Tuple Coding Exercise:

1. Elementwise sum:

   Create a function that takes two tuples and returns a tuple containing the element-wise sum of the input tuples.

   Example

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
output_tuple = tuple_elementwise_sum(tuple1, tuple2)
print(output_tuple)  # Expected output: (5, 7, 9)
```

   SOLUTION 1

```
def tuple_elementwise_sum(t1, t2):
    if len(t1) != len(t2):
        raise ValueError("Input tuples must have the same length.")

    result = tuple(a + b for a, b in zip(t1, t2))
    return result
```

   Explanation

   `def tuple_elementwise_sum(t1, t2):` - Define a function called "tuple_elementwise_sum" that takes two tuples "t1" and "t2" as arguments.

   `if len(t1) != len(t2):` - Check if the lengths of the input tuples are not equal.

   `raise ValueError("Input tuples must have the same length.")` - If the lengths of the tuples are not equal, raise a ValueError with a descriptive message.

   `result = tuple(a + b for a, b in zip(t1, t2))` - Use the `zip` function to pair corresponding elements of the input tuples, then use a generator expression to compute the element-wise sum, and finally pass the generator expression to the `tuple` constructor to create a new tuple with the sums.

   `return result` - Return the resulting tuple containing the element-wise sums.

   Time and Space Complexity

`if len(t1) != len(t2):` - Constant time complexity O(1) for checking the lengths of the tuples. Constant space complexity O(1) as it does not use any additional memory.

`raise ValueError("Input tuples must have the same length.")` - Constant time complexity O(1) for raising an exception. Constant space complexity O(1) for creating the exception object.

`result = tuple(a + b for a, b in zip(t1, t2))` - Linear time complexity O(n), where n is the length of the input tuples, as it iterates through each pair of elements and computes the element-wise sum. Linear space complexity O(n) because it creates a new tuple with the same length as the input tuples to store the element-wise sums.

`return result` - Constant time complexity O(1) for returning the result. No additional space complexity.

Overall time complexity of the function is O(n) because it iterates through each pair of elements in the input tuples once. The overall space complexity is O(n) because the function creates a new tuple with the same length as the input tuples to store the element-wise sums.

SOLUTION 2

```
def tuple_elementwise_sum(tuple1, tuple2):
   return tuple(map(sum, zip(tuple1, tuple2)))
```

Explanation

`def tuple_elementwise_sum(tuple1, tuple2):` - Define a function called "tuple_elementwise_sum" that takes two tuples "tuple1" and "tuple2" as arguments.

`return tuple(map(sum, zip(tuple1, tuple2)))` - This line has multiple operations: a. `zip(tuple1, tuple2)` - Use the `zip` function to pair corresponding elements of the input tuples. The `zip` function returns an iterator over these pairs. b. `map(sum, zip(tuple1, tuple2))` - Use the `map` function to apply the `sum` function to each pair of elements created by the `zip` function. The `map` function returns an iterator that applies the `sum` function to each pair, resulting in the element-wise sums. c. `tuple(map(sum, zip(tuple1, tuple2)))` - Pass the iterator returned by the `map` function to the `tuple` constructor to create a new tuple with the element-wise sums. d. `return tuple(map(sum, zip(tuple1, tuple2)))` - Return the resulting tuple containing the element-wise sums.

Time and Space Complexity

`def tuple_elementwise_sum(tuple1, tuple2):` - Define a function called "tuple_elementwise_sum" that takes two tuples "tuple1" and "tuple2" as arguments. No time or space complexity associated with this line as it is just a function definition.

`return tuple(map(sum, zip(tuple1, tuple2)))` - This line has multiple operations: a. `zip(tuple1, tuple2)` - The `zip` function has a linear time complexity O(n), where n is the length of the input tuples, as it iterates through each element in both input tuples to create pairs. The space complexity is also O(n) because it creates an iterator containing n pairs. b. `map(sum, zip(tuple1, tuple2))` - The `map` function has a linear time complexity O(n), as it applies the `sum` function to each pair created by the `zip` function. The space complexity is O(n) because it creates an iterator containing n element-wise sums. c. `tuple(map(sum, zip(tuple1, tuple2)))` - The `tuple` constructor has a linear time complexity O(n) because it iterates through the iterator returned by the `map` function to create a new tuple. The space complexity is O(n) because it creates a new tuple with n element-wise sums.

The time complexities of the `zip`, `map`, and `tuple` operations are all linear, O(n), but they are combined in a single line, so the overall time complexity for this line is still O(n).

The overall time complexity of the function is O(n) because it iterates through each pair of elements in the input tuples once. The overall space complexity is O(n) because the function creates a new tuple with the same length as the input tuples to store the element-wise sums.

2. Insert at the beginning:

Write a function that takes a tuple and a value, and returns a new tuple with the value inserted at the beginning of the original tuple.

Example

```
input_tuple = (2, 3, 4)
value_to_insert = 1
output_tuple = insert_value_front(input_tuple, value_to_insert)
print(output_tuple)  # Expected output: (1, 2, 3, 4)
```

SOLUTION - Time and Space Complexity of Insert at the Beginning

```
def insert_value_at_beginning(input_tuple, value_to_insert):
    return (value_to_insert,) + input_tuple
```

Explanation

`def insert_value_at_beginning(input_tuple, value_to_insert):` - Define a function called "insert_value_at_beginning" that takes a tuple "`input_tuple`" and a value "`value_to_insert`" as arguments.

`return (value_to_insert,) + input_tuple` - Create a new tuple with the given value as the first element and concatenate the original tuple "`input_tuple`" to it. The comma after the value is necessary to create a single-element tuple. Return the new tuple.

Time and Space Complexity

`def insert_value_at_beginning(input_tuple, value_to_insert):` - Define a function called "insert_value_at_beginning" that takes a tuple "`input_tuple`" and a value "`value_to_insert`" as arguments. No time or space complexity associated with this line as it is just a function definition.

`return (value_to_insert,) + input_tuple` - This line creates a new tuple with the given value as the first element followed by the elements of the original tuple. The tuple concatenation operation has a linear time complexity O(n), where n is the length of the input tuple, as it creates a new tuple by copying the elements from the original tuple. The space complexity is also O(n) because it creates a new tuple with n+1 elements.

The overall time complexity of the function is O(n) because it iterates through the elements of the input tuple once to create a new tuple. The overall space complexity is O(n) because it creates a new tuple with n+1 elements.

3. Concatenate

Concatenate

Write a function that takes a tuple of strings and concatenates them, separating each string with a space.

Example

```
input_tuple = ('Hello', 'World', 'from', 'Python')
output_string = concatenate_strings(input_tuple)
print(output_string)  # Expecte

def concatenate_strings(input_tuple):
   return ' '.join(input_tuple)
```

Explanation

`def concatenate_strings(input_tuple):` - Define a function called "concatenate_strings" that takes a tuple of strings "input_tuple" as an argument.

`return ' '.join(input_tuple)` - Use the `join` method on a space character `' '` to concatenate the strings in the input tuple "input_tuple" with a space as the separator. Return the resulting concatenated string.

Time and Space Complexity

`def concatenate_strings(input_tuple):` - Define a function called "concatenate_strings" that takes a tuple of strings "input_tuple" as an argument. No time or space complexity associated with this line as it is just a function definition.

`return ' '.join(input_tuple)` - This line uses the `join` method on a space character `' '` to concatenate the strings in the input tuple "input_tuple" with a space as the separator. The `join` method has a linear time complexity O(n) because it iterates through each string in the input tuple. The space complexity is also O(n) because it creates a new concatenated string with the length equal to the sum of the lengths of the strings in the input tuple plus the spaces in between.

The overall time complexity of the function is O(n) because it iterates through the strings in the input tuple once to create a new concatenated string. The overall space complexity is O(n) because it creates a new concatenated string with the length equal to the sum of the lengths of the strings in the input tuple plus the spaces in between.

4. Diagonal

Diagonal

Create a function that takes a tuple of tuples and returns a tuple containing the diagonal elements of the input.

Example

```
input_tuple = (
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9)
)
output_tuple = get_diagonal(input_tuple)
print(output_tuple)  # Expected output: (1, 5, 9)
```

SOLUTION - Time and Space Complexity of Diagonal

```
def get_diagonal(input_tuple):
```

```
    return tuple(input_tuple[i][i] for i in
range(len(input_tuple)))
```

Explanation

`def get_diagonal(input_tuple):` - Define a function called "get_diagonal" that takes a tuple of tuples "input_tuple" as an argument.

`return tuple(input_tuple[i][i] for i in range(len(input_tuple)))` - Use a generator expression to iterate through the indices `i` from `0` to the length of the input tuple minus one, and select the diagonal elements by indexing the inner tuples with the same index `i`. Create a new tuple containing the selected diagonal elements and return it.

Time and Space Complexity

`def get_diagonal(input_tuple):` - Define a function called "get_diagonal" that takes a tuple of tuples "input_tuple" as an argument. No time or space complexity associated with this line as it is just a function definition.

`return tuple(input_tuple[i][i] for i in range(len(input_tuple)))` - This line uses a generator expression to iterate through the indices `i` from `0` to the length of the input tuple minus one, and select the diagonal elements by indexing the inner tuples with the same index `i`. The time complexity is O(n), where n is the length of the input tuple because it iterates through the indices once. The space complexity is O(n) because it creates a new tuple containing the diagonal elements, which has a length equal to the length of the input tuple.

The overall time complexity of the function is O(n) because it iterates through the indices of the input tuple once to create a new tuple with the diagonal elements. The overall space complexity is O(n) because it creates a new tuple containing the diagonal elements, which has a length equal to the length of the input tuple.

5. Common Elements

Common Elements

Write a function that takes two tuples and returns a tuple containing the common elements of the input tuples.

Example

```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (4, 5, 6, 7, 8)
output_tuple = common_elements(tuple1, tuple2)
print(output_tuple)  # Expected output: (4, 5)
```

SOLUTION - Time and Space Complexity of Common Elements

```
def common_elements(tuple1, tuple2):
```

```
    return tuple(set(tuple1) & set(tuple2))
```

Explanation

`def common_elements(tuple1, tuple2):` - Define a function called "common_elements" that takes two tuples "tuple1" and "tuple2" as arguments.

`return tuple(set(tuple1) & set(tuple2))` - Convert both input tuples into sets using the `set()` constructor, then use the set intersection operator `&` to find the common elements between the two sets. Convert the resulting set of common elements back to a tuple and return it.

Time and Space Complexity

`def common_elements(tuple1, tuple2):` - Define a function called "common_elements" that takes two tuples "tuple1" and "tuple2" as arguments. No time or space complexity associated with this line as it is just a function definition.

`return tuple(set(tuple1) & set(tuple2))` - This line creates two sets from the input tuples using the `set()` constructor, and then computes the set intersection using the `&` operator. The time complexity of creating each set is O(n), where n is the length of the input tuple. The time complexity of computing the set intersection is O(min(n,m)), where m is the length of the second input tuple. Since the two input tuples are of equal length, the overall time complexity of the function is O(n). The space complexity is also O(n) because the size of the resulting set will be no larger than the size of the smaller of the two input tuples.

Therefore, the overall time complexity of the function is O(n), and the overall space complexity is also O(n), where n is the length of the input tuples.