Functions in Terraform

Functions are expressions to transform and combine values in order to manipulate these values to be used in other ways. Functions can also be nested within each other. Most Terraform functions follow a common syntax, for example: <FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>)

Terraform functions are **built-in, reusable code blocks that perform specific tasks within Terraform configurations**
The general syntax for function calls is a function name followed by comma-separated arguments in parentheses: function (argument1, argument2)

Example:

```
> max(5, 12, 9)
12
```

These functions are split into multiple categories:

- String
- Numeric
- Collection
- Date and Time
- Crypto and Hash
- Filesystem
- IP Network
- Encoding
- Type Conversion

ToType Functions

ToType is not an actual function; rather, many functions can help you change the type of a variable to another type.

tonumber(argument) – With this function, you can change a string to a number, anything else apart from another number and null will result in an error.

tostring(argument) – Changes a number/bool/string/null to a string.

tobool(argument) – Changes a string (only “true” or “false”)/bool/null to a bool.

tolist(argument) – Changes a set to a list.

toset(argument) – Changes a list to a set.

tomap(argument) – Converts its argument to a map.

In Terraform, you are rarely going to need to use these types of functions, but I still thought they are worth mentioning.

format(string_format, unformatted_string)

The format function is similar to the printf function in C and works by formatting a number of values according to a specification string.

It can be used to build different strings that may be used in conjunction with other variables. Here is an example of how to use this function:

```
locals {
 string1       = "str1"
 string2       = "str2"
 int1          = 3
 apply_format  = format("This is %s", local.string1)
 apply_format2 = format("%s_%s_%d", local.string1, local.string2, local.int1)
}

output "apply_format" {
 value = local.apply_format
}
output "apply_format2" {
 value = local.apply_format2
}
```

This will result in:

```
apply_format  = "This is str1"
apply_format2 = "str1_str2_3"
```

formatlist(string_format, unformatted_list)

The formatlist function uses the same syntax as the format function but changes the elements in a list.

Here is an example of how to use this function:

```
locals {
 format_list = formatlist("Hello, %s!", ["A", "B", "C"])
}

output "format_list" {
 value = local.format_list
}
```

The output will be:

```
format_list = tolist(["Hello, A!", "Hello, B!", "Hello, C!"])
```

length(list / string / map)

Returns the length of a string, list, or map.

```
locals {
 list_length   = length([10, 20, 30])
 string_length = length("abcdefghij")
}

output "lengths" {
 value = format("List length is %d. String length is %d", local.list_length, local.string_length)
}
```

This will result in:

```
lengths = "List length is 3. String length is 10"
```

join(separator, list)

Another useful function in Terraform is [“join”](). This function creates a string by concatenating together all elements of a list and a separator. For example, consider the following code:

```
locals {
 join_string = join(",", ["a", "b", "c"])
}

output "join_string" {
 value = local.join_string
}
```

The output of this code will be “a, b, c”.

try(value, fallback)

Sometimes, you may want to use a value if it is usable but fall back to another value if the first one is unusable. This can be achieved using the “try” function.

For example:

```
locals {
 map_var = {
   test = "this"
 }
 try1 = try(local.map_var.test2, "fallback")
}

output "try1" {
 value = local.try1
}
```

The output of this code will be “fallback”, as the expression local.map_var.test2 is unusable.

can(expression)

A useful function for validating variables is “can”. It evaluates an expression and returns a boolean indicating if there is a problem with the expression. For example:

```
variable "a" {
 type = any
 validation {
  condition     = can(tonumber(var.a))
  error_message = format("This is not a number: %v", var.a)
 }
 default = "1"
}
```

The validation in this code will give you an error: “This is not a number: 1”.

flatten(list)

In Terraform, you may work with complex data types to manage your infrastructure. In these cases, you may want to flatten a list of lists into a single list.

This can be achieved using the “flatten” function, as in this example:

```
locals {
 unflatten_list = [[1, 2, 3], [4, 5], [6]]
 flatten_list   = flatten(local.unflatten_list)
}

output "flatten_list" {
 value = local.flatten_list
}
```

The output of this code will be [1, 2, 3, 4, 5, 6].

keys(map) & values(map)

It may be useful to extract the keys or values from a map as a list. This can be achieved using the “keys” or “values” functions, respectively.

For example:

```
locals {
 key_value_map = {
   "key1" : "value1",
   "key2" : "value2"
 }
 key_list   = keys(local.key_value_map)
 value_list = values(local.key_value_map)
}

output "key_list" {
 value = local.key_list
}

output "value_list" {
 value = local.value_list
}
```

The output of this code will be:

```
key_list = ["key1", "key2"]
value_list = ["value1", "value2"]
```

slice(list, startindex, endindex)

Slice returns consecutive elements from a list from a startindex (inclusive) to an endindex (exclusive).

```
locals {
 slice_list = slice([1, 2, 3, 4], 2, 4)
}



output "slice_list" {
 value = local.slice_list
}
```

The output for the above would be slice_list = [3].

range

Creates a range of numbers:

- one argument(limit)
- two arguments(initial_value, limit)
- three arguments(initial_value, limit, step)

```
locals {
 range_one_arg    = range(3)
 range_two_args   = range(1, 3)
 range_three_args = range(1, 13, 3)
}

output "ranges" {
 value = format("Range one arg: %v. Range two args: %v. Range three args: %v",
local.range_one_arg, local.range_two_args, local.range_three_args)
}
```

The output for this code would be:

```
range = "Range one arg: [0, 1, 2]. Range two args: [1, 2]. Range three args: [1, 4, 7, 10]"
```

lookup(map, key, fallback_value)

Retrieves a value from a map using its key. If the value is not found, it will return the default value instead.

```
locals {
 a_map = {
   "key1" : "value1",
   "key2" : "value2"
 }
 lookup_in_a_map = lookup(local.a_map, "key1", "test")
}



output "lookup_in_a_map" {
 value = local.lookup_in_a_map
}
```

This will return: lookup_in_a_map = "key1"

Learn more about the Terraform lookup function.

concat(lists)

Takes two or more lists and combines them in a single one.

```
locals {
 concat_list = concat([1, 2, 3], [4, 5, 6])
}



output "concat_list" {
 value = local.concat_list
}
```

This will return: concat_list = [1, 2, 3, 4, 5, 6]

merge(maps)

The merge function takes one or more maps and returns a single map that contains all of the elements from the input maps. The function can also take objects as input, but the output will always be a map.

Let’s take a look at an example:

```
locals {
 b_map = {
   "key1" : "value1",
   "key2" : "value2"
 }
 c_map = {
   "key3" : "value3",
   "key4" : "value4"
 }
 final_map = merge(local.b_map, local.c_map)
}


output "final_map" {
 value = local.final_map
}
```

The above code will return:

```
final_map = {
  "key1" = "value1"
  "key2" = "value2"
  "key3" = "value3"
  "key4" = "value4"
}
```

zipmap(key_list, value_list)

Constructs a map from a list of keys and a list of values.

```
locals {
 key_zip    = ["a", "b", "c"]
 values_zip = [1, 2, 3]
 zip_map    = zipmap(local.key_zip, local.values_zip)
}

output "zip_map" {
 value = local.zip_map
}
```

This code will return:

```
zip_map = {
  "a" = 1
  "b" = 2
  "c" = 3
}
```

expanding function argument …

This special argument works only in function calls and expands a list into separate arguments. Useful when you want to merge all maps from a list of maps.

```
locals {
 list_of_maps = [
   {
     "a" : "a"
     "d" : "d"
   },
   {
     "b" : "b"
```

```
    "e" : "e"
  },
  {
    "c" : "c"
    "f" : "f"
  },
 ]
 expanding_map = merge(local.list_of_maps...)
}

output "expanding_map" {
 value = local.expanding_map
}
```

This will result in:

```
expanding_map = {
  "a" = "a"
  "b" = "b"
  "c" = "c"
  "d" = "d"
  "e" = "e"
  "f" = "f"
}
```

file(path_to_file)

Reads the content of a file as a string and can be used in conjunction with other functions like jsondecode / yamldecode.

```
locals {
  a_file = file("./a_file.txt")
}

output "a_file" {
 value = local.a_file
}
```

The output would be the content of the file called a_file as a string.

templatefile(path, vars)

Reads the file from the specified path and changes the variables specified in the file between the interpolation syntax ${ ... } with the ones from the vars map.

```
locals {
 a_template_file = templatefile("./file.yaml", { "change_me" : "awesome_value" })
}



output "a_template_file" {
 value = local.a_template_file
}
```

This will change the ${change_me} variable to awesome_value.

jsondecode(string)

Interprets a string as json.

```
locals {
 a_jsondecode = jsondecode("{\"hello\": \"world\"}")
}



output "a_jsondecode" {
 value = local.a_jsondecode
}
```

This will return:

```
jsondecode = {
 "hello" = "world"
}
```

jsonencode(string)

Encodes a value to a string using json.

```
locals {
 a_jsonencode = jsonencode({ "hello" = "world" })
}



output "a_jsonencode" {
 value = local.a_jsonencode
}
```

This results in:

```
a_jsonencode = "{\"hello\":\"world\"}"
```

yamldecode(string)

Parses a string as a subset of YAML and produces a representation of its value.

```
locals {
 a_yamldecode = yamldecode("hello: world")
}



output "a_yamldecode" {
 value = local.a_yamldecode
}
```

This returns:

```
a_yamldecode = {
 "hello" = "world"
}
```

yamlencode(value)

Encodes a given value to a string using YAML.

```
locals {
 a_yamlencode = yamlencode({ "a" : "b", "c" : "d" })
}



output "a_yamlencode" {
 value = local.a_yamlencode
}
```

This will return:

```
a_yamlencode = <<EOT
"a": "b"
"c": "d"

EOT
```