spacelift

// The Most Flexible IaC Platform

Terraform functions

tonumber(number)

Changes an expression to a number, if it cannot do it, you will receive an error.

Example definition:

```
locals {
  not_a_number = "1"
  a_number     = tonumber(local.not_a_number)
}
```

Output definition:

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

Literal output:

```
a_number = 1
```

tostring(string)

Changes an expression to a string, if it cannot do it, you will receive an error.

Example definition:

```
locals {
  not_a_string = 100
  a_string     = tostring(local.not_a_string)
}
```

Output definition:

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

Literal output:

```
a_string = "100"
```

tobool, tolist, toset, tomap have a similar behavior to tonumber and tostring

formatlist(string_format, unformatted_list)

Uses the same syntax as format, but changes the elements in a list.

Example definition:

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

Output definition:

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

Literal output:

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

length(list / string / map)

Returns the length of a string, list or map.

Example definition:

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

Output definition:

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

Literal output:

```
lengths = "list length is 3, string length is 10"
```

try(value, fallback)

Will try to use the first value, but will automatically fallback to the second one if the first one is unusable.
Try will not catch errors that are invalid before dynamic expressions evaluation (malformed references, undeclared variables, etc.).

Example definition:

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

Output definition:

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

Literal output:

```
try1 = "fallback"
```

can(expression)

Evaluates an expression and returns a boolean indicating if there is a problem with your expression.
Can is primarily used in variables validation, for other use cases you should use try as can doesn't have any fallback.

Example definition:

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

Literal output:

```
the validation will give you an error: this is not a number "1"
```

flatten(list)

Takes an arbitrary number of lists and turns them in a single list.
Really useful when you are using complex data types to manage your infrastructure.

Example definition:

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

Output definition:

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

Literal output:

```
flatten_list = [1, 2, 3, 4, 5, 6]
```

zipmap(key_list, value_list)

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

Example definition:

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

Output definition:

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

Literal output:

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

keys / values (map)

Return the keys / values from a map as a list.

Example definition:

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

Output definition:

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

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

Literal output:

```
key_list = ["key1", "key2"]

value_list = ["value1",
"value2"]
```

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.

Example definition:

```
locals {
  list_of_maps = [
    {
      "a" : "a"
      "d" : "d"
    },
    {
      "b" : "b"
      "e" : "e"
    },
    {
      "c" : "c"
      "f" : "f"
    },
  ]
  expanding_map = merge(local.list_of_maps...)
}
```

Output definition:

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

Literal output:

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

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.

Example definition:

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

Output definition:

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

Literal output:

```
this will change the ${change_me} variable to awesome_value
```

format(string_format, unformatted_string)

Similar to *printf* in C, works by formatting a number of values according to a specification string. Can be used to build different strings, that may be used in conjunction with other variables.

Example definition:

```
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 definition:

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

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

Literal output:

```
apply_format = "this is str1"

apply_format2 = "str1_str2_3"
```

range

Creates a range of numbers:
one argument: range(limit).
two arguments: range(initial_value, limit).
three arguments: range(initial_value, limit, step).

Example definition:

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

Output definition:

```
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)
}
```

Literal output:

```
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.

Example definition:

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

Output definition:

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

Literal output:

```
lookup_in_a_map = "value1"
```

merge(maps)

Takes a number of maps and or objects and returns a single map / object with a merged set of elements.

Example definition:

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

Output definition:

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

Literal output:

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

file(path_to_file)

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

Example definition:

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

Output definition:

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

Literal output:

```
content of a_file as a string
```

yamldecode(string)

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

Example definition:

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

Output definition:

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

Literal output:

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

yamlencode(value)

Encodes a given value to a string using YAML.

Example definition:

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

Output definition:

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

Literal output:

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

EOT
```

slice(list, startindex, endindex)

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

Example definition:

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

Output definition:

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

Literal output:

```
slice_list = [3]
```

jsondecode(string)

Interprets a string as json.

Example definition:

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

Output definition:

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

Literal output:

```
a_jsondecode = {"hello": "world"}
```

jsonencode(value)

Encodes a value to a string using json.

Example definition:

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

Output definition:

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

Literal output:

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

join(delimiter, list)

Creates a string by concatenating together all elements of a list and a separator.

Example definition:

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

Output definition:

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

Literal output:

```
join_string = "a, b, c"
```

concat(lists)

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

Example definition:

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

Output definition:

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

Literal output:

```
concat_list = [1, 2, 3, 4, 5, 6]
```