Scoping

One important way that members of the Django-type languages differ is in how they handle scoping. To illustrate this issue, consider the case of the *for* tag. In Jinja, the following template will evaluate to “1000”. In Twig, however, the template evaluates to “6050”.

{%- set value = 1000 -%}

{%- for item in range(0,100) -%}

{%- set value = value + item -%}

{%- endfor -%}

{{- value -}}

This happens because Jinja and Twig apply different scoping rules. In Jinja, values that are updated inside the for loop carry forward from iteration to iteration, but fall out of scope when the loop terminates. In Twig, values set inside the loop do not fall out of scope and persist after the for loop ends.

Blended solves this problem by specifying specific scoping rules and enforcing them through the Blended-Lint program. These rules are constructed to ensure that a valid Blended template will execute in exactly the same way regardless of what Blended-compatible runtime is used, and to ensure that any template code that executes differently in the various Blended-compliant runtimes is rejected by Blended-Lint. In Blended, the above example template would be invalid.[[1]](#footnote-0)

We facilitate this analysis by defining a series of nested scopes. These scoping concepts serve the purpose of making it easier to reason about Blended’s scoping rules--but they also form the basis of Blended-Lint’s semantic analysis. The core ideas are these:

* Names defined inside parent or ancestor scopes are available for use within child or descendant scopes, but not vice-versa;
* Names defined inside parent or ancestor scopes cannot be overwritten or modified within child or descendant scopes--they are read-only;
* Names defined inside a scope are unavailable after the scope exits--they are local to that scope;
* Macros are different: the only external objects automatically accessible from inside a macro are built-in functions, user functions, and whatever other macros were defined inside the same scope. Inside a macro, you can also import macros from other files, but the only way that a macro can get access to outside variables is for them to be passed in as arguments.

For someone whose only objective is to write Blended-compatible templates for their website, knowing the above four points should be sufficient. As long as your template is accepted by Blended-Lint, it will be compatible. However, more detail is required to create a new Blended-compatible runtime—and occasionally, perhaps, to debug an error emitted by Blended-Lint. The remainder of this document attempts to exhaustively cover all points relevant to scoping in the Blended Language.

1. All Blended code is comprised of static text that is (optionally) interspersed by tags. Tags are distinguished from static text through the use of three pairs of delimiters: “{%” and “%}”; “{{“ and “}}”; and “{#”, and “#}”. Occasionally, these delimiters are augmented by whitespace removal indicators, in the form of the minus sign (i.e. “-”).
2. Inside of these delimiters, internal syntax varies from tag to tag. A tag that uses the delimiters “{%” and “%}” always begins its internal syntax with a name that identifies its purpose. This document refers to such tags as “named tags” or “named statement tags”; they are to be distinguished from “expression tags”, which use the “{{“ and “}}” delimiters, and “comment tags”, which use the “{#” and “#}” delimiters.
3. Some tags work together in pairs or in groups, which pairs or groups are treated as single units when rendered. Colloquially, these pairs or groups of tags might be referred to as “blocks”, but in this document the terms “block” and “blocks” are reserved for use in describing “block tags” and their role in template inheritance; therefore, it is incorrect to refer to any pair or group of tags as a “block” unless in the context of template inheritance (i.e. the term “if block” should not be used). Instead, we refer to these pairs or groups of tags as belonging to the same “statement.” “Statements” may consist of individual tags (for instance, the “set” statement), or they may be comprised of multiple tags (for instance, the “if” and “for” statements).
4. We use the terms “segment” and “code segment” to refer to any Blended code found between two tags of the same statement. A code segment acts as a partial template that is rendered according to the prescribed behavior of the statement to which it belongs.
5. Statements are the control structures of Blended. They operate on the context and its data, each statement contributing to the final rendered output of a template. When Blended code is parsed and compiled, the resultant template instance retains the ordered relationship of the static text to any statements contained within that text. If any named statement tag is immediately followed by a newline character, that character is removed from the template (with the exception of endverbatim tags).
6. Access to the data that statements operate upon is restricted by scope. Nine (9) types of scope are defined for Blended templates.
   * “Built-in” scope: built-in functions provided by Blended
   * “User-function” scope: custom functions provided by the application
   * “Render” scope: data provided by the application at the time that “render” is invoked on a template.
   * “Template” scope: names assigned in the template, outside any blocks
   * “Macro” scope: template code inside *macro* tags
   * “Block” scope: template code inside *block* tags
   * “Iterator” scope: iteration objects used by *for...else* block
   * “Loop” scope: template code inside *for* section of the *for...else* block
   * “Else” scope: template code inside the *else* section of a *for...else* block
7. Ten (12) data types exist in Blended templates: strings, numbers, booleans, lists (array type), dictionaries (object type), functions, macros, modules, templates, pair lists, timestamps (datetime type), safe strings, and escaped strings.
   * Strings, numbers, booleans, lists, and dictionaries (i.e. basic variable types) are added to a scope as literal values passed in as input to a tag, or by being part of the application-provided context. They can also be added to a scope as a transformation performed on other variable data via filters, or as the return values of functions, which may then be passed in as input to a tag.
   * Pair lists, timestamps, templates, safe strings, and escaped strings (i.e. derived variable types) are added to a scope through transformations performed on core variable types via filters, or as the return values of functions, which may then be passed in as input to a tag.
     + Pair lists may be introduced by applying the *items* filter to dictionary objects.
     + Timestamps may be introduced by applying the *datetime* filter to properly formatted lists or to properly formatted strings.
     + Templates may be introduced through the use of the *template* filter, which compiles a string into a template object.
     + Safe strings may be introduced by applying the *safe* filter, or as the output of any macro invocation.
     + Escaped strings may be introduced by applying the *escape* filter (alias *e*) to string objects.
   * Functions are provided by the “environment” in the form of built-ins or user-functions, and cannot be overridden.
   * Macros are added to a scope via the *macro* tag, or the *from...import* tag.
   * Modules are added to a scope through the use of the *import* tag.
8. Data is typically referenced and accessed by name. All data names occupy the same namespace. Any rules regarding names in general apply equally to variable names, function names, macro names and module names (i.e. to all data types). At any point in a file, the current namespace consists of the current scope (read-write), and all parent and ancestor scopes (read-only).
9. Tags, statements, segments and expressions are assessed top to bottom, left to right. Data names may be referenced or accessed by a tag or expression if and only if it has *previously* been defined within the current scope or within a parent or ancestor scope.
10. No data names used in any parent or ancestor scope may be modified or overridden within any child or descendant scope--they are read-only within the child or descendant scope. This means that names in use by the parent scope (or in any ancestor scope) may not be used as a target for *set*, as a macro name, as a name for a module imported using the *import* tag, or as an iteration item in the *for* tag. Only names declared within the current scope may be overwritten.
11. Built-in function names are defined within the outermost scope, the “built-in” scope. By implication, no functions, variables, macros or imported modules may share a name with any built-in functions. Built-in functions are defined as part of the language specification itself; a list can be found [here](about:blank) [TBD].
12. User-defined functions inhabit the second-level scope, the “user-function” scope, which is the child of the “built-in” scope. As follows from the rules above, no user-defined function may share the name of any built-in function, nor may any template-defined macro, variable or imported module share the name of any user-defined function. For purposes of Blended-Lint, user-defined functions are declared in a specifically-formatted JSON file that specifies each function’s name, argument data types, and possible return types.
13. “Application” scope is the child of the “user-function” scope, and is the parent of the template scope. As a result, it follows naturally that it is both immutable, and global within the template. Variables defined in the application scope are accessible inside all templates and tags, but no such variable may be modified or overwritten in any template by any tag.
14. Any variable name that is referenced or accessed within the body of a template, but which has not been declared within the body of the template, should be assumed to have been introduced within the application scope. However, a JSON schema document may be provided to Blended-Lint which declares all of the application variables for a given template file. In the case that such a schema file is provided, any variable name that is referenced or accessed within the body of a template which has neither been declared within the body of the template nor declared within the JSON schema document shall be deemed an error (“variable undefined”), causing the file to be declared invalid.
15. The “template” scope is the scope of all variables, macros and modules defined inside the root template, outside of all *macro* tags, *block* tags, and *for...else* statements.
16. A *macro* statement registers the name of its macro within the scope that contains the *macro* statement. The code segment within a *macro* statement is coincident with the “macro” scope for that macro. Any “macro” scope is the child of the template’s “user-function” scope, and as a result it will not have access to the template’s “application” scope, “template” scope, or any descendant scope. The code segment within a macro and its conicident scope is evaluated by Blended-Lint at the point that it is defined, \*not\* where it is used.
17. Passing variable names as arguments is the only way for a macro to have access to any portion of the caller’s scope. The values are passed by reference, so while the values themselves are immutable, what values the argument names refer to may change within the “macro” scope. These variables are part of the “macro” scope.
18. In order to give a macro access to any macros defined in other files, the *import* tag or the *from...import* tag must be used inside the macro’s body. However, macros defined within the same containing scope (sibling macros) are automatically imported into the each other’s body scope. When a macro is invoked, its “macro” scope will automatically be given access to these sibling macros, without any explicit import being required. This is the case in spite of the fact that no other (non-macro) data from the macro statement’s containing scope will be accessible within that macro’s scope.
19. Macros can make recursive calls to themselves, but circular calls between and among macros is prohibited.
20. No macro may share the name of any other object defined before or after its definition in the same scope, or in any parent scope. By implication, no macro may be imported using the *from...import* tag if the name of the macro conflicts with any other object defined before or after its definition in the same scope, or in any parent scope. In order to import a macro with a conflicting name, the *from...import* tag’s *as* keyword should be used to change the name of the macro.
21. The *macro* statement may not appear inside an *if...else* statement, nor inside a *for...else* statement. Defining a macro using a *macro* tag inside of another *macro* tag, however, is permitted. *Macro* tags may only be used directly within the “template” scope, directly within the scope of a *block* statement, or within a “macro” scope.
22. No *macro* statement--nor any other statement other than the initial *extends* tag--is permitted to exist outside of *block* statements in any child or descendant templates (i.e. templates that start with the *extends* tag). In such templates, macros must be defined inside *block* statements. However, macros defined inside of any *block* statement cannot be imported into another template.
23. “Block” scope describes the contents of *block* statements. Each separate block within a template has its own, independent scope, which does not have access or reference to any other block.
24. “Block” scope works slightly differently in root templates (i.e. templates that do not extend any other template) and child and descendant templates (i.e. templates that contain the *extends* tag).
25. In root templates, instances of the “block” scope will always be always evaluated by Blended-Lint as children of the “template” scope that contains it.
26. In child and descendant templates, a “block” scope instance will typically be evaluated twice by Blended-Lint. First, it will be evaluated on a stand-alone basis, as a child of the “render” scope. Second, it will be evaluated as a child of the root template’s “template” scope, in the same position as and in replacement of the *block* statement that shares its name. However, it is permitted for a child template to define a *block* that is not found in the root or any ancestor; in such a case, only the first evaluation will be made.
27. Some *block* statement contents will be evaluated by Blended-Lint yet again if invoked by a *parent* tag inside a child block. If any “block” scope contains a *parent* tag, then the closest ancestor to contain a *block* statement with the same name is found, and that block is evaluated as a child of the original “block” scope at the position of the *parent* tag. If, however, no parent *block* statement is found, then the *parent* tag is ignored.
28. Nested blocks are treated in much the same way that *parent* blocks are. They are evaluated as direct children of the template scope at the position of the outer-most containing block. Then, the contents, as-evaluated, are inserted into the template at the position of the block block declaration. Nested blocks are \*not\* evaluated at the position of their containing block, and do not have access to any element of any other block scope, even the scope of a containing block.
29. The order in which Blended-Lint evaluates blocks is important. First, all referenced templates are evaluated separately, including the root template. Then, the root template is evaluated, with each block being replaced by the block with the same name that is defined in the furthest descendant template.
30. The *for...else* statement effectively has three distinct scoping levels: the “iterator” scope, the “loop” scope, and the “else” scope. The *for...else* statement’s “iterator” scope persists for the duration of the tag’s execution, and contains the named item variables and the *forloop* variable. The *for...else* statement’s “loop” scope is a child scope encapsulated by the “iterator” scope. By this arrangement, the *forloop* variable and the named item variable(s) are prohibited from being modified inside the “loop” scope.
31. The *for...else* statement’s loop scope (which is coincident with the contents of the template segment that follows the *for* tag, termed the “loop segment”) is assumed to be cleared upon each iteration; therefore, no value may be accessed at the top of a *for* statement’s loop segment that is set at the bottom of the loop segment.
32. Because the *for...else* statement’s “iterator” scope and “loop” scope are child and descendant of whatever scope contains the *for...else* statement itself, any and all variables defined within the *for...else* statement fall out of scope when the *for...else* statement exits.
33. The *for...else* statement’s “else” scope (which is coincident with the contents of the template segment that follows the *else* tag, termed the “else segment”) is a sibling of the “iterator” scope, and therefore only shares with it and the “loop” scope whatever variables are defined in the scope that contains the *for...else* statement itself. It does not have access to the *forloop* variable nor the named item variables from the “iterator” scope.
34. *If...else* statements share scope with their container. Variables may be set inside *if...else* statements and used outside the statements; also, variables that are set before entering an *if...else* statement can be overridden inside the statement.
35. However, a variable set inside an *if...else* statement may only be accessed after that statement if the variable is guaranteed to have already been set. If a variable is used after an *if...else* statement, that variable MUST have been (a) defined before entering the *if...else* statement, or (b) defined in both the *if* section and the *else* section of the *if...else* statement. If a variable is set only in the *if* section, or only in the *else* section, without being defined earlier on within the same scope, then it will not be available after the *if...else* statement.
36. The *import* tag adds a module to the current scope. The macros declared inside the imported file, and only those macros, are added as members of the module, accessible through dot notation. Because the imported template is independently evaluated by Blended-Lint prior to importation, it is ensured that all macros accessible via the module will be without conflict amongst themselves. All other contents of the imported template (including any variables set or macros or modules imported) are ignored and discarded upon import. Variable data (i.e. non-macro data) is not imported.
37. The *from...import* tag adds macro names to the current scope. Unlike the *macro* tag declaration, the macros added using *from...import* are not accessible from inside macros that are defined within the same scope. As mentioned above, if an import is required, a macro will need to contain the import instruction inside the “macro” scope itself.
38. When any template file is imported--whether it is imported using the *from...import* tag or the *import* tag--that file is evaluated without having any reference to the current namespace that contains the tag. The imported file shares the “built-in” scope and the “user-function” scope of the master template, but it is provided with an empty “application” scope before it is evaluated.
39. The *include* tag makes no modification to any scope of the master template; it neither adds nor modifies any object. The *include* tag simply renders the referenced template in place.
40. An included template shares the “built-in” scope and the “user-function” scope of the template from which it has been referenced (i.e. the master template). The “application” scope of the included template varies depending on whether or not the *only* keyword is present inside the *include* tag. If the *only* keyword is present, then the “application” scope of the included template is populated only by the dictionary passed in using the *with* keyword (if present). If the *only* keyword is missing, then the “application” scope for the included template is populated by the master template’s entire namespace at the point of the *include* template’s evaluation, optionally updated and overwritten by the contents of the dictionary passed into the *import* tag using the *with* keyword. Below the level of the “application” scope (starting with the “template” scope), each imported template is to be analyzed as per the rules described herein, as if it were a standalone template.
41. No template containing *block* tags or an *extends* tag may be imported with the *import* tag or *from...import* tag, nor included with the *include* tag.
42. The “block”, “super”, “\_”, “on”, “off” and “forloop” variable names are treated as reserved keywords that may not be used as variable names, nor accessed within expressions (with the exception of the “forloop” dictionary being referenced inside the “loop” scope). Certain function names are also reserved. Specifically, “super” and “\_” are reserved as function names and may not be used by user functions or macros. “True”, “true”, “False” and “false” are similarly reserved.

In order to facilitate semantic analysis that includes application scope and user-function scope, schema-definition and user-function-definition files *may* be provided. The format and location of these optional files is specified [here](about:blank) [TBD]. If these files are specified, the lint will use them to populate the “user-function” scope and the “application” scope for purposes of analysis.

If schema and user-function definition files are *not* provided, then two things are assumed by the lint: (1) any variable that is accessed by a tag or expression that has not previously been set is assumed to belong to the application scope; and (2) any function/macro that is called that has not previously been imported via *from...import* or declared via *macro* is assumed to be a function belonging to the user-function scope.

With regards to any template inserted via the *include* tag: FIRST, an isolated semantic analysis is performed on the included template; then, SECOND, semantic analysis is performed on the master template, whereupon reaching the *include* tag it is determined whether or not the included template’s application scope expectation is satisfied by the values being passed into that template by the master template.

An included template’s application scope expectation is defined by that file’s schema-definition and user-function-definition files if they exist; if they don’t exist, it is defined as all of the “application” and “user-function” scoped values consumed and required by that template in its execution. If the *only* keyword IS USED in the *include* tag, then the included template’s application scope expectation is only compared to whatever values are supplied using the *include* tag’s *with* argument, to determine if they are sufficient. However, if the *only* keyword IS NOT USED in the *include* tag, the contents of the current scope, plus whatever values are supplied using the *include* tag’s *with* argument, are compared to the included template’s application scope expectation.

If no schema-definition or user-function-definition files are provided for the master template--and by implication if the application scope is to be dynamically calculated for the master template--then the application scope expectation of the included template is accreted to the application scope of the master template.

In order to allow proper evaluation of the external template references in the *extends*, *import, from...import*, and *include* tags, Blended-Lint needs to know where to look for those templates:

1. By default, Blended-Lint will search for named templates in the current directory.
2. The template directory can be changed using the --root flag.
3. In order to analyze template objects (template source strings compiled using the *template* filter), or in order to load a file location from a string in the context, a JSON string defining values for those variables can be passed into Blended-Lint using the *--context* flag. Alternately, if no *--context* flag is specified, Blended-Lint will look for a JSON file with a name in the form *“<template\_name>.context”* in the same directory as the template file.
4. If Blended-Lint encounters a template object, it will look for the referenced template source inside the context JSON, and perform analysis on that. If the context JSON specifies the filesystem location of the template source file (using a JSON $ref), Blended-Lint will perform analysis on that file with reference to that file. If the context JSON includes the template source inline, then Blended-Lint will perform analysis on that data without reference to any file.
5. If Blended-Lint cannot follow a template reference because the reference is in the form of an undefined variable, or because no file can be found at the indicated address, then a warning is displayed. However, Blended-Lint will nonetheless perform a complete analysis of the file(s) to which it has access.

1. Instead, you would implement the functionality as a recursive macro:

   {%- macro get\_value(start, index) -%}

   {%- if index >= 100 -%}

   {{- start+1 -}}

   {%- else -%}

   {{- get\_value(start+1, index+1) -}}

   {%- endif -%}

   {%- endmacro -%}

   {{- get\_value(1000, 0) -}}

   Of course, the last thing you would want to do is calculate a number like this in this way in a template language. We use this example here because it is simple to understand. [↑](#footnote-ref-0)