Understanding and Resolving the “SyntaxError: positional argument follows keyword argument” in Python

Authored by

Mohammed looti

November 2, 2025

The Python programming language is known for its readability and strict syntax rules. When writing complex function calls, developers occasionally encounter a specific compilation issue related to argument parsing. One of the most frequently misunderstood runtime errors is the following:

SyntaxError: positional argument follows keyword argument

This SyntaxError is not arbitrary; it is a direct consequence of how the Python interpreter must evaluate the parameters passed to a function. It occurs exclusively when a developer attempts to place a non-keyworded argument--a positional argument--immediately after an argument that has already been explicitly named using its parameter name--a keyword argument. Understanding the fundamental distinction between these two argument types is essential for mastering function definition and invocation in Python.

Understanding the Positional Argument Follows Keyword Argument Error

The core philosophy behind Python's argument parsing is to ensure clarity and prevent ambiguity during the function call. When a function is invoked, the interpreter needs a predictable method to map the values provided by the caller to the specific parameters defined in the function signature. Positional arguments rely entirely on order, while keyword arguments rely on explicit naming. When these two methods are mixed incorrectly, the interpreter cannot reliably determine the intended assignment for the remaining arguments, leading to the reported SyntaxError.

This error is specifically classified as a SyntaxError because the structure of the call itself violates the language rules, meaning the code cannot even be compiled or executed successfully. The parser stops immediately upon encountering this sequence because it represents an impossible structure to resolve based on Python's strict argument order policy. To resolve this, one must strictly adhere to the rule that all positional arguments must precede all keyword arguments within a single function call.

We can simplify the rule into a memorable mnemonic: once you start naming arguments (using keywords), you cannot go back to simply listing them (using position). This rule ensures that if the function has many optional parameters, the developer can name only the ones they need to change, but they must first fulfill any required positional parameters in the correct order. Any deviation from this sequence results in the interpreter being unable to map the remaining positional values correctly.

Defining Function Arguments: Positional vs. Keyword

To properly diagnose and fix this error, we must clearly differentiate the two primary ways arguments can be passed to a function in Python: by position and by keyword. This distinction is

fundamental to how function signatures are executed and how the interpreter manages scoping and variable assignment during a function call.

A positional argument is a value supplied to a function without explicitly referencing the parameter name defined in the function signature. The assignment of the value relies entirely on its sequence or position relative to other arguments. The first value supplied maps to the first parameter, the second value to the second parameter, and so forth.

Example of Positional Arguments: In the call **my_function(2, 2)**, the value 2 is assigned to the first parameter defined in `my_function`, and the second value 2 is assigned to the second parameter, regardless of what those parameters are named (e.g., `a` and `b`).

Conversely, a keyword argument is a value supplied to a function where the parameter name is explicitly stated using the syntax `name=value`. Because the assignment is explicit, the order in which keyword arguments are listed does not matter, provided they appear after all positional arguments.

Example of Keyword Arguments: In the call **my_function(a=2, b=2)**, the interpreter knows exactly that the first 2 goes to parameter `a` and the second 2 goes to parameter `b`. This explicit naming enhances code clarity, especially when dealing with functions that have many parameters with default values.

The error arises precisely when you attempt to mix these two styles in the sequence of **Keyword, Positional**. For example, the call **my_function(a=2, 2)** is invalid syntax because the positional value `2` follows the keyword assignment `a=2`. This sequence creates an insurmountable ambiguity for the Python parser.

Why Argument Order Matters in Python

The mandatory ordering rule--positional arguments first, then keyword arguments--is critical for the efficient operation of the Python Virtual Machine (PVM). When the PVM processes a function signature, it uses a fixed sequence to assign values.

First, the interpreter scans the arguments and assigns all positional values based on their sequential order to the corresponding parameters defined in the function signature.

Second, the interpreter processes the keyword arguments, assigning them to their explicitly named parameters, regardless of their position relative to other keyword arguments.

Finally, the interpreter checks that all required parameters have received exactly one value, either through position or keyword, and that no unexpected or duplicate keyword arguments were supplied.

If a positional argument is encountered after a keyword argument, the parsing sequence breaks down. Once the parser sees an argument assigned via keyword (e.g., `a=4`), it concludes the positional parsing phase. The following positional value (e.g., `10`) is then left without a defined parameter to map to. The interpreter cannot simply assume it should map to the next available parameter (like `b`) because the positional phase has already terminated, resulting in the immediate raising of the SyntaxError.

Practical Demonstration of Valid Function Calls

To illustrate the correct usage of arguments, let us define a simple function in Python that performs a calculation involving three parameters:

```
def do_stuff(a, b, c):
return a * b / c
```

The following examples demonstrate the three valid methods for invoking this function, all of which respect the argument parsing rules enforced by Python.

Valid Way #1: Using All Positional Arguments

In this scenario, we rely entirely on the sequence defined in the function signature (a, b, c). The values 4, 10, and 5 are mapped sequentially.

```
do_stuff(4, 10, 5)
```

8.0

No error is raised here because Python unambiguously assigns `a=4`, `b=10`, and `c=5` based purely on position. This is the simplest and most common form of a function call when the order is clear.

Valid Way #2: Using All Keyword Arguments

Here, we explicitly name every parameter. Since we are using keyword arguments exclusively, the order could technically be shuffled (e.g., `c=5, a=4, b=10`) without causing an error, though maintaining a logical order is generally good practice.

```
do_stuff(a=4, b=10, c=5)
```

8.0

Again, no error is thrown because Python knows exactly which parameter receives which value

due to the explicit naming convention inherent in keyword arguments. This method is often preferred for functions with complex signatures to enhance clarity.

Valid Way #3: Positional Arguments Before Keyword Arguments

This is the only valid way to mix argument types. We fulfill the initial parameters using positional assignment and then use keyword assignment for the remaining parameters.

do_stuff(4, b=10, c=5)

8.0

In this successful function call, Python first assigns the value `4` to the first parameter, `a` (positional phase). It then proceeds to the keyword phase, assigning `b=10` and `c=5`. Since the positional argument (4) came before the keyword arguments (b=10, c=5), the parsing rules are satisfied, and the function executes correctly.

Analyzing the Invalid Syntax and the Root Cause

The issue arises when the parsing sequence demonstrated in Valid Way #3 is reversed. When the interpreter encounters a keyword assignment, it assumes the positional phase is complete. If a plain, unnamed value follows, the interpreter cannot backtrack or re-evaluate its assignment strategy, leading to the abrupt termination of the parsing process and the subsequent error message.

Invalid Way: Positional Arguments After Keyword Arguments

Consider the following attempt to invoke the `do_stuff` function:

do_stuff(a=4, 10, 5)

SyntaxError: positional argument follows keyword argument

The error is thrown immediately because the structure is fundamentally flawed. When the interpreter sees `a=4`, it concludes that parameter `a` is satisfied and the positional argument phase is over. It then expects only further keyword assignments. However, it encounters `10` and `5`, which are supplied as positional arguments.

The interpreter is faced with an unsolvable ambiguity: should `10` be assigned to `b` and `5` to `c`, or is the user trying to assign `10` to `c` and `5` to some other parameter? Because the developer has already provided an explicit keyword assignment (`a=4`), the positional context has been lost for

subsequent arguments. To avoid this ambiguity and maintain a predictable parsing mechanism, Python enforces the strict ordering rule, making the above sequence illegal syntax.

Strategies for Resolving the SyntaxError

Resolving the `positional argument follows keyword argument` SyntaxError is straightforward. The solution involves restructuring the function call to ensure all positional arguments appear first, or, alternatively, converting all arguments into keyword arguments.

Convert all trailing positional arguments to keyword arguments: This is generally the cleanest and most recommended fix, as it clarifies the intent of the programmer and improves code readability. Instead of relying on position for `10` and `5`, we explicitly name them:

 # Corrected: All arguments are now keyworded
do_stuff(a=4, b=10, c=5)

Convert the leading keyword argument to a positional argument: If the leading argument (in this case, `a=4`) is one of the early parameters in the function signature, you can simply drop the keyword assignment and rely on position for that value, ensuring it comes before any remaining keyword arguments.

 # Corrected: 4 is positional; 10 and 5 are now positional
do_stuff(4, 10, 5)

Ensure all positional arguments precede all keyword arguments: If you intend to use a mix of both, make sure the values assigned strictly by order (position) are listed first, followed by the values assigned explicitly by name (keyword). For example, if `a` and `b` are positional, and `c` is keyworded:

 # Corrected: 4 and 10 are positional; c=5 is keyworded
do_stuff(4, 10, c=5)

By implementing any of these strategies, developers can satisfy the Python interpreter's requirement for predictable argument resolution, thereby eliminating the SyntaxError and ensuring successful compilation and execution of the function.

Additional Resources

The following tutorials explain how to fix other common errors encountered during Python development, helping developers maintain clean and valid code structures: