Let's implement a complete tokenizer class in Python with an `encode` method that splits text into tokens and carries out the string-to-integer mapping to produce token IDs via the vocabulary. In addition, we'll implement a `decode` method that carries out the reverse integer-to-string mapping to convert the token IDs back into text. The following listing shows the code for this tokenizer implementation.

Listing 2.3 Implementing a simple text tokenizer

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

Stores the vocabulary as a class attribute for access in the encode and decode methods

Creates an inverse vocabulary that maps token IDs back to the original text tokens

Processes input text into token IDs

Converts token IDs back into text

Removes spaces before the specified punctuation

Using the `SimpleTokenizerV1` Python class, we can now instantiate new tokenizer objects via an existing vocabulary, which we can then use to encode and decode text, as illustrated in figure 2.8.

Let's instantiate a new tokenizer object from the `SimpleTokenizerV1` class and tokenize a passage from Edith Wharton's short story to try it out in practice:

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,"
       Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)
```

The preceding code prints the following token IDs:

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108,
754, 793, 7]
```

Next, let's see whether we can turn these token IDs back into text using the decode method:

```
print(tokenizer.decode(ids))
```

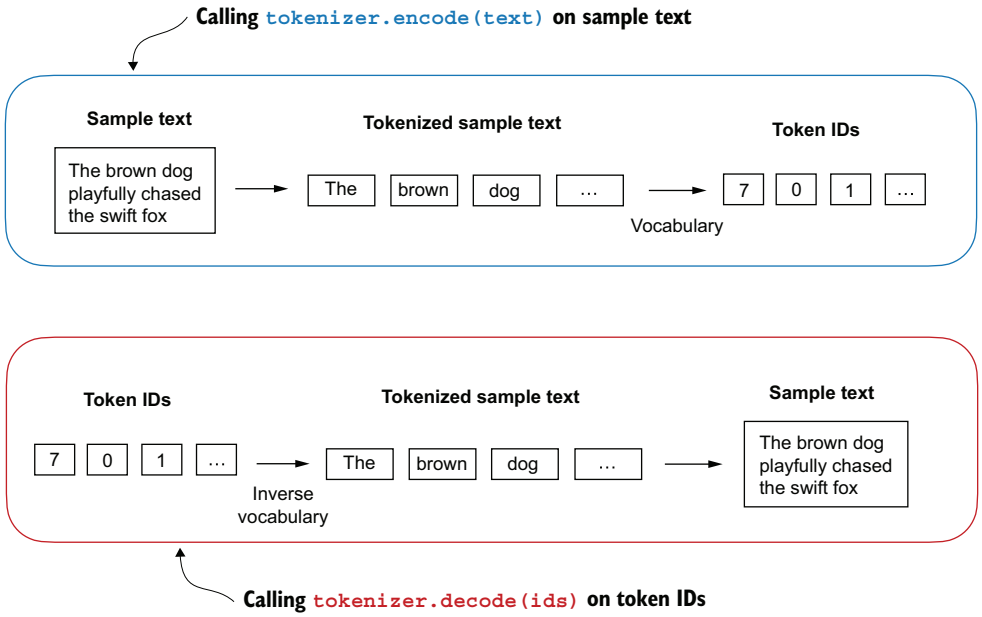

Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text tokens, and concatenates the text tokens into natural text.

This outputs:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with
pardonable pride.'
```

Based on this output, we can see that the decode method successfully converted the token IDs back into the original text.

So far, so good. We implemented a tokenizer capable of tokenizing and detokenizing text based on a snippet from the training set. Let's now apply it to a new text sample not contained in the training set:

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))
```

Executing this code will result in the following error:

```
KeyError: 'Hello'
```

The problem is that the word "Hello" was not used in the "The Verdict" short story. Hence, it is not contained in the vocabulary. This highlights the need to consider large and diverse training sets to extend the vocabulary when working on LLMs.