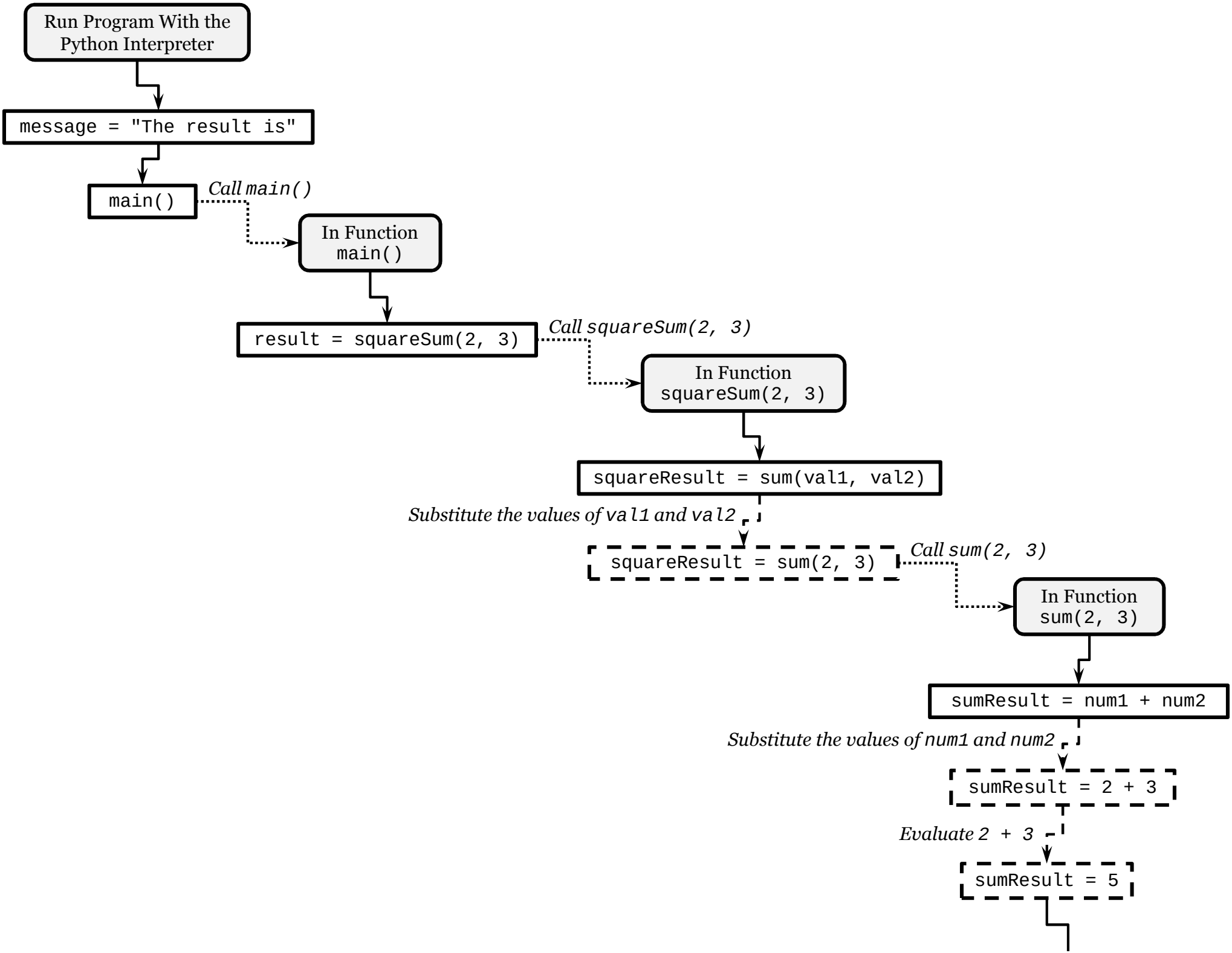
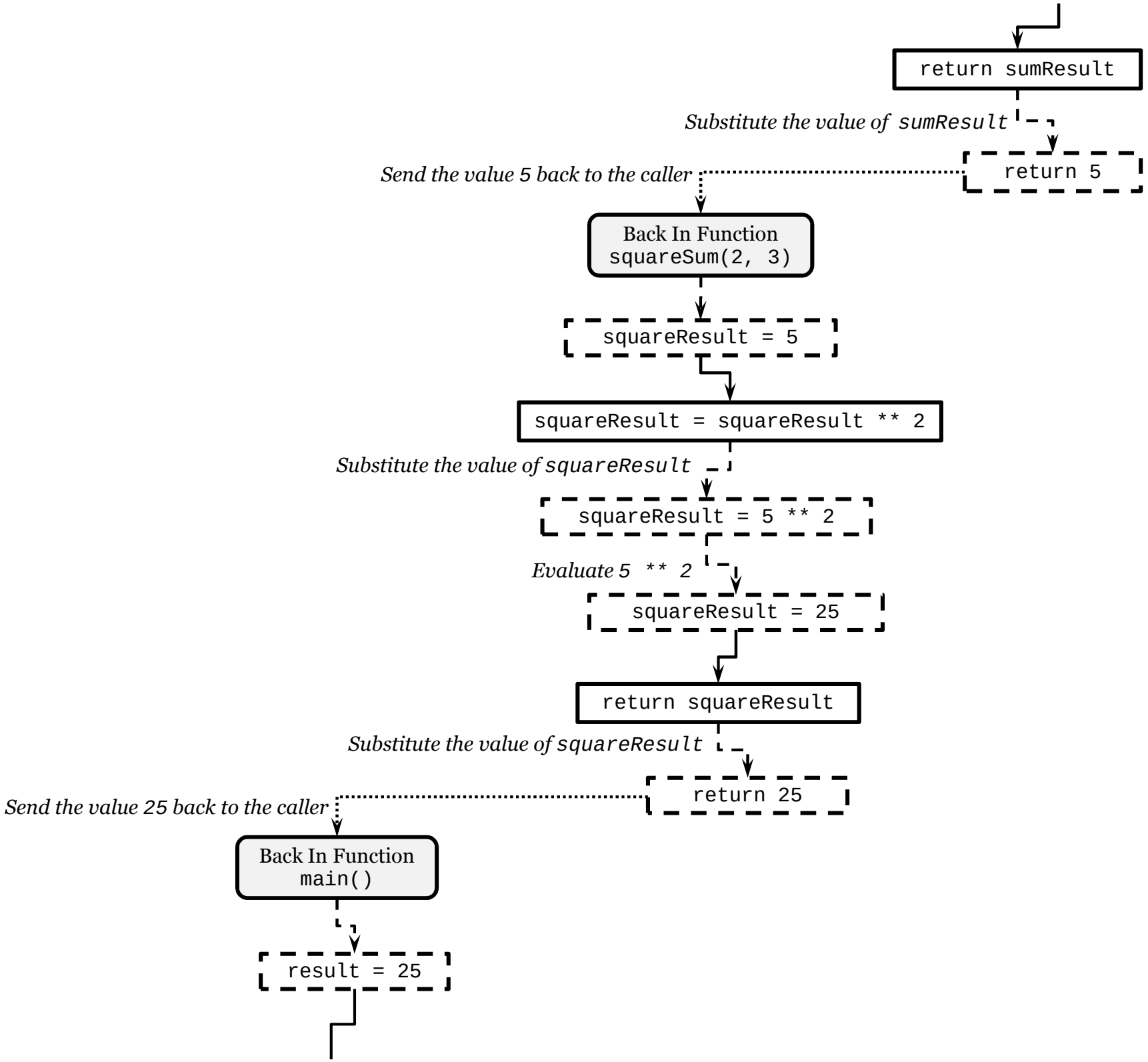
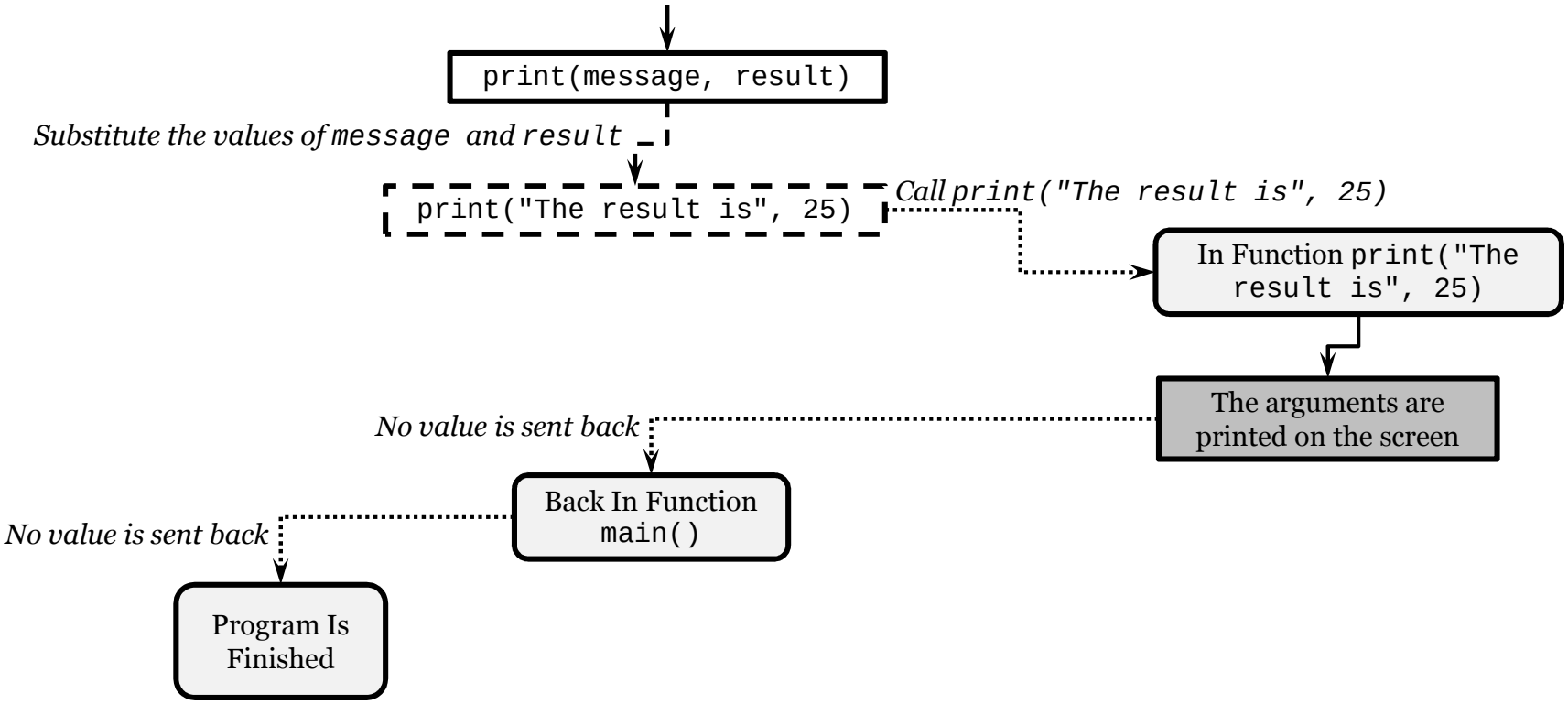
Consider the following Python program:

```
message = "The result is"

def sum(num1, num2):
    sumResult = num1 + num2
    return sumResult

def squareSum(val1, val2):
    squareResult = sum(val1, val2)
    squareResult = squareResult ** 2
    return squareResult

def main():
    result = squareSum(2, 3)
    print(message, result)

main()
```

When the Python interpreter runs this program, what is the sequence of commands executed?


Run Program With the Python Interpreter
message = "The result is"
main()
Call main()
In Function main()
result = squareSum(2, 3)
Call squareSum(2, 3)
In Function squareSum(2, 3)
squareResult = sum(val1, val2)
Substitute the values of val1 and val2
squareResult = sum(2, 3)
Call sum(2, 3)
In Function sum(2, 3)
sumResult = num1 + num2
Substitute the values of num1 and num2
sumResult = 2 + 3
Evaluate 2 + 3
sumResult = 5

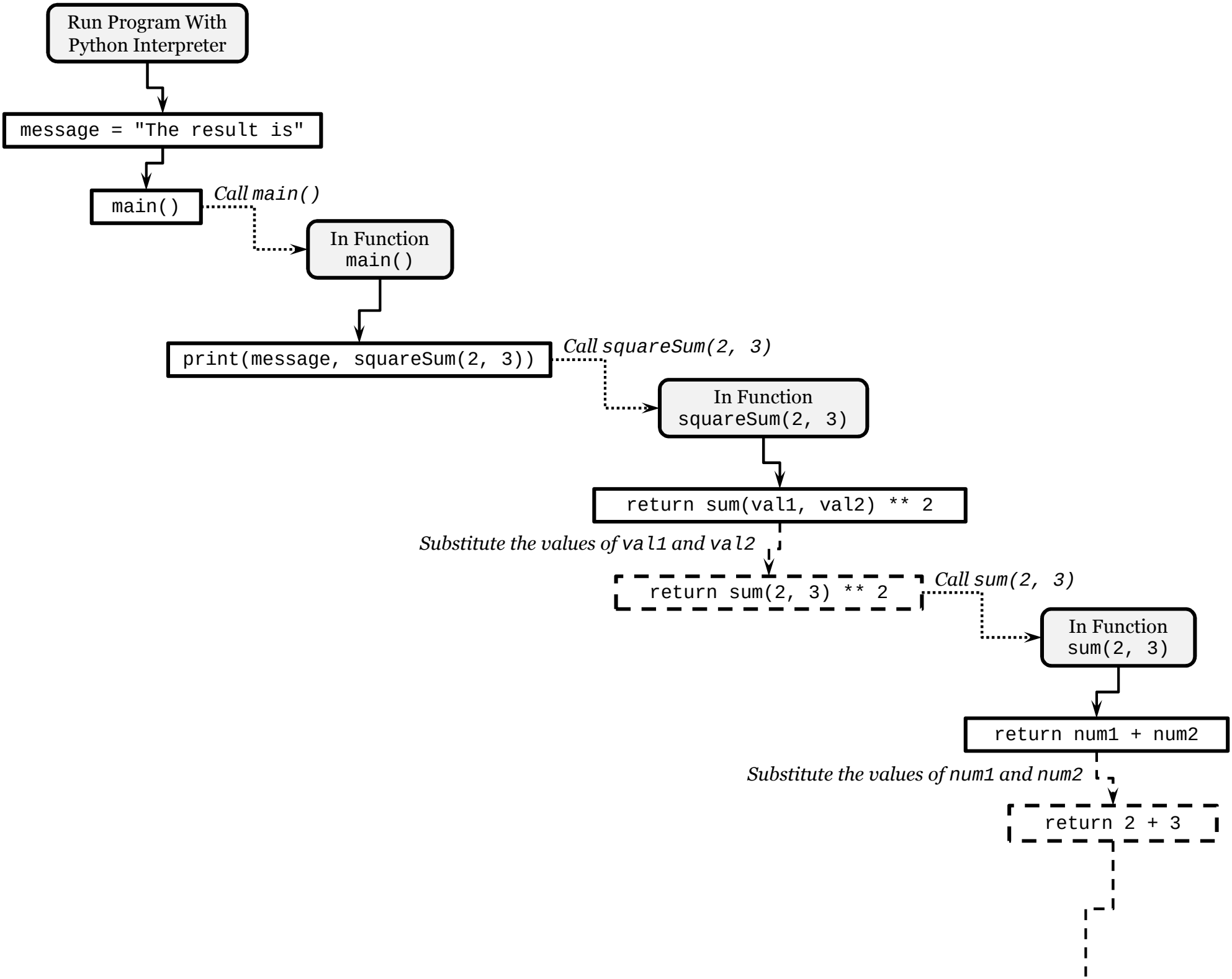

return sumResult
Substitute the value of sumResult
return 5
Send the value 5 back to the caller
Back In Function
squareSum(2, 3)
squareResult = 5
squareResult = squareResult ** 2
Substitute the value of squareResult
squareResult = 5 ** 2
Evaluate 5 ** 2
squareResult = 25
return squareResult
Substitute the value of squareResult
return 25
Send the value 25 back to the caller
Back In Function
main()
result = 25

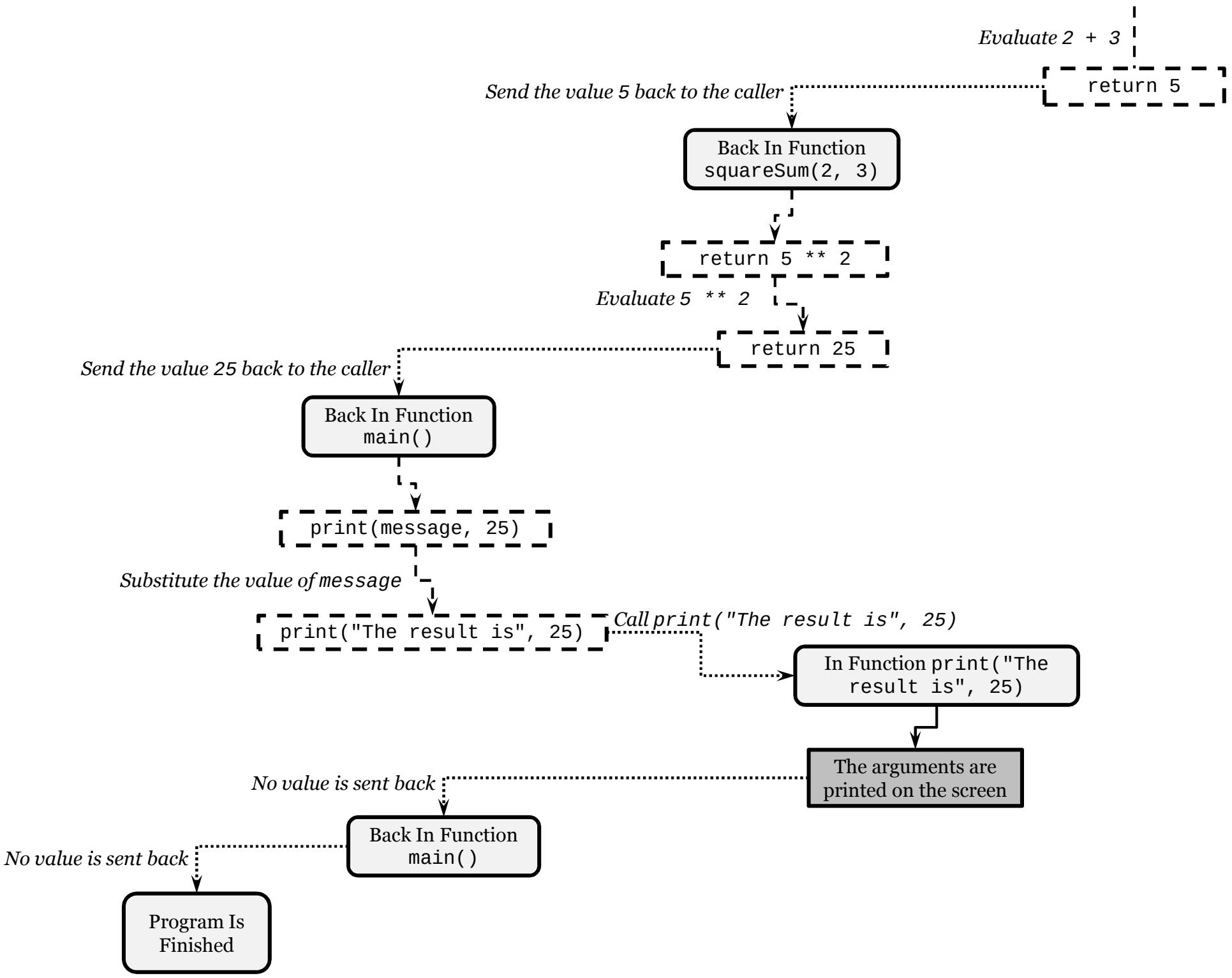


Now consider the following shorter version of the previous program:

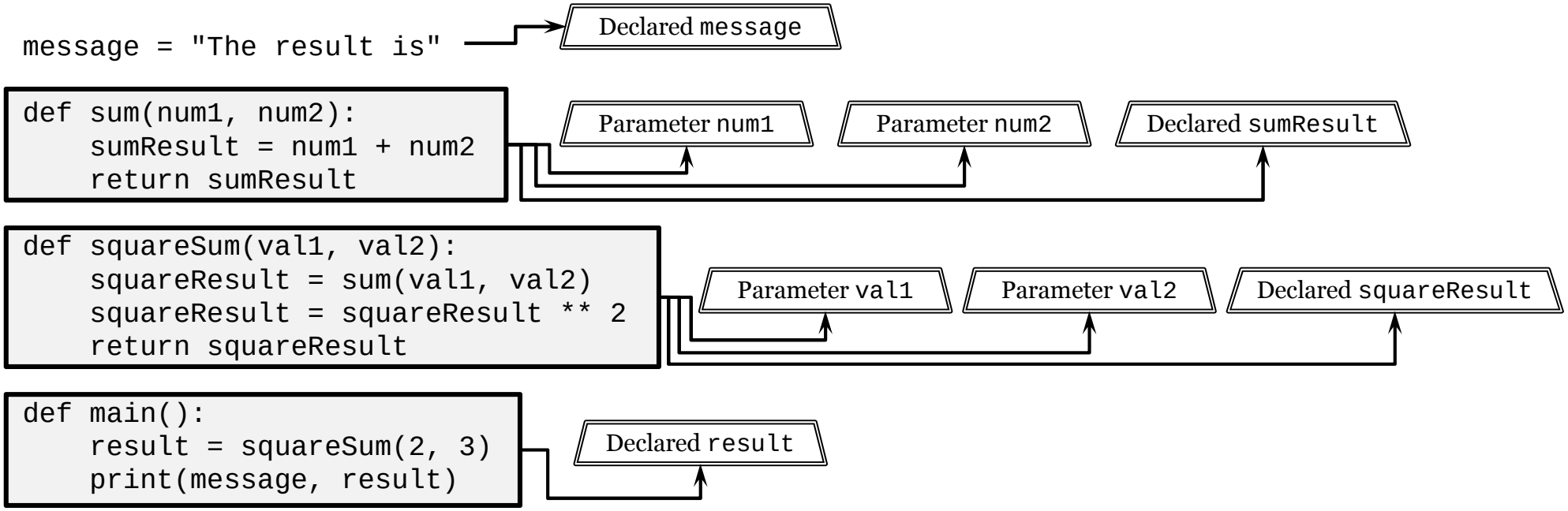
```
message = "The result is"

def sum(num1, num2):
    return num1 + num2

def squareSum(val1, val2):
    return sum(val1, val2) ** 2

def main():
    print(message, squareSum(2, 3))

main()
```

When the Python interpreter runs this program, what is the sequence of commands executed?


Run Program With Python Interpreter
message = "The result is"
main()
Call main()
In Function main()
print(message, squareSum(2, 3))
Call squareSum(2, 3)
In Function squareSum(2, 3)
return sum(val1, val2) ** 2
Substitute the values of val1 and val2
return sum(2, 3) ** 2
Call sum(2, 3)
In Function sum(2, 3)
return num1 + num2
Substitute the values of num1 and num2
return 2 + 3


Evaluate 2 + 3
return 5
Send the value 5 back to the caller
Back In Function squareSum(2, 3)
return 5 ** 2
Evaluate 5 ** 2
return 25
Send the value 25 back to the caller
Back In Function main()
print(message, 25)
Substitute the value of message
print("The result is", 25)
Call print("The result is", 25)
In Function print("The result is", 25)
The arguments are printed on the screen
No value is sent back
Back In Function main()
No value is sent back
Program Is Finished

What are the variables in this Python program and where are they defined?



The function `sum(...)` has three *local* variables that only exist in `sum(...)`: parameters `num1` and `num2`, and declared variable `sumResult`. These variables can **not** be used in any function other than `sum(...)`; their *scope* is their function, represented here by the grey rectangle around `sum(...)`.

Think of these variables as having a "local scope rule" implicitly attached to their names: $\texttt{num1}_{\texttt{only exists in sum(...)}}$, $\texttt{num2}_{\texttt{only exists in sum(...)}}$ and $\texttt{sumResult}_{\texttt{only exists in sum(...)}}$. These attachments aren't actually used in Python; they're imaginary devices used here to illustrate a variable's scope.

The function `squareSum(...)` has its own local variables: parameters `val1` and `val2`, as well declared variable `squareResult`. The function `main()` has one local declared variable: `result`.

Now consider introducing the following bug in `main()`:

```
def main():
    result = squareSum(2, 3)
    print("The values are", num1, "and", num2)
    print(message, result)
```

The bug may not be obvious here. Try attaching the "local scope rule" to the variable names for `result`, `num1` and `num2`:

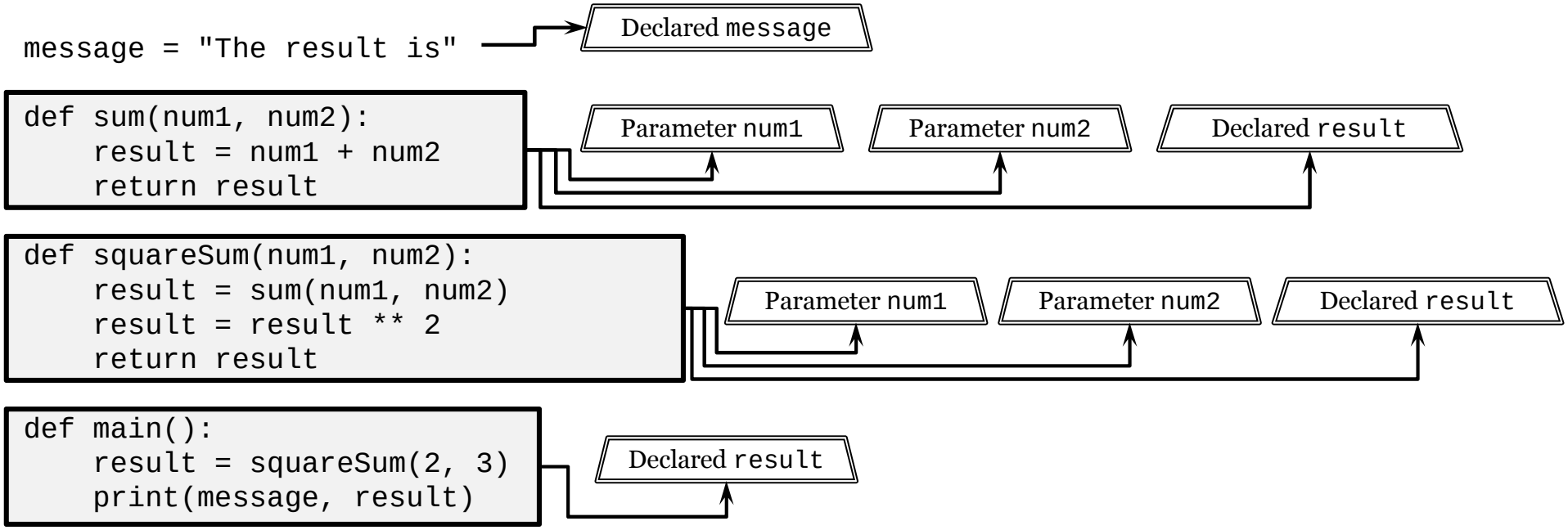
```
def main():
    result only exists in main() = squareSum(2, 3)
    print("The values are", num1 only exists in sum(...), "and", num2 only exists in sum(...))
    print(message, result only exists in main())
```

The bug becomes more clear now: the `main()` function is attempting to use variables that only exist in the function `sum(...)`. But why isn't this a problem with variable `message`? It's being used in `main()` without issue.

The variable `message` is a *global* variable; it is not declared inside any function. This means that `message` can be used everywhere and its *scope* is the entire program. In this case it's only used in `main()`, but it could also be used in `sum(...)` or `squareSum(...)` or both. Think of this variable as having a "global scope rule" implicitly attached to its name: `message` `exists everywhere`. If you attached all the "scope rules" to the variable names in `main()` it would look like this:

```
def main():
    result only exists in main() = squareSum(2, 3)
    print(message exists everywhere, result only exists in main())
```

What if the variable names are changed?



Even though many of the variables have the same names, the program operates in exactly the same way. There are still three independent sets of local variables, listed below with their "scope rules" attached to distinguish between them:

- `num1`$_{\texttt{only exists in sum(...)}}$, `num2`$_{\texttt{only exists in sum(...)}}$ and `result`$_{\texttt{only exists in sum(...)}}$ local to function `sum(...)`
- `num1`$_{\texttt{only exists in squareSum(...)}}$, `num2`$_{\texttt{only exists in squareSum(...)}}$ and `result`$_{\texttt{only exists in squareSum(...)}}$ local to function `squareSum(...)`
- `result`$_{\texttt{only exists in main()}}$ local to function `main()`

There are two separate variables in this program with name `num1`, two separate variables with name `num2`, and three separate variables with name `result`. A function can only use its own local variables and any global variables; it can not use variables from other functions, even if they have the same names.