Data Structures in Python

Eng. Ali Ahmed

February 27, 2024

Agenda

1. Introduction to Data Structures...
2. Lists in python
3. Tuples
4. Sets
5. Dictionary

Introduction to Data Structures

- **Definition of Data Structures:** Data structures are a way of organizing and storing data to enable efficient access and modification. They are essential in programming for managing and manipulating data effectively.
- **Importance in Programming:** Efficient data structures are crucial for optimizing algorithms and improving overall program performance.
- **Types of Data Structures:** The basic Python data structures in Python include **list**, **set**, **tuple**, and **dictionary**. Each of the data structures is unique in its own way. Data structures are "containers" that organize and group data according to type.

List in python

- Lists are one of the most commonly used data structures in Python.
- They are ordered, mutable (changeable), and allow duplicate elements.
- Syntax for creating lists:
  - my_list = [1, 2, 3, 4, 5]
- Lists can contain elements of different data types.
  - mixed_list = [1, "hello", 3.5, True]

Accessing Elements

- Elements in a list are accessed using square brackets [ ] and indexing. Indexing starts from 0.
- Negative indexing is also supported, starting from -1.
- Examples
  - my_list = [10, 20, 30, 40, 50]
    print(my_list[0]) # Output: 10
    print(my_list[-1]) # Output: 50
- To create list of zeros with length 1000
  - list_of_zeros= [0]*1000
  - list=[[0]*8]*10 # 2D list of zeros with len 10

List Methods

- Various methods available to manipulate lists:
  - **append()**: Adds an element to the end of the list.
  - **extend()**: Extends the list by appending elements from another list.
  - **insert()**: Inserts an element at a specified position.
  - **remove()**: Removes the first occurrence of a value.
  - **pop()**: Removes and returns the element at a specified index.
  - **index()**: Returns the index of the first occurrence of a value.
  - **count()**: Returns the number of occurrences of a value.
  - **sort()**: Sorts the list in ascending order.
  - **reverse()**: Reverses the elements of the list.
- Examples
  - my_list = [3, 1, 2, 5, 4]
    my_list.sort()
    print(my_list) # Output: [1, 2, 3, 4, 5]

List Comprehensions

- List comprehensions provide a concise way to create lists.
- Examples
  - squares = [x**2 for x in range(10)]
    print(squares) # Output [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
  - even_numbers = [x for x in range(10) if x % 2 == 0]
    print(even_numbers) # Output [0, 2, 4, 6, 8]

Tuples

- Tuples are ordered, immutable (unchangeable) and allow duplicate elements.
- They are often used to store related pieces of information.
- Examples
  - my_tuple = (1, 2, 3)
- Also tuples can contain elements of different data types
  - mixed_tuple=(5,4,10.5,"mostafa",True)

  Elements in a tuple are accessed similar to lists using indexing.
  - my_tuple = (10, 20, 30, "mohamed",True)

```
print(my_tuple[0])      # Output: 10
print(my_tuple[-1])     # Output: True
print(my_tuple[:-2])    # Output: (10, 20, 30)
```

Tuple Methods

- Tuples have fewer methods compared to lists due to their immutability.
- Common methods include **count()** and **index()**.
- Examples
  - my_tuple = (1, 2, 3, 2)

```
print(my_tuple.count(2))      # Output: 2
print(my_tuple.count(5))      # Output: 0
print(my_tuple.index(3))      # Output: 2
print(my_tuple.index(10))     # Output: Error
```

Sets

- Sets are unordered collections of unique elements.
- They are mutable and support mathematical set operations.
- Examples
  - my_set = {5,4,3,2,0,1,0}
    print(my_set) #Output: {0, 1, 2, 3, 4, 5}
- Various set operations available:
  - **add()** , **remove()**, **discard()**, **pop()**, **clear()**
  - Mathematical set operations: **union()**, **intersection()**, **difference()**, **symmetric_difference()**

Set Operations

- Add (**add()**): Adds a single element to the set.
  - my_set = {1, 2, 3}
    my_set.add(4)
- Update **update()**: method is useful when you want to combine the elements of multiple sets or add elements from another iterable to an existing set.
  - set1 = {1, 2, 3}
    set2 = {4, 5, 6}
    set1.update(set2)
    print(set1) # Output: {1, 2, 3, 4, 5, 6}
- Remove (**remove()**): Removes a specified element from the set. Raises a KeyError if the element is not present.
  - my_set = {1, 2, 3}
    my_set.remove(2)

Set Operations (Cont..)

- Discard (**discard()**): Removes a specified element from the set. Does nothing if the element is not present.
  - my_set = {1, 2, 3}
    my_set.discard(2)
- Pop (**pop()**): Removes and returns an arbitrary element from the set.
  - my_set = {1, 2, 3}
    popped_element = my_set.pop()
- Clear (**clear()**): Removes all elements from the set.
  - my_set = {1, 2, 3}
    my_set.clear()

Mathematical set operations

- Union (**union()**): Combines elements from two sets, excluding duplicates.
  - set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    union_set = set1.union(set2)
    print(union_set) # Output: {1, 2, 3, 4, 5}
- Intersection (**intersection()**): Returns common elements present in both sets.
  - set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    intersection_set = set1.intersection(set2)
    print(intersection_set) # Output: {3}

Mathematical set operations (Cont...)

- Difference (**difference()**): Returns elements present in the first set but not in the second set.
  - set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    difference_set = set1.difference(set2)
    print(difference_set) # Output: {1, 2}
- Symmetric Difference (**symmetric_difference()**): Returns elements present in either set, but not in both.
  - set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    symmetric_difference_set = set1.symmetric_difference(set2)
    print(symmetric_difference_set) # Output: {1, 2, 4, 5}

Dictionary

- Dictionaries are collections of key-value pairs.
- They are mutable and can be indexed by keys.
- Syntax for creating dictionaries:
  - my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
- Elements in a dictionary are accessed using keys.
  - print(my_dict['name']) # Output: John

Dictionary Methods

- Keys (**keys()**): Returns a view of all keys in the dictionary.
  - print(my_dict.keys()) # Output: dict_keys(['name', 'age', 'city'])
- Values (**values()**): Returns a view of all values in the dictionary.
  - print(my_dict.values()) # Output: dict_values(['John', 30, 'New York'])
- Items (**items()**): Returns a view of all key-value pairs in the dictionary.
  - print(my_dict.items()) # Output: dict_items([('name', 'John'), ('age', 30), ('city', 'New York')])
- Update (**update()**): Updates the dictionary with key-value pairs from another dictionary or iterable.
  - other_dict = 'gender': 'Male'
    my_dict.update(other_dict)
    print(my_dict) # Output: {'name': 'John', 'age': 30, 'city': 'New York', 'gender': 'Male'}

Dictionary Methods

- Pop (**pop()**): Removes the specified key and returns its associated value. Raises a KeyError if the key is not found.
  - age = my_dict.pop('age')
    print(age) # Output: 30
- Popitem (**popitem()**): Removes and returns an arbitrary key-value pair as a tuple. Raises a KeyError if the dictionary is empty.
  - item = my_dict.popitem()
    print(item) # Output: ('city', 'New York')
- Copy (**copy()**): Copy items from dictionary to another one.
  - coped_dict=my_dict.copy()
    coped_dict.update("gender":"male")
    print(my_dict) # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
    print(coped_dict) # Output: {'name': 'John', 'age': 30, 'city': 'New York', 'gender': 'male'}
- Clear (**clear()**): Removes all elements from the dictionary.
  - my_dict.clear()