PYTHON PART - 2

Python, including methods, examples, and practical use cases. This overview aims to provide a comprehensive understanding within a reasonable length.

List

Description: A list in Python is a mutable, ordered collection of items. It allows duplicate elements and provides various methods for manipulation and traversal.

Syntax: Lists are defined using square brackets `[]`.

Example:

python
```
my_list = [1, 2, 3, 4, 5]
```

Methods and Usage:

- **Append**: Adds an element to the end of the list.

  python
  ```
  my_list.append(6)
  # Result: [1, 2, 3, 4, 5, 6]
  ```

- **Extend**: Appends elements from another iterable to the end of the list.

  python
  ```
  another_list = [7, 8, 9]
  my_list.extend(another_list)
  # Result: [1, 2, 3, 4, 5, 6, 7, 8, 9]
  ```

- **Insert**: Inserts an element at a specified position.

  python
  ```
  my_list.insert(2, 10)
  # Result: [1, 2, 10, 3, 4, 5, 6, 7, 8, 9]
  ```

- **Remove**: Removes the first occurrence of a value.

  python
  ```
  my_list.remove(3)
  # Result: [1, 2, 10, 4, 5, 6, 7, 8, 9]
  ```

- **Pop**: Removes and returns an element at a specified index (default is last element).

  python

```
popped_value = my_list.pop(1)
# Result: popped_value = 2, my_list = [1, 10, 4, 5, 6, 7, 8, 9]
```

- **Index**: Returns the index of the first occurrence of a value.

  python
  ```
  index = my_list.index(5)
  # Result: index = 4
  ```

- **Count**: Returns the number of occurrences of a value.

  python
  ```
  count = my_list.count(4)
  # Result: count = 1
  ```

- **Sort**: Sorts the list in place.

  python
  ```
  my_list.sort()
  # Result: [1, 4, 5, 6, 7, 8, 9, 10]
  ```

- **Reverse**: Reverses the elements of the list in place.

  python
  ```
  my_list.reverse()
  # Result: [10, 9, 8, 7, 6, 5, 4, 1]
  ```

- **Copy**: Returns a shallow copy of the list.

  python
  ```
  copied_list = my_list.copy()
  # Result: copied_list = [10, 9, 8, 7, 6, 5, 4, 1]
  ```

Use Cases: Lists are commonly used for:

- Storing collections of similar items.
- Maintaining ordered sequences of data.
- Dynamically building and modifying datasets.

Tuple

Description: A tuple in Python is an immutable, ordered collection of items. Once created, its elements cannot be changed.

PYTHON PART - 2

Syntax: Tuples are defined using parentheses `()`.

Example:

python
```
my_tuple = (1, 2, 3, 4, 5)
```

Methods and Usage:

Count:

- **Description**: Returns the number of occurrences of a specified value in the tuple.
- **Syntax**: tuple.count(value)
- **Example**:

  python
  ```
  my_tuple = (1, 2, 2, 3, 4, 2)
  count = my_tuple.count(2)
  print(count)  # Output: 3
  ```

Index:

- **Description**: Returns the index of the first occurrence of a specified value.
- **Syntax**: tuple.index(value[, start[, end]])
- **Parameters**:
  - value: Required. The value to search for in the tuple.
  - start: Optional. The index at which to start the search.
  - end: Optional. The index at which to end the search.
- **Example**:

  python
  ```
  my_tuple = (1, 2, 3, 4, 5, 2)
  index = my_tuple.index(2)
  print(index)  # Output: 1 (index of the first occurrence of 2)
  ```

Concatenation:

- **Description**: Tuples can be concatenated using the + operator to create a new tuple.
- **Example**:

  python
  ```
  tuple1 = (1, 2, 3)
  tuple2 = (4, 5, 6)
  concatenated_tuple = tuple1 + tuple2
  print(concatenated_tuple)  # Output: (1, 2, 3, 4, 5, 6)
  ```

PYTHON PART - 2

Repetition:

- **Description**: Tuples can be repeated using the `*` operator to create a new tuple with repeated elements.
- **Example**:

```
python
tuple1 = ('a', 'b')
repeated_tuple = tuple1 * 3
print(repeated_tuple)  # Output: ('a', 'b', 'a', 'b', 'a', 'b')
```

Membership Test (in and not in):

- **Description**: Tuples support membership testing to check if a value exists in the tuple.
- **Example**:

```
python
my_tuple = (1, 2, 3, 4, 5)
print(3 in my_tuple)    # Output: True
print(6 not in my_tuple)  # Output: True
```

Use Cases: Tuples are useful for:

- Storing data that should not be changed, such as configuration settings.
- Representing fixed collections of elements like coordinates or dimensions.

Set

Description: A set in Python is an unordered collection of unique elements. It is mutable, allowing for dynamic addition and removal of items.

Syntax: Sets are defined using curly braces `{}` or the `set()` function.

Example:

```
python
my_set = {1, 2, 3, 4, 5}
```

Methods and Usage:

- **Add**: Adds an element to the set.

```
python
my_set.add(6)
# Result: {1, 2, 3, 4, 5, 6}
```

- **Remove**: Removes a specified element from the set. Raises KeyError if the element is not present.

  ```
  python
  my_set.remove(3)
  # Result: {1, 2, 4, 5, 6}
  ```

- **Discard**: Removes a specified element from the set if it is present.

  ```
  python
  my_set.discard(2)
  # Result: {1, 4, 5, 6}
  ```

- **Union**: Returns a new set with elements from both sets.

  ```
  python
  set1 = {1, 2, 3}
  set2 = {3, 4, 5}
  union_set = set1.union(set2)
  # Result: {1, 2, 3, 4, 5}
  ```

- **Intersection**: Returns a new set with elements common to both sets.

  ```
  python
  intersection_set = set1.intersection(set2)
  # Result: {3}
  ```

- **Difference**: Returns a new set with elements in the set that are not in the other set.

  ```
  python
  difference_set = set1.difference(set2)
  # Result: {1, 2}
  ```

- **Clear**: Removes all elements from the set.

  ```
  python
  my_set.clear()
  # Result: set()
  ```

Use Cases: Sets are ideal for:

- Checking membership efficiently (due to hash-based storage).
- Performing operations like union, intersection, and difference.
- Removing duplicates from a sequence.

PYTHON PART - 2

Dictionary

Description: A dictionary in Python is a mutable, unordered collection of key-value pairs. Each key must be unique, but values can be duplicated.

Syntax: Dictionaries are defined using curly braces {}, with key-value pairs separated by colons : (key: value).

Example:

python

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

Methods and Usage:

- **Accessing Values**: Retrieve values using keys.

  python

  ```
  print(my_dict['name'])  # Accessing value by key
  # Result: 'Alice'
  ```

- **Adding or Modifying Entries**:

  python

  ```
  my_dict['gender'] = 'Female'  # Adding a new key-value pair
  my_dict['age'] = 31  # Modifying an existing value
  ```

- **Removing Entries**:

  python

  ```
  del my_dict['city']  # Deleting a specific key-value pair
  my_dict.pop('age')   # Removing and returning the value of a specific key
  ```

- **Keys and Values**:

  python

  ```
  keys = my_dict.keys()    # Returns a view object of all keys
  values = my_dict.values()  # Returns a view object of all values
  ```

- **Clear**: Removes all elements from the dictionary.

  python

  ```
  my_dict.clear()
  # Result: {}
  ```

- **Copy**: Returns a shallow copy of the dictionary.

```
python
copied_dict = my_dict.copy()
# Result: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

Use Cases: Dictionaries are commonly used for:

- Storing data with a unique identifier (key) for efficient retrieval.
- Representing structured data such as JSON objects.
- Mapping relationships between entities.

SLICING

Slicing is a powerful operation in Python that allows you to extract a portion of a sequence like strings, lists, tuples, or other iterable objects. It provides a flexible way to access multiple elements based on their indices. Here’s a detailed explanation of slicing and its application with examples:

Syntax of Slicing

The syntax for slicing follows the general pattern:

```
python
sequence[start:stop:step]
```

- start: Optional. The starting index of the slice. If omitted, it defaults to 0 (beginning of the sequence).
- stop: Required. The ending index of the slice. The slice extends up to, but does not include, this index.
- step: Optional. The step size used to skip elements in the sequence. If omitted, it defaults to 1.

Examples of Slicing

Let's explore slicing with different types of sequences:

1. Slicing Lists

```
python
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get a slice from index 2 to 5 (exclusive)
slice1 = my_list[2:5]
print(slice1)  # Output: [3, 4, 5]

# Get elements from the beginning to index 4 (exclusive)
```

```
slice2 = my_list[:4]
print(slice2)  # Output: [1, 2, 3, 4]

# Get elements from index 5 to the end
slice3 = my_list[5:]
print(slice3)  # Output: [6, 7, 8, 9]

# Get every second element starting from index 1
slice4 = my_list[1::2]
print(slice4)  # Output: [2, 4, 6, 8]
```

2. Slicing Tuples

python

```
my_tuple = (1, 2, 3, 4, 5)

# Get a slice from index 1 to 4 (exclusive)
slice_tuple = my_tuple[1:4]
print(slice_tuple)  # Output: (2, 3, 4)
```

3. Slicing Strings

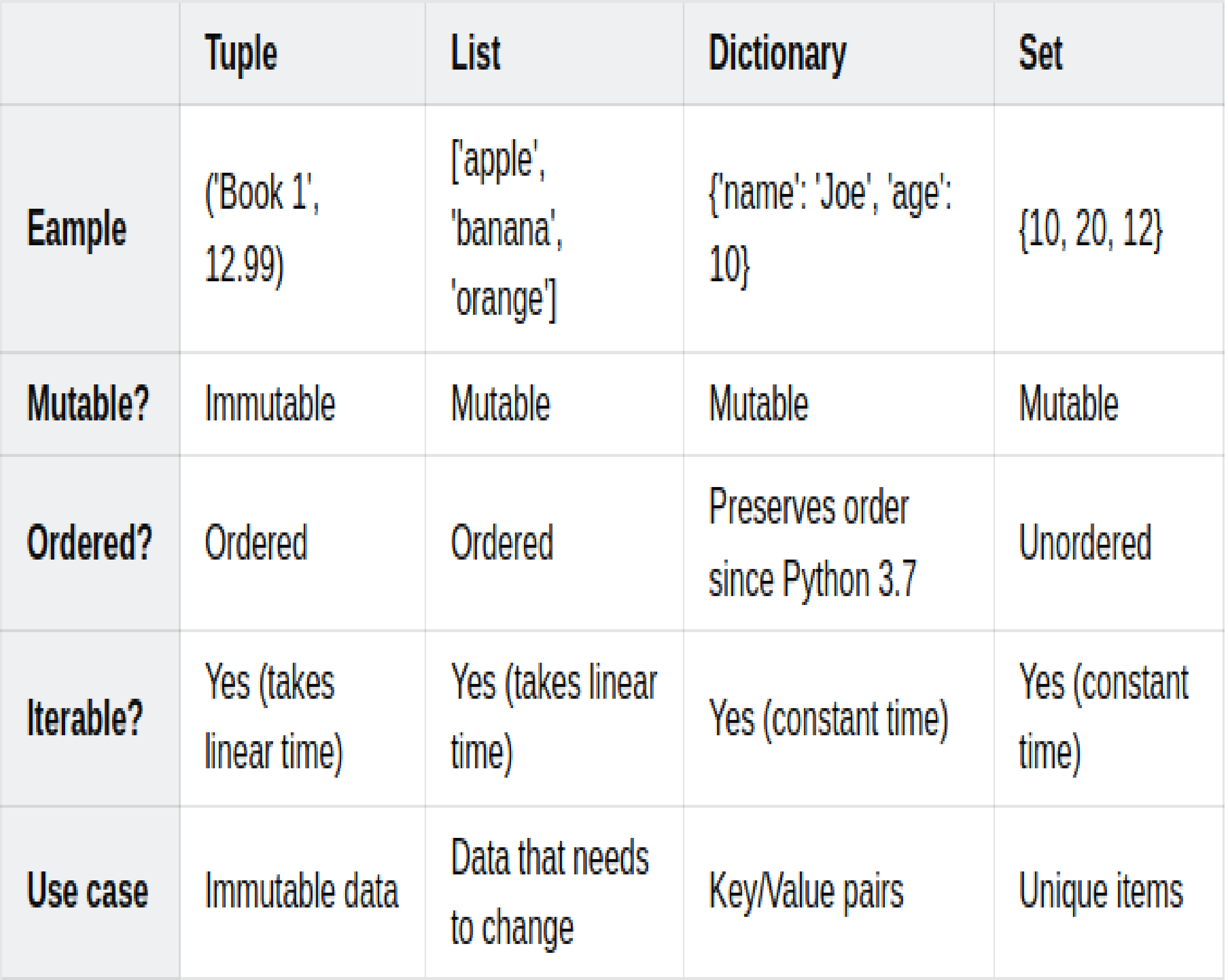
python

```
my_string = "Hello, World!"

# Get a slice from index 1 to 8 (exclusive)
slice_string = my_string[1:8]
print(slice_string)  # Output: "ello, W"

# Get every second character starting from index 0
slice_string2 = my_string[::2]
print(slice_string2)  # Output: "Hlo ol!"
```

PYTHON PART - 2



	Tuple	List	Dictionary	Set
Eample	('Book 1', 12.99)	['apple', 'banana', 'orange']	{'name': 'Joe', 'age': 10}	{10, 20, 12}
Mutable?	Immutable	Mutable	Mutable	Mutable
Ordered?	Ordered	Ordered	Preserves order since Python 3.7	Unordered
Iterable?	Yes (takes linear time)	Yes (takes linear time)	Yes (constant time)	Yes (constant time)
Use case	Immutable data	Data that needs to change	Key/Value pairs	Unique items