Basic Data Structures

Lecture Flow

- Lists
- Tuples
- Sets
- Dictionaries

Lists

What are lists?

- Lists are fundamental data structures in Python used to store collections of data.
- They can hold items of any data type, including numbers, strings, and even other lists.
- Lists are ordered, changeable, and allow duplicate values.

Creating lists

- Lists can be created using square **brackets []** and separating items with commas.
- The **list()** constructor can also be used to create lists.

```
# Creating a list using square brackets
fruits = ["apple", "banana", "cherry"]

# Convert iterables to list using the list() constructor
numbers = list((1, 2, 3, 4, 5))
```

List data types

List items can be of any data type

- `list1 = ["apple", "banana", "cherry"]`
- `list2 = [1, 5, 7, 9, 3]`
- `list3 = [True, False, False]`
- `list4 = ["abc", 34, True, 40, "male"]`

Accessing List Items

- List items are accessed using their index number, starting from 0.
- Negative indexing can be used to access items from the end of the list.

```
nums = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

# Accessing the first item

nums[0]             # 10
# Accessing the last item

nums[-1]        # 19
```

Slicing Lists

- Slicing allows extracting a sublist from a list.
- Slicing uses the **colon (:)** to separate start and end indices (inclusive).

```
nums = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
# Extracting a sublist from index 2 to index 4

nums[2:5]          # [12, 13, 14]

nums[-4 : -1]  ??
```

Modifying Lists

- Lists are mutable, allowing you to change their contents.
- You can modify items using their index or extend the list using **append**() and **insert**().
- You can also remove items using **remove**() and **pop**().

Examples

```
fruits = ["apple", "banana", "cherry"]
# Changing the first item

fruits[0] = "orange"  # fruits = ["orange", "banana", "cherry"]

# Adding an item to the end

fruits.append("mango")  # fruits = ["orange", "banana", "cherry", "mango"]

# Removing an item by value

fruits.remove("cherry")  # fruits = ["orange", "banana", "mango"]

# Removing the last item

removed_item = fruits.pop()  # removed_item = "mango", fruits =
["orange", "banana"]
```

Common List Operations

- Checking if an item exists: in keyword
- Sorting a list: **sort**() method
- **sorted** (nums , key = myFunction ( ), reverse = True/False)
- Reversing a list: **reverse**() method

Examples

```
fruits = ["orange", "banana"]
# Checking if "apple" exists in the list
if "apple" in fruits:
    print("Yes, apple is in the list")
# Sorting the list in ascending order
fruits.sort()  # fruits = ["banana", "orange"]
# Reversing the sorted list
fruits.reverse()  # fruits = ["orange", "banana"]
```

Combining Lists

- Concatenating lists using the + operator or extend() method
- Adding items from one list to another individually

Examples

```
numbers = [1, 2, 3]

fruits = ["orange", "banana"]
# Concatenating lists using '+' operator

new_list = fruits + numbers  # new_list = ["orange", "banana", 1, 2, 3]

# Extending a list using extend() method

fruits.extend(numbers)  # fruits = ["orange", "banana", 1, 2, 3]
```

Traversing Lists

- Iterating through lists using for loops
- Accessing both index and value using enumerate() function

- ```
  for index in range(len(nums)):
      print(nums[index])
  ```

- ```
  for num in nums:
      print(num)
  ```

- ```
  for index, num in enumerate(nums):
      print(index, num)
  ```
  ```

List Comprehension

- Creating new lists based on existing lists
- Using expressions and conditions to filter and transform list elements

```
# Creating a list of even numbers from a list of numbers
numbers = [1, 2, 3, 4, 5]
even_numbers = [num for num in numbers if num % 2 == 0]
# even_numbers = [2, 4]
```

Why List Comprehension?

```
my_list = [[0]] * 5
my_list = ?  #[[0], [0], [0], [0], [0]]
my_list[0][0] = 1
my_list = ?
```

Why List Comprehension?

```
my_list = [[0]] * 5
my_list[0][0] = 1
my_list = [[1], [1], [1], [1], [1]] # Why?
```

Other List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuples

What are Tuples?

- A tuple is a collection which is ordered, allows duplicates and is **unchangeable**. Tuples are also known as Immutable Lists.

- Tuples are written with round brackets.
  - fruits = ("apple", "banana", "cherry")
  - fruit = ("apple",)

Creating Tuples

- Tuples are written with round brackets **()**.
- This is called ‘packing’ a tuple.

```
fruits = ("apple", "banana", "cherry")

fruit = ("apple",) # or just () to create an empty one
```

- The **tuple()** constructor:

```
fruits = tuple(["apple", "banana", "cherry"])

numbers = tuple()
```

Unpacking tuples

- In Python, we are also allowed to extract the values back into variables. This is called "unpacking".

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

fruits = ("apple", "banana", "cherry", “oranges”, “pineapples”)

green, yellow, *red = fruits
```

Unpacking tuples

- In Python, we are also allowed to extract the values back into variables. This is called "unpacking".

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

fruits = ("apple", "banana", "cherry", “oranges”, “pineapples”)

(green, yellow, *red) = fruits  #red = [“cherry”,“oranges”, “pineapples”]
```

Tuples

- Is it possible to
  - add an element to a Tuple? How?
  - delete an element?
  - join two tuples?

Tuple Similarities with List

- Similar data types
- Slicing and Indexing
- Similar Iteration

Q: Is it possible to have “Tuple Comprehension” ?

Tuple Methods

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Practice Problems

List Comprehension
Runner-up Score
Nested LIsts
Lists

Sets

What Is Set?

Set

- A set is a built-in data type in Python that represents an unordered, unindexed collection of unique and unchangeable elements.

Initialization

- set() constructor

```
empty_set = set() #empty set
numbers_set = set([1, 2, 3, 4, 5]) #
Converts any iterable to a set
```

- Curly braces { }

```
numbers_set = {1, 2, 3, 4, 5}  # Creates a set with
elements
empty_set = {} # Valid ?
```

What Data Type Can Set Store?

- It can store elements of various data types, as long as those elements are immutable.
- Integers, Floats, Strings,Tuples and Booleans.

```
my_set = {1, 2.5, “apple”, (1, 2, 3), True}

invalid_set = {1, 2, [3, 4]} # Why invalid?
```

Basics of how Set works

- Sets in Python use a hash table to store elements which is like a big storage box with lots of compartments (buckets).
- When you add an element to the set, a special function (hash function) turns the element into a unique code that determines which bucket the element goes into.
- When you want to check if an element is in the set (lookup), the hash function is used again to find the compartment where that element should be.

Access Elements

- You cannot access items in a set by referring to an index or a key.
- To check if an item is in a set, you use the in operator.

```
my_set = {1, 2, 3, 4, 5}
if 3 in my_set:
    #code
```

- You can iterate through all items in a set using a for-in loop

```
for item in my_set:
    #code
```

Add And Remove Elements

- The add() method is used to add a single element to a set.
- The update() method is used to add multiple elements using iterables (lists, tuples)
- The union() method Return a set containing the union of sets

```
my_set = set()
my_set.add(1)
my_set.add(2)
my_set.add(3)
print(my_set) # {1,2,3}
```

```
my_set.update([3, 4, 5, 6])
print(my_set) # {1, 2, 3, 4, 5, 6}
```

Add And Remove Elements

- The clear() method Removes all the elements from the set.
- The remove() and discard() methods Remove the specified item.
- pop() Removes an element from the set.

```
my_set = {1,2,3,4,5}
my_set.remove(2) # ?
my_set.remove(6) # ?
my_set.discard(2) # ?
my_set.discard(6) # ?
```

```
my_set.pop() # ?
my_set.clear() # ?
```

What Are Valid And Invalid Operators In Set?

Valid Operators

- **Union** (|)
  ```
  union_set = set1 | set2
  ```

- **Intersection** (&)
  ```
  intersection_set = set1 & set2
  ```

- **Difference** (-)
  ```
  difference_set = set1 - set2
  ```

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
```

Valid Operators

Symmetric Difference (^)

```
    symmetric_difference_set = set1 ^ set2
```

Subset (<=) and Superset (>=)

```
    is_subset = set1 <= set2
    is_superset = set1 >= set2
```

Equality(==)

```
    set1 == set2
```

Invalid Operators

- **Concatenation** (**+**)
- **Multiplication** (*****)
- **Indexing** ([ ]) **and** **Slicing** ([:])

Set Comprehension

- A set comprehension in Python is a concise way to create a set using curly braces {} .

```
Set_name = {expression for item in iterable if condition}

squares_set = {x**2 for x in range(10)}
squares_set = {x**2 for x in range(10) if x % 2}
```

Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Advantage of sets

1. Uniqueness of Elements
2. Fast Membership Testing
3. Mathematical Set Operations

Frozenset

- a frozenset is an immutable and hashable version of a set.

```
frozen_set = frozenset([1, 2, 3])
frozen_set = frozenset([1, 2, 3])

frozen_set.add(4) # possible?
union_frozenset = frozenset1 | frozenset2 # possible?
intersection_frozenset = frozenset1 & frozenset2 # possible?
```

Exercises

1. Check if All the Integers in a Range Are Covered - LeetCode
2. Union of two arrays | Practice | GeeksforGeeks
3. Check if two arrays are equal or not | Practice | GeeksforGeeks
4. Array Subset of another array | Practice | GeeksforGeeks

Dictionaries

What Is Dictionary?

Dictionary

- A dictionary is an ordered (as of Python version 3.7) collection of key-value pairs, where each key must be unique and is associated with a specific value.
- They are also mutable and dynamic data structures

Initialization

- dict() constructor

```
empty_dictionary = dict() #empty dictionary
dictionary_from_list = dict([('name', 'John'),('age',
30),('city','New York')])
```

- Curly braces { }

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

- Dictionary comprehension

```
sqr_dict = {num: num**2 for num in range(1, 11)}
```

What data types can be used as dictionary keys in Python?

- Dictionary keys must be immutable data types to maintain data integrity.
- Integers, Floats, Strings,Tuples and Booleans.

```
my_dictionary = {(1, 2): "Tuple Key", False: "Key"}
my_dictionary = {[1, 2]: "List Key"} #why invalid?
```

Common Operations In Dictionary

- **Access**

```
age = my_dict["age"]
age = my_dict.get("age", 0) #why safe?
```

- **Add or Update**

```
my_dict["age"] = 20
my_dict["age"] = my_dict.get("age",0) + 10
```

- **Removing Key**

```
value = my_dict.pop("age")
```

Common Operations In Dictionary

- **Checking if the key exist**

```
if "age" in my_dict
```

- **Iterating**
  - Through Keys:
    ```
    for key in my_dict:
    ```
  - Through key-value pairs:
    ```
    for key, value in my_dict.items():
    ```
  - Through values:
    ```
    for value in my_dict.values():
    ```

Dictionary Copying

- **Assignment Operator (=):**

```
my_dict1 = {'key1': 'value1', 'key2': 'value2'}
my_dict2 = my_dict1
my_dict2['key1'] = 'value3'
print(my_dict1) # Output?
```

  - **Note:** In Python, when you use the assignment operator (=) to assign any object to another variable, both variables reference the same underlying iterable in memory. modifications made to the iterable through one variable will be visible when accessing the iterable through the other variable, and vice versa.

Shallow Copy

- A shallow copy creates a new dictionary but does not create new copies of the objects inside the dictionary.
- Changes made to mutable objects (like lists) within the original dictionary will affect the corresponding objects in the copied dictionary, and vice versa.

Shallow Copy

- Shallow copy is performed using methods like copy(), dict(), or dictionary unpacking ({**original_dict}).

Shallow Copy

```
original_dict = {'key1': ['value1'], 'key2': 'value2'}
shallow_copied_dict = original_dict.copy()

original_dict['key1'].append('value3')
Original_dict['key2'] = ‘value4’

print(shallow_copied_dict)  # Output?
```

Deep Copy

- A deep copy creates a new dictionary and recursively creates new copies of all objects inside the original dictionary, including nested objects.
- Changes made to mutable objects within the original dictionary will not affect the corresponding objects in the copied dictionary, and vice versa.

Deep Copy

- Deep copy is performed using the deepcopy function from the copy module.

Deep Copy

```
import copy

original_dict = {'key1': ['value1'], 'key2': 'value2'}
deep_copied_dict = copy.deepcopy(original_dict)

original_dict['key1'].append('value3')

print(deep_copied_dict)  # Output?
```

Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Advantage of Dictionaries

1. Efficient Data Retrieval
2. Fast Membership Testing
3. Dynamic Size

Exercises

1. Missing Number - LeetCode
2. Find Players With Zero or One Losses - LeetCode
3. Day 8: Dictionaries and Maps | HackerRank

“Success is neither magical nor mysterious. Success is the natural consequence of consistently applying the basic fundamentals.”

- Jim Rohn

Africa To Silicon Valley