Terraform Functions

What are Terraform Functions? *Terraform functions* are **built-in, reusable code blocks that perform specific tasks within Terraform configurations**. They make your code more dynamic and ensure your configuration is DRY. Functions allow you to perform various operations, such as converting expressions to different data types, calculating lengths, and building complex variables.

These functions are split into multiple categories:

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

This split, however, can become overwhelming to someone who doesn't have that much experience with Terraform. For example, the formatlist list function is considered to be a string function even though it modifies elements from a list. A list is a collection though; some may argue that this function should be considered a collection function, but still, at its core, it does change to strings.

For that particular reason, I won't specify the function type when I'll describe them, but just go with what you can do with them. Of course, I will not go through all of the available functions, but through the ones I am using throughout my configurations.

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.str
}
```

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.r
}
```

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,
```

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"`

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
```

You can use Terraform functions to make your life easier and to write better code. Keeping your configuration as DRY as possible improves readability and makes updating it easier, so functions are a must-have in your configurations.

These are just the functions that I am using the most, but some honorable mentions are `element`, `base64encode`, `base64decode`, `formatdate`, `uuid`, and `distinct`.

Terraform Expressions

Expressions are the core of HCL itself – the logic muscle of the entire language. *Terraform expressions* allow you to get a value from somewhere, calculate or evaluate it. You can use them to refer to the value of something, or extend the logic of a component – for example, make one copy of the resource for each value contained within a variable, using it as an argument.

They are used pretty much everywhere – the most simple type of expression would be a literal value – so, there is a great chance that you have already used them before.

As it is an extremely extensive topic, I couldn't possibly fit everything into this single article. Instead, I can talk about what in my humble opinion are the most notable/interesting expressions: Operators, Conditional expressions, Splat Expressions, Constraints, and Loops.

Operators

Dedicated to logical comparison and arithmetic operations, operators are mostly used for doing math and basic Bool's algebra. If you need to know if number A equals number B, add them together, or determine if both boolean A and boolean B are "true", Terraform offers the following operators:

Types of Terraform Operators

- Arithmetic operators – the basic ones for typical math operations (+, -, *, /) and two special ones: "X % Y" would return the remainder of dividing X by Y, and "-X", which would return X multiplied by -1. Those can only be used with numeric values.
- Equality operators – "X == Y" is *true*, if X and Y have both the same value and type, "X != Y" would return *false* in this case. This one will work with any type of value.

- Comparison operators – “<, >, <=, >=” – exclusive to numbers, returns *true* or *false* depending on the condition.
- Logical operators – the Bool’s algebra part of the pack, work only with the boolean values of *true* and *false*.
  – “X || Y” returns *true* if either X or Y is *true*, *false* if any of them is *false*.
  – “X && Y” returns *true* only, if both X and Y are *true*, *false* if any of them is *false.*
  – “!X” is true, if X is false, false if X is true.

Conditionals

Sometimes, you might run into a scenario where you’d want the argument value to be different, depending on another value. The conditional syntax is as such:

```
condition ? true_val : false_val
```

The condition part is constructed using previously described operators. In this example, the bucket_name value is based on the “test” variable—if it’s set to true, the bucket will be named “dev” and if it’s false, the bucket will be named “prod”:

```
bucket_name = var.test == true ? "dev" : "prod"
```

Splat Expressions

Splat expressions are used to extract certain values from complicated collections – like grabbing a list of attributes from a list of objects containing those attributes. Usually you would need an “for” expression to do this, but humans are lazy creatures who like to make complicated things simpler.

For example, if you had a list of objects such as these:

```
test_variable = [
 {
   name  = "Arthur",
   test  = "true"
 },
 {
   name  = "Martha"
   test  = "true"
 }
]
```

Instead of using the entire “for” expression:

```
[for o in var.test_variable : o.name]
```

you could go for the splat expression form:

```
var.test_variable[*].name
```

And in both cases, get the same result:

```
["Arthur", "Martha"]
```

Do note, that this behavior applies only if splat was used on a list, set, or tuple. Anything else (Except *null*) will be transformed into a tuple, with a single element inside, *null* will simply stay as is. This may be good or bad, depending on your use case.

Constraints

In simple terms, constraints regulate what can be what, and where something can, or cannot be used. There are two main types of constraints—*for* types and *versions*.

- Type constraints regulate the values of variables and outputs. For example, a string is represented by anything enclosed in quotes, bool value is either a literal *true* or *false*, a list is always opened with square brackets [ ], a map is defined with curly brackets { }.
- Version constraints usually apply to modules and regulate which versions should or should not be used.

Terraform Loops

Ah yes, the elephant in the room. Mentioned a few times, but still unexplained—until now.

Terraform loops are used to handle collections, and to produce multiple instances of a resource or module without repeating the code. There are three loops provided by Terraform to date:

Count

Count is the most primitive—it allows you to specify a whole number, and produces as many instances of something as this number tells it to. For example, the following would order Terraform to create ten S3 buckets:

```
resource "aws_s3_bucket" "test" {
 count = 10
[...]
}
```

When *count* is in use, each instance of a resource or module gets a separate index, representing its place in the order of creation. To get a value from a single resource created in this way, you must refer to it by its index value, e.g. if you wished to see the ID of the fifth created S3 bucket, you would need to call it as such:

```
aws_s3_bucket.test[5].id
```

Although this is fine for identical, or nearly identical objects, as previously mentioned, *count* is pretty primitive. When you need to use more distinct, complex values – *count* yields to *for_each*.

For_each

As mentioned earlier, sometimes you might want to create resources with distinct values associated with each one – such as names or parameters (memory or disk size for example). For_each will let you do just that. Merely provide a variable—map, or a set of strings, and the resources can access values contained within, via *each.key* and *each.value*:

```
test_map = {
 test1 = "test2",
 test2 = "test4"
}

resource "test_resource" "thing" {
 for_each = var.test_map

 test_attribute_1 = each.key
 test_attribute_2 = each.value
}
```

As you can see, for_each is quite powerful, but you haven't seen the best yet. By constructing a map of objects, you can leverage a resource or module to create multiple instances of itself, each with multiple declared variable values:

```
my_instances = {
 instance_1 = {
   ami   = "ami-00124569584abc",
   type  = "t2.micro"
 },
 instance_2 = {
   ami   = "ami-987654321xyzab",
   type  = "t2.large"
 },
}

resource "aws_instance" "test" {
 for_each = var.my_instances

 ami           = each.value["ami"]
 instance_type = each.value["type"]
}
```

Using this approach, you don't have to touch anything except the *.tfvars* file, to provide new instances of resources you have already declared in your configuration. Absolutely brilliant.