Python Data Structures

strings, lists, tuples, and directionaries

Table of contents

1 Common Methods

1.1 List Methods

Assume initial `lst` is `[1, 2, 3]` before each method's usage example.

Method	Description	Usage Example	Output
`append(x)`	Adds an item `x` to the end of the list.	`lst.append(9)`	`[1, 2, 3, 9]`
`extend(iterable)`	Adds all items in `iterable` to the list.	`lst.extend([10, 11])`	`[1, 2, 3, 9, 10, 11]`
`insert(i, x)`	Inserts an item `x` at a given position `i`.	`lst.insert(1, 'a')`	`[1, 'a', 2, 3, 9, 10, 11]`
`remove(x)`	Removes the first item from the list whose value is `x`.	`lst.remove('a')`	`[1, 2, 3, 9, 10, 11]`

Method	Description	Usage Example	Output
`pop([i])`	Removes the item at the given position in the list and returns it. If no index is specified, removes and returns the last item.	`lst.pop()`	`11` (list becomes `[1, 2, 3, 9, 10]`)
`clear()`	Removes all items from the list.	`lst.clear()`	`[]`
`index(x)`	Returns the index of the first item whose value is `x`.	`lst.index(9)`	`3`
`count(x)`	Returns the number of times `x` appears in the list.	`lst.count(2)`	`1`
`sort(key=None, reverse=False)`	Sorts the items of the list in place.	`lst.sort()`	`[1, 2, 3, 9, 10]` (if `lst` is reset before use)
`reverse()`	Reverses the elements of the list in place.	`lst.reverse()`	`[10, 9, 3, 2, 1]` (if `lst` is reset before use)
`copy()`	Returns a shallow copy of the list.	`lst2 = lst.copy()`	`[1, 2, 3, 9, 10]` (creates `lst2`)

1.1.1 Shallow Copy

In Python, a shallow copy creates a new object, but the new object's contents reference the same memory locations as the original object's contents.

- **New object:** A shallow copy creates a distinct object, meaning changes to the copy itself will not affect the original.
- **Shared references:** The elements within the new object (if it is a container like a list or dictionary) point to the same underlying data as the elements in the original object.
- **Mutable elements:** If any of the elements within the container are mutable (like lists or dictionaries), modifying them through the copy will also modify them in the original.
- **Common use cases:** Shallow copies are suitable when you want a separate object but do not anticipate modifying the nested, mutable elements. They are also faster to create than deep copies.

```python
original_list = [[1, 2], 3, 4]
shallow_copy = original_list.copy()
print("Original list:", original_list)
print("Shallow copy:", shallow_copy)
```

```
Original list: [[1, 2], 3, 4]
Shallow copy: [[1, 2], 3, 4]
```

```python
# Modifying the copy itself
shallow_copy.append(5)
print("Original list:", original_list)
print("Shallow copy:", shallow_copy)
```

```
Original list: [[1, 2], 3, 4]
Shallow copy: [[1, 2], 3, 4, 5]
```

```python
# Modifying a mutable element within the copy
shallow_copy[0].append(3)
print("Original list:", original_list)
print("Shallow copy:", shallow_copy)
```

```
Original list: [[1, 2, 3], 3, 4]
Shallow copy: [[1, 2, 3], 3, 4, 5]
```

Note: If you need to ensure that modifications to nested, mutable elements in the copy do not affect the original, you should use a deep copy instead.

Reference: copy in Python (Deep Copy and Shallow Copy)

1.2 Dictionary Methods

Assume initial `dct` is `{'apple': 2, 'banana': 3}` before each method's usage example.

Method	Description	Usage Example	Output
`get(key)`	Returns the value for `key` if `key` is in the dictionary, else `None`.	`dct.get('apple')`	`2`
`keys()`	Returns a view object displaying a list of all the keys.	`dct.keys()`	`dict_keys(['apple', 'banana'])`
`values()`	Returns a view object displaying a list of all the values.	`dct.values()`	`dict_values([2, 3])`
`items()`	Returns a view object containing a tuple for each key-value pair.	`dct.items()`	`dict_items([('apple', 2), ('banana', 3)])`
`update([other])`	Updates the dictionary with the key/value pairs from `other`, overwriting existing keys.	`dct.update({'cherry': 5})`	`{... 'cherry': 5}`

Method	Description	Usage Example	Output
`pop(key)`	Removes the specified key and returns the corresponding value. If key is not found, `d` is returned if given, otherwise KeyError is raised.	`dct.pop('apple')`	2 (dict becomes `{'banana': 3, 'cherry': 5}`)
`popitem()`	Removes and returns a `(key, value)` pair as a 2-tuple. Pairs are returned in LIFO order.	`dct.popitem()`	`('cherry', 5)`
`clear()`	Removes all items from the dictionary.	`dct.clear()`	`{}`
`copy()`	Returns a shallow copy of the dictionary.	`dct2 = dct.copy()`	`{'banana': 3}` (creates `dct2`)
`setdefault(key, default=None)`	If `key` is in the dictionary, return its value. If not, insert `key` with a value of `default` and return `default`.	`dct.setdefault('banana', 5)`	3

1.3 Tuple Methods

Assume `tpl` is `(1, 2, 3)` for the usage example.

Method	Description	Usage Example	Output
`count(x)`	Returns the number of times `x` appears in the tuple.	`tpl.count(1)`	`1`
`index(x)`	Finds the first occurrence of `x` in the tuple and returns its index.	`tpl.index(3)`	`2`

2 Exercises

2.1 Exercise: String Reversal

Reverse a given string without using loops or built-in functions.

- **Sample Input**: `'hello'`
- **Sample Output**: `'olleh'`.

2.1.1 Solution:

```python
input_string = 'hello'
input_string[::-1]
```

```
'olleh'
```

2.1.2 Explanation:

String slicing allows you to reverse a string with `[::-1]`, where `:` specifies the whole string and `-1` dictates the step, reversing the order.

2.2 Exercise: Palindrome Check

Check if a given string is a palindrome without using loops.

- **Sample Input**: `'radar'`
- **Sample Output**: `True`.

2.2.1 Solution:

```python
input_string = 'radar'
input_string == input_string[::-1]
```

```
True
```

2.2.2 Explanation:

A palindrome reads the same backward as forward. Comparing the original string with its reversed version checks for this condition.

2.3 Exercise: List Sum

Find the sum of elements in a list without using loops.

- **Sample Input**: `[1, 2, 3, 4, 5]`
- **Sample Output**: `15`.

2.3.1 Solution:

```
numbers = [1, 2, 3, 4, 5]
sum(numbers)
```

```
15
```

2.3.2 Explanation:

The `sum` function calculates the total of all numbers in the list directly.

2.4 Exercise: List Element Swap

Swap the second and last elements of a list without using additional variables.

- **Sample Input**: `[1, 2, 3, 4]`
- **Sample Output**: `[1, 4, 3, 2]`

2.4.1 Solution:

```
lst = [1, 2, 3, 4]
lst[1], lst[-1] = lst[-1], lst[1]
lst
```

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

2.4.2 Explanation:

This solution uses tuple unpacking to swap the values of the second and last elements in the list, demonstrating an efficient way to rearrange elements.

2.5 Exercise: Unique Elements

Create a list of unique elements from the given list without using loops.

- **Sample Input**: `[1, 2, 2, 3, 3, 3, 4]`
- **Sample Output**: `[1, 2, 3, 4]`

2.5.1 Solution:

```python
lst = [1, 2, 2, 3, 3, 3, 4]
unique_lst = list(set(lst))
unique_lst
```

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

2.5.2 Explanation:

Converts the list to a set, using list elements as set entries, then converts the set keys back to a list.

2.6 Exercise: Accessing Dictionary Elements

Access a specific element by key in a dictionary.

- **Sample Input**: `{'name': 'John', 'age': 30}`, Key: `'age'`
- **Sample Output**: `30`.

2.6.1 Solution:

```python
person = {'name': 'John', 'age': 30}
person.get('age')
```

```
30
```

2.6.2 Explanation:

The `get` method accesses the value for a given key.

2.7 Exercise: Merge Two Dictionaries

Merge two dictionaries into one without using loops.

- **Sample Input**: `dict1 = {'a': 1, 'b': 2}, dict2 = {'c': 3, 'd': 4}`
- **Sample Output**: `{'a': 1, 'b': 2, 'c': 3, 'd': 4}`

2.7.1 Solution:

```python
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
{**dict1, **dict2}
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

2.7.2 Explanation:

The `**` operator unpacks the dictionaries, and combining them in `{}` creates a new dictionary containing all pairs.

2.8 Exercise: Tuple Swap

Swap the first and last elements of a tuple.

- **Sample Input**: `(1, 2, 3, 4)`
- **Sample Output**: `(4, 2, 3, 1)`

2.8.1 Solution:

```python
original_tuple = (1, 2, 3, 4)
original_tuple[-1:] + original_tuple[1:-1] + original_tuple[:1]
original_tuple
```

```
(1, 2, 3, 4)
```

2.8.2 Explanation:

Slicing and concatenating tuples allows swapping the first and last elements without additional variables.

2.9 Exercise: Nested Data Extraction

Extract a value from a nested dictionary using a list of keys.

- **Sample Input**: `data = {'a': {'b': {'c': 'd'}}}`, `keys = ['a', 'b', 'c']`
- **Sample Output**: `'d'`

2.9.1 Solution:

```python
data = {'a': {'b': {'c': 'd'}}}
keys = ['a', 'b', 'c']
data[keys[0]][keys[1]][keys[2]]
```

```
'd'
```

2.9.2 Explanation:

Sequential access using keys from the list navigates through the nested dictionaries to the desired value.

2.10 Exercise: Tracing String Operations

2.10.1 Code

```python
s = 'Python'
output = s[1:4] + s[:2] + s[-2:]
```

2.10.2 Solution

```
'ythPyon'
```

2.10.3 Explanation

The code concatenates slices of the string `'Python'`. The slices are `'yth'` (`s[1:4]`), `'Py'` (`s[:2]`), and `'on'` (`s[-2:]`), resulting in `'ythPyon'`. Slicing allows you to extract parts of a string. This exercise demonstrates how slices can be combined to rearrange and create new strings.

2.11 Exercise: Tracing List Operations

2.11.1 Code

```
lst = [1, 2, [3, 4], (5, 6, 7)]
output = lst[2][1] + lst[3][1]
```

2.11.2 Solution

```
10
```

2.11.3 Explanation

The output is `10`, as it adds the second element of the third item (`list [3, 4]`) and the second element of the fourth item (`tuple (5, 6, 7)`). This shows how to access nested data structures. The exercise highlights indexing within complex data types.

2.12 Exercise: Tracing Dictionary Operations

2.12.1 Code

```
d = {'a': 1, 'b': 2, 'c': 3}
keys = list(d.keys())
output = d[keys[1]] + d.get('c', 0) + len(keys)
```

2.12.2 Solution

```
8
```

2.12.3 Explanation

The output is 8, coming from adding the value of `'b' (2)`, `'c' (3)`, and the number of keys `(3)`. The exercise illustrates dictionary key access, the use of the `get` method for safe value retrieval, and how to work with dictionary keys as a list.

2.13 Exercise: Tracing Tuple and String Operations

2.13.1 Code

```python
t = ('a', 'b', 'c', 'd', 'e')
output = t[1:-1] + tuple('x') + t[:1]
```

2.13.2 Solution:

```
('b', 'c', 'd', 'x', 'a')
```

2.13.3 Explanation:

The output is `('b', 'c', 'd', 'x', 'a')`, showcasing tuple slicing and concatenation to reorder and modify tuples. This exercise demonstrates slicing tuples to extract parts, adding elements by converting a string to a tuple, and appending tuples to each other.