

Substituting this back into the expression for (S_n) , we get:

$$S_n = \sum_{k=1}^n \sqrt{(2k-1)^2 + \left(\frac{17(2k-1)}{n^2}\right)^2} = \sqrt{n^4 + 289}$$

We need $(\sqrt{n^4 + 289})$ to be an integer. Let (m) be this integer, so:

$$m^2 = n^4 + 289$$

$$m^2 - n^4 = 289$$

This can be factored as:

$$(m - n^2)(m + n^2) = 289$$

The factor pairs of 289 are (1, 289) and (17, 17). The valid factor pair is (1, 289), leading to:

$$m - n^2 = 1$$

$$m + n^2 = 289$$

Adding and subtracting these equations, we find:

$$2m = 290 \implies m = 145$$

$$2n^2 = 288 \implies n^2 = 144 \implies n = 12$$

Thus, the unique positive integer (n) is $\boxed{12}$.

Listing 6 | An example SFT trajectory from reasoning data related to code.

```
## Question
In this kata, your job is to create a class Dictionary which you can add words to and
their entries. Example:
'''python
>>> d = Dictionary()

>>> d.newentry('Apple', 'A fruit that grows on trees')

>>> print(d.look('Apple'))
A fruit that grows on trees

>>> print(d.look('Banana'))
Can't find entry for Banana
'''

Good luck and happy coding!

Here is the setup code:
'''python
class Dictionary():
    def __init__(self):
        # Your code
        pass

    def newentry(self, word, definition):
        # Your code
        pass

    def look(self, key):
        # your code
        pass
'''

## Response
<think>
Okay, I need to create a Python class called Dictionary. The class should allow adding
```

words with their definitions and looking them up. Hmm, let's think about the setup.

So, when a Dictionary object is created, like `d = Dictionary()`, I probably need some kind of data structure to store the words and their entries. A dictionary seems perfect for this because it can hold key-value pairs, where the key is the word and the value is the definition.

So in the `__init__` method, I should initialize an empty dictionary. Like `self.entries = {}`.

Then, there's the `newentry` method. This method takes two arguments: word and definition. So in `newentry`, I need to add the word as the key and the definition as the value to the `self.entries` dictionary.

Now, the `look` method is called with a key. It needs to return the definition if the key exists, else return a string saying it can't find the entry. Like, for example, looking up 'Apple' returns the definition, but looking up 'Banana' returns 'Can't find entry for Banana'.

Wait, how does it handle case sensitivity? I guess we're assuming the keys are case-sensitive as per the string passed. So 'Apple' is different from 'apple'.

So the steps for the `look` function are: check if the key is in `self.entries`. If yes, return the value. If no, return the specific string.

Putting it all together:

For the `__init__`:

```
def __init__(self):
    self.entries = {}
```

For `newentry`:

```
def newentry(self, word, definition):
    self.entries[word] = definition
```

For `look`:

```
def look(self, key):
    if key in self.entries:
        return self.entries[key]
    else:
        return f"Can't find entry for {key}"
```

Wait, but in the example, when `print(d.look('Banana'))`, it prints the message without quotes. So the method returns the string which when printed gives that output.

So, yeah, that's correct.

Let me test this logic.

Testing with the example:

```
d = Dictionary()
d.newentry('Apple', 'A fruit...') --> adds to entries.
d.look('Apple') returns the string, so print shows it correctly.
d.look('Banana') returns the can't find message.
```

That should work.

I think that's the basic plan. So the setup code is given, I have to fill in the methods. So in code, it's as I wrote.

Wait, any edge cases? Like, what if word is an empty string? Or any other type? The problem doesn't specify handling those, so I think the code should work as intended.

Another thing: in the look method, when we do 'if key in self.entries', does it check for the presence as a key? Yes, so that's correct.

So, to implement:

Start by adding the `__init__`, `newentry`, and `look` methods as described.
</think>

To solve this problem, we need to create a Python class called `Dictionary` that allows adding words with their definitions and looking them up. The class should return the definition if the word exists and a specific message if it does not.

Approach

1. **Data Structure**: Use a dictionary to store the words as keys and their definitions as values. This allows efficient lookups and additions.
2. **Initialization**: In the `__init__` method, initialize an empty dictionary to store the word definitions.
3. **Adding Entries**: The `newentry` method takes a word and its definition, then adds this pair to the dictionary.
4. **Looking Up Entries**: The `look` method checks if the word exists in the dictionary. If it does, it returns the definition; otherwise, it returns a message indicating the word was not found.

Solution Code

```
'''python
class Dictionary():
    def __init__(self):
        self.entries = {}

    def newentry(self, word, definition):
        self.entries[word] = definition

    def look(self, key):
        if key in self.entries:
            return self.entries[key]
        else:
            return f"Can't find entry for {key}"
'''
```

Explanation

1. **Initialization**: The `__init__` method sets up an empty dictionary `self.entries` to store the words and their definitions.
2. **Adding Entries**: The `newentry` method takes two parameters, `word` and `definition`, and adds them to `self.entries` where `word` is the key and `definition` is the value.
3. **Looking Up Entries**: The `look` method checks if the given key exists in `self.entries`. If the key is found, its corresponding definition is returned. If not, a message is returned indicating the absence of the entry.

This approach ensures efficient storage and retrieval of word definitions using a dictionary data structure, providing an optimal solution for the problem.

Listing 7 | An example SFT trajectory from non-reasoning data related to writing.

Question

write a short poem ending in "was tentacles all along"