index and corresponding values from the **sequence**. This allows iteration over both the index and the value of each element in the sequence.

if element == target:

- **if** is a keyword used to start a conditional statement in Python.
- **element == target** is a condition that checks if the current element being examined (**element**) is equal to the target value that we are searching for (**target**).

return index

- **return** is a keyword used to return a value from the function.
- **index** represents the index of the element in the sequence where the target value was found. If the condition **element == target** is **True**, it means the target value has been found, so the function returns the index of that element.

return -1

- This **return** statement is outside the **if** block, meaning it executes if the **if** condition (**element == target**) is not met during any iteration of the loop.
- **-1** is returned to indicate that the target value was not found in the sequence.

Counting frequency of elements

Counting the frequency of elements in a list using a dictionary

In the code give below,

- We define a list called **my_list** containing some elements.
- We initialize an empty dictionary called **frequency_dict** to store the frequencies of elements.
- We iterate through each item in the list.
- For each item, we check if it's already a key in the dictionary. If it is, we increment its count by 1. If it's not, we add it to the dictionary with a count of 1.
- Finally, we print out the frequency of each element by iterating through the dictionary.

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

frequency_dict = {}
for item in my_list:
    if item in frequency_dict:
        frequency_dict[item] += 1
    else:
        frequency_dict[item] = 1

print(frequency_dict)  # Output: {1: 2, 2: 4, 3: 2, 4: 1}
```

Counting frequency of elements

Explanation

```python
#Create a list
my_list = [1, 2, 3, 4, 2, 3, 1, 2, 2]

# Initialize an empty dictionary to store the frequencies
frequency_dict = {}

# Iterate through the list
for item in my_list:
  # Check if the item is already in the dictionary
    if item in frequency_dict:
        # If it is, increment its count
        frequency_dict[item] += 1
    else:
        # If it's not, add it to the dictionary with a count of 1
        frequency_dict[item] = 1

# Print the frequency of each element
print(frequency_dict)  # Output: {1: 2, 2: 4, 3: 2, 4: 1}
```

Thank you

Contact us for further inquiries

Akshar Computer Centre | +0191-2585266 | doeaccaksharcomputercentre.com