6.6 The Full Python Memory Model: Function Calls

So far in this chapter, we have talked only about variables defined within the Python console. In 2.3 Local Variables and Function Scope, we saw how to represent function scope in the value-based memory model using separate “tables of values” for each function call. In this section, we’ll see how to represent function scope in the full Python memory model so that we can capture exactly how function scope works and impacts the variables we use throughout the lifetime of our programs.

Stack frames

Suppose we define the following function, and then call it in the Python console:

```python
def repeat(n: int, s: str) -> str:
    message = s * n
    return message


# In the Python console
>>> count = 3
>>> word = 'abc'
>>> result = repeat(count, word)
```

Consider what the state of memory is when `repeat(count, word)` is called, *immediately before* the `return message` statement executes. Let’s first recall how we would draw the *value-based* memory model for this point:

`__main__`

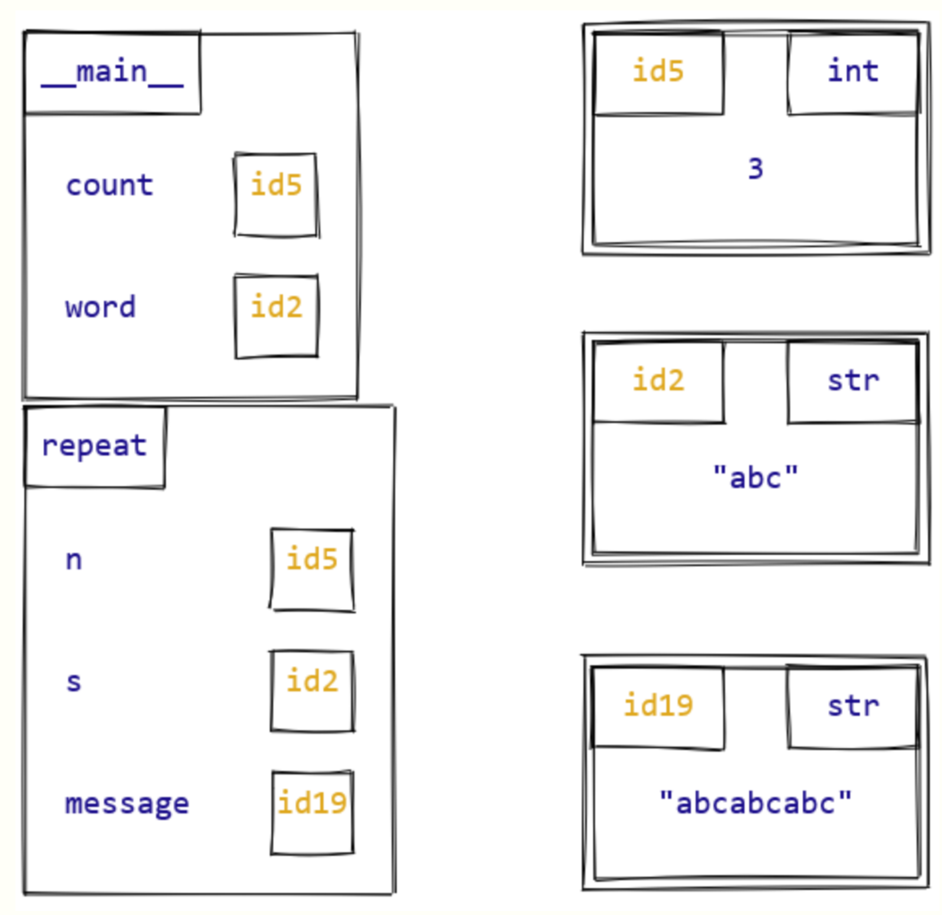
Variable	*Value*
`count`	`3`
`word`	`'abc'`

`repeat`

Variable	*Value*
`n`	`3`
`s`	`'abc'`
`message`	`'abcabcabc'`

This memory model shows two tables, showing the variables defined in the Python console (`count`, `word`), and the variables local to the function `repeat` (`n`, `s`, and `message`).

Here is how we would translate this into a full Python memory model diagram:



As with the diagrams we saw in the previous sections of this chapter, our variables are on the left side of the diagram, and the objects on the right. The variables are separated into two separate boxes, one for the Python console and one for the function call for `repeat`. All variables, regardless of which box they’re in, store only ids that refer to objects on the right-hand side. Notice that `count` and `n` are aliases, as are `word` and `s`.

Now that we have this full diagram, we’ll introduce a more formal piece of terminology. Each “box” on the left-hand side of our diagram represents a **stack frame** (or just **frame** for short), which is a special data type used by the Python interpreter to keep track of the functions that have been called in a program, and the variables defined within each function. We call the collection of stack frames the **function call stack**.

Every time we call a function, the Python interpreter does the following:

1. Creates a new stack frame and add it to the call stack.
2. Evaluates the arguments in the function call, yielding the ids of objects (one per argument). Each of these ids is assigned to the corresponding parameter, as an entry in the new stack frame.
3. Executes the body of the function.
4. When a return statement is executed in the function body, the id of the returned object is saved and the stack frame for the function call is removed from the call stack.

Argument passing and aliasing

What we often call “parameter passing” is a special form of variable assignment in the Python interpreter. In the example above, when we called `repeat(count, word)`, it is as if we wrote

```python
n = count
s = word
```

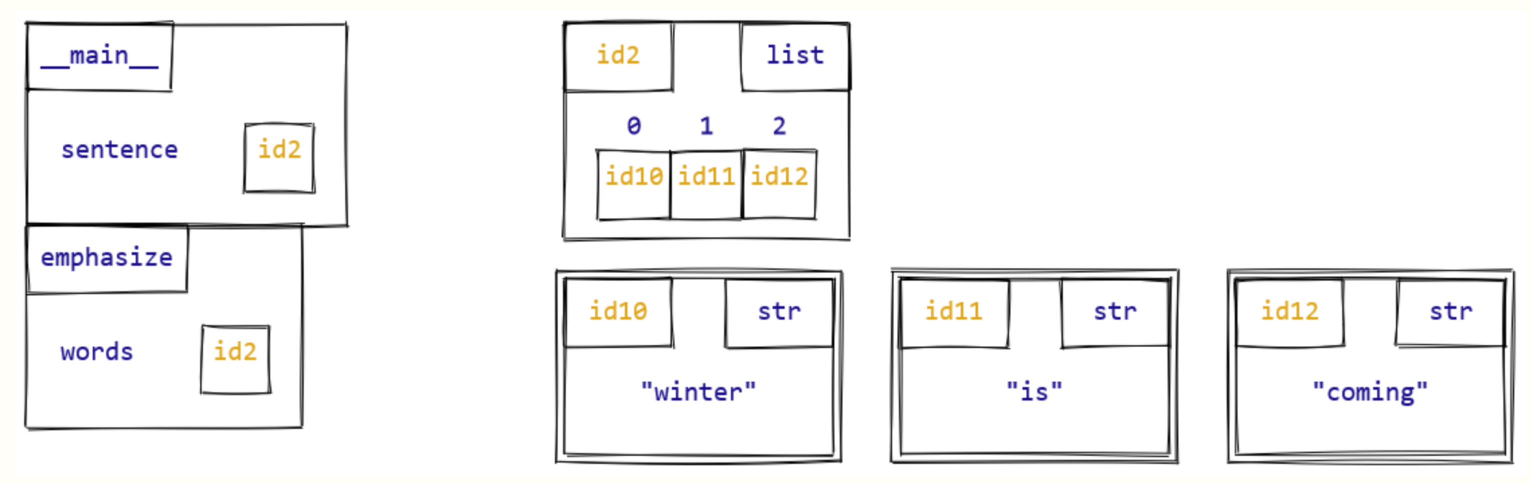
before executing the body of the function.

This aliasing is what allows us to define functions that mutate their argument values, and have that effect persist after the function ends. Here is an example:

```python
def emphasize(words: list[str]) -> None:
    """Add emphasis to the end of a list of words."""
    new_words = ['believe', 'me!']
    list.extend(words, new_words)


# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize(sentence)
>>> sentence
['winter', 'is', 'coming', 'believe', 'me!']
```

When `emphasize(sentence)` is called in the Python console, this is the state of memory:



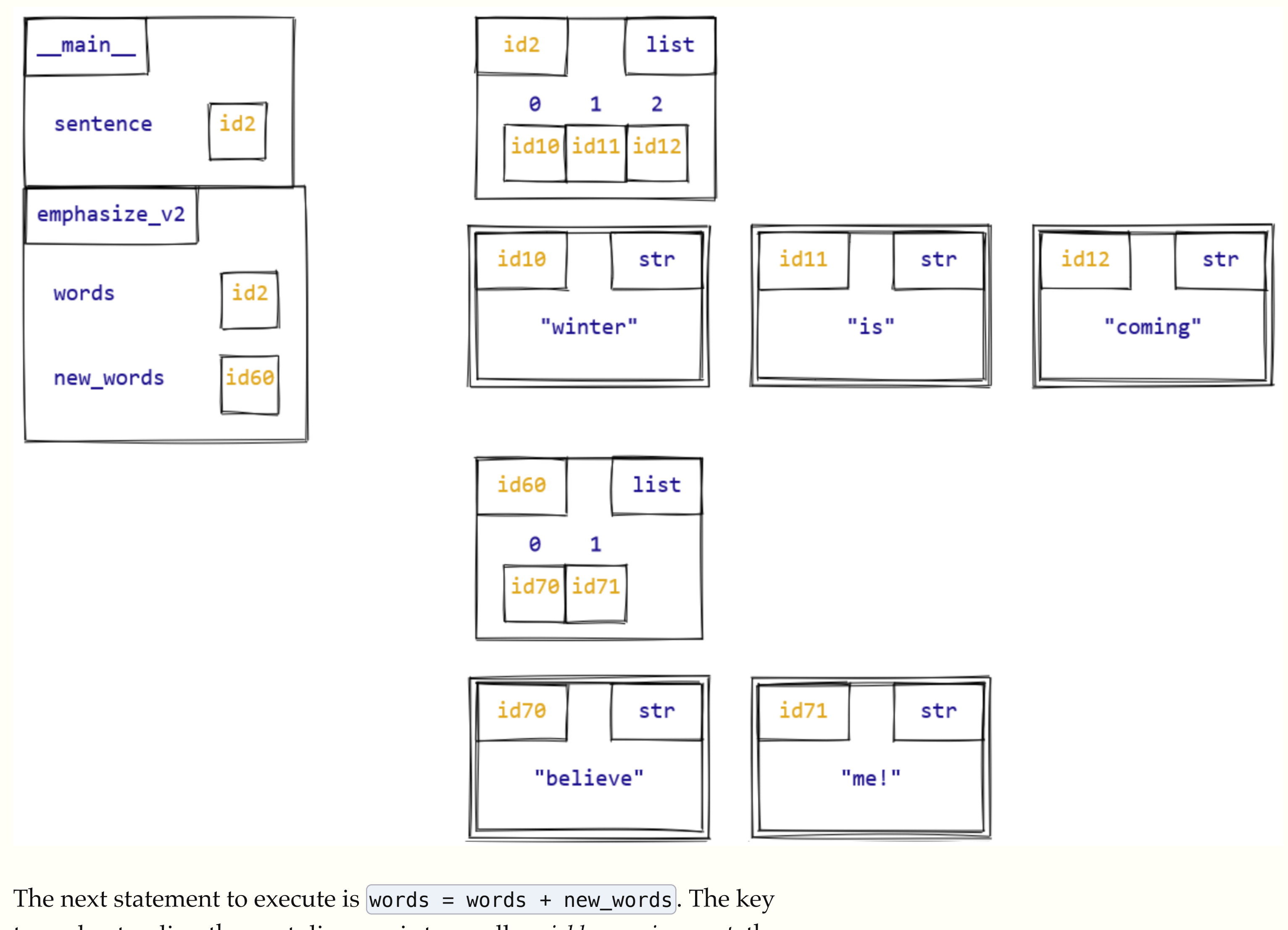
In this case, `words` and `sentence` are aliases, and so mutating `words` within the function causes a change to occur in `__main__` as well.

On the other hand, consider what happens with this version of the function:

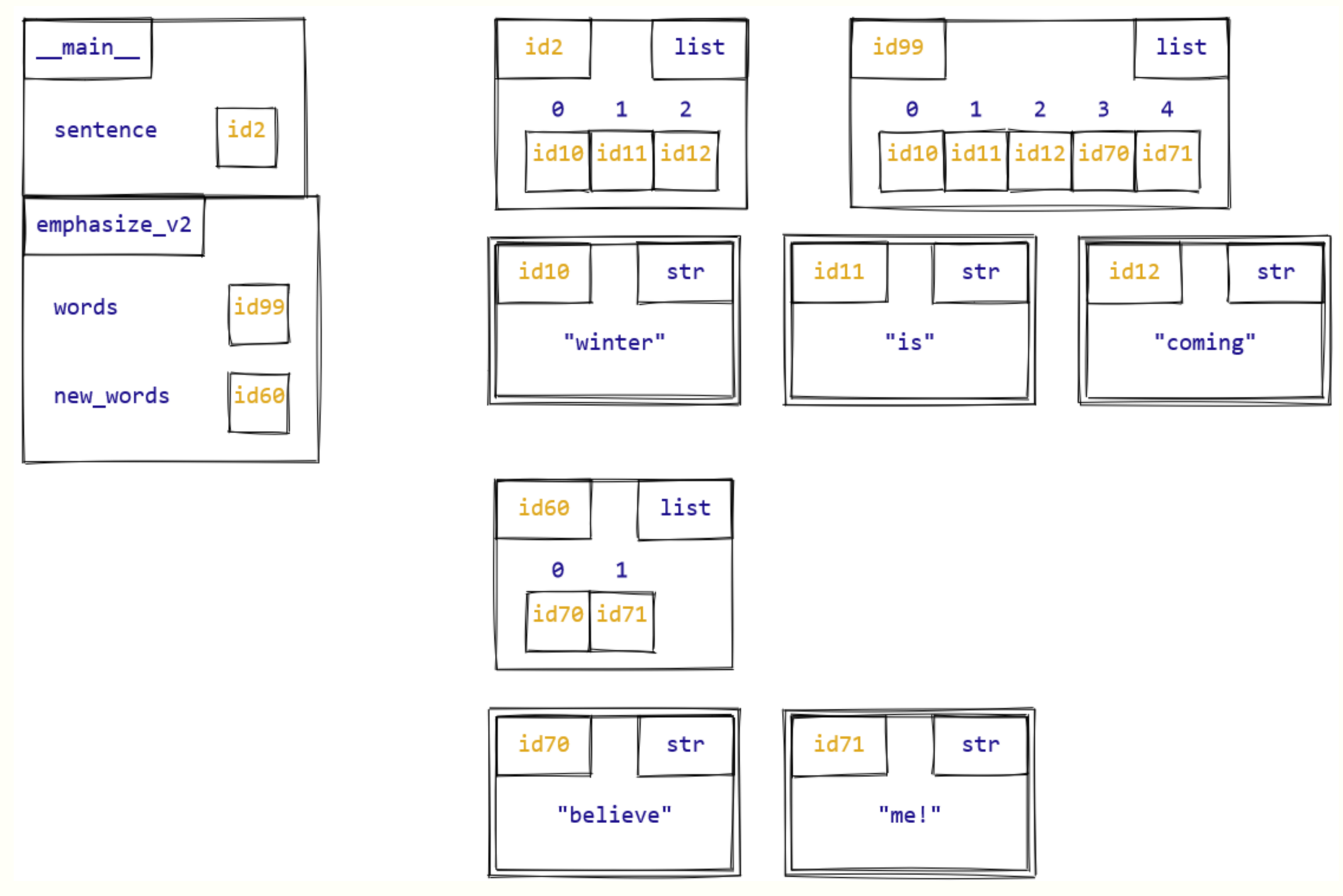
```python
def emphasize_v2(words: list[str]) -> None:
    """Add emphasis to the end of a list of words."""
    new_words = ['believe', 'me!']
    words = words + new_words


# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize_v2(sentence)
>>> sentence
['winter', 'is', 'coming']
```

After we call `emphasize_v2` in the Python console, the value of `sentence` is unchanged! To understand why, let’s look at two memory model diagrams. The first shows the state of memory immediately after `new_words = ['believe', 'me!']` is executed:



The next statement to execute is `words = words + new_words`. The key to understanding the next diagram is to recall *variable reassignment*: the right-hand side (`words + new_words`) is evaluated, and then the resulting object id is assigned to `words`. *List concatenation with* `+` *creates a new list object*.



Notice that in this diagram, `words` and `sentence` are no longer aliases! Instead, `words` has been assigned to a new list object, but `sentence` has remained unchanged.[1] This illustrates the importance of keeping variable reassignment and object mutation as distinct concepts. Even though the bodies of `emphasize` and `emphasize_v2` look very similar, the end result is very different: `emphasize` mutates its argument object, while `emphasize_v2` actually leaves it unchanged!

[1] Remember the rule of variable reassignment: an assignment statement `<name> = ...` only changes what object the variable `<name>` refers to, but never changes any other variables.