The result for the unnormalized attention score is

```
tensor(1.8524)
```

Again, we can generalize this computation to all attention scores via matrix multiplication:

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

All attention scores for given query

As we can see, as a quick check, the second element in the output matches the `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Now, we want to go from the attention scores to the attention weights, as illustrated in figure 3.16. We compute the attention weights by scaling the attention scores and using the softmax function. However, now we scale the attention scores by dividing them by the square root of the embedding dimension of the keys (taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

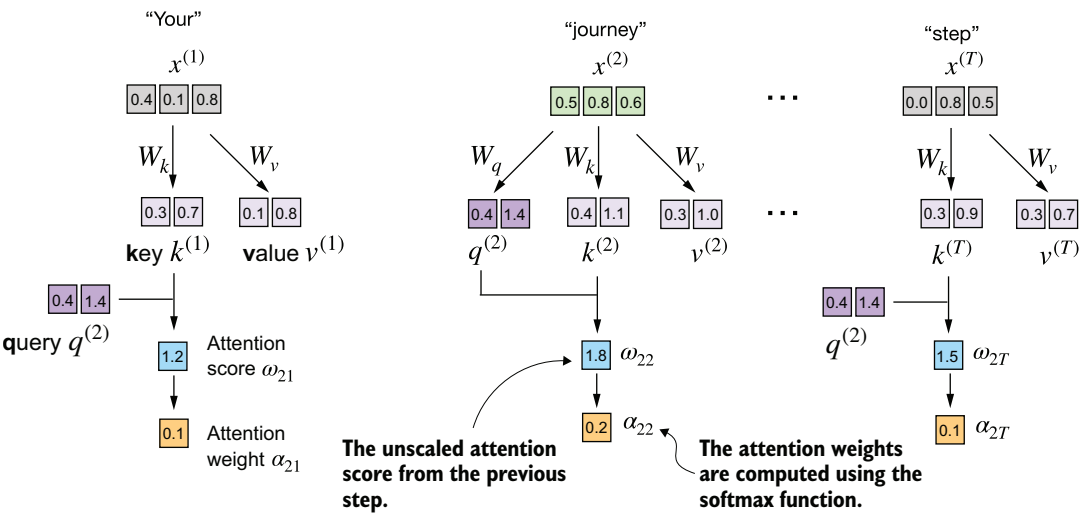

Figure 3.16 After computing the attention scores ω, the next step is to normalize these scores using the softmax function to obtain the attention weights α.

The resulting attention weights are

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than 1,000 for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in figure 3.17.

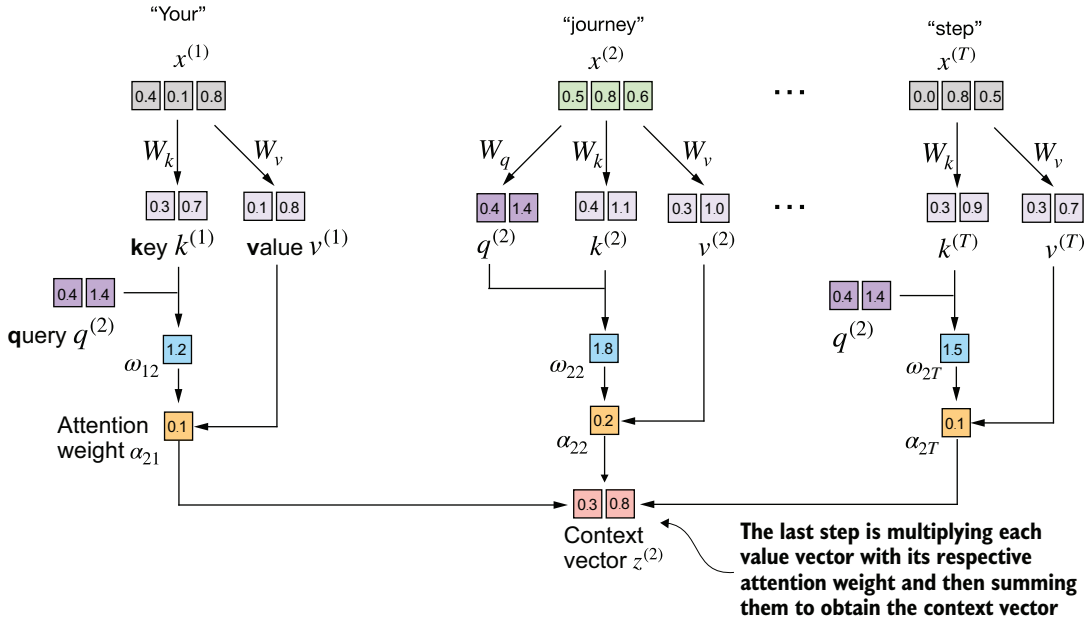

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.

Similar to when we computed the context vector as a weighted sum over the input vectors (see section 3.3), we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs

the respective importance of each value vector. Also as before, we can use matrix multiplication to obtain the output in one step:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061, 0.8210])
```

So far, we've only computed a single context vector, $z^{(2)}$. Next, we will generalize the code to compute all context vectors in the input sequence, $z^{(1)}$ to $z^{(T)}$.

Why query, key, and value?

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A *query* is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The *key* is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match the query.

The *value* in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

3.4.2 Implementing a compact self-attention Python class

At this point, we have gone through a lot of steps to compute the self-attention outputs. We did so mainly for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class, as shown in the following listing.

Listing 3.1 A compact self-attention class

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
```