**TERRAFORM FUNCTIONS**

**Collection Function**:

**All True**:

All true returns true if all elements in a given collection are true or "true". It also returns true if the collection is empty.

alltrue(list)

**Examples**

> alltrue(["true", true])

true

> alltrue([true, false])

false

**Any True**:

anytrue returns true if any element in a given collection is true or "true". It also returns false if the collection is empty.

## Examples

> anytrue(["true"])

true

> anytrue([true])

true

> anytrue([true, false])

true

> anytrue([])

false

**Index** :

index finds the element index for a given value in a list.

## Examples

> index(["a", "b", "c"], "b")

1

**Keys**:

keys takes a map and returns a list containing the keys from that map.

The keys are returned in lexicographical order, ensuring that the result will be identical as long as the keys in the map don't change.

## Examples

> keys({a=1, c=2, d=3})

[

"a",

"c",

"d",

]

**Length**:

length determines the length of a given list, map, or string.

If given a list or map, the result is the number of elements in that collection. If given a string, the result is the number of characters in the string.

## Examples

> length([])

0

> length(["a", "b"])

2

> length({"a" = "b"})

1

> length("hello")

5

**List**:

The list function is no longer available. Prior to Terraform v0.12 it was the only available syntax for writing a literal list inside an expression, but Terraform v0.12 introduced a new first-class syntax.

To update an expression like list(a, b, c), write the following instead:

tolist([a, b, c])

**LoopkUp**:

lookup retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

## Examples

> lookup({a="ay", b="bee"}, "a", "what?")

ay

> lookup({a="ay", b="bee"}, "c", "what?")

what?

**Map**:

The map function is no longer available. Prior to Terraform v0.12 it was the only available syntax for writing a literal map inside an expression, but Terraform v0.12 introduced a new first-class syntax.

To update an expression like map("a", "b", "c", "d"), write the following instead:

tomap({

a = "b"

c = "d"

})

**Merge**:

merge takes an arbitrary number of maps or objects, and returns a single map or object that contains a merged set of elements from all arguments.

## Examples

> merge({a="b", c="d"}, {e="f", c="z"})

{

"a" = "b"

"c" = "z"

"e" = "f"

}

**Reverse**:

reverse takes a sequence and produces a new sequence of the same length with all of the same elements as the given sequence but in reverse order.

## Examples

> reverse([1, 2, 3])

[

3,

2,

1,

]

**Sort**:

sort takes a list of strings and returns a new list with those strings sorted lexicographically.

The sort is in terms of Unicode codepoints, with higher codepoints appearing after lower ones in the result.

## Examples

> sort(["e", "d", "a", "x"])

[

"a",

"d",

"e",

"x",

]

**Sum**:

sum takes a list or set of numbers and returns the sum of those numbers.

## Examples

> sum([10, 13, 6, 4.5])

33.5

**Values**:

values takes a map and returns a list containing the values of the elements in that map.

The values are returned in lexicographical order by their corresponding keys, so the values will be returned in the same order as their keys would be returned from [keys](https://www.terraform.io/docs/language/functions/keys.html).

## Examples

> values({a=3, c=2, d=1})

[

3,

2,

1,

]

**Encoding Functions**:

**Yamldecode**:

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

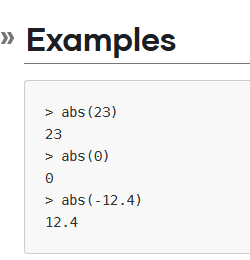
This function supports a subset of [YAML 1.2](https://yaml.org/spec/1.2/spec.html), as described below.

This function maps YAML values to [Terraform language values](https://www.terraform.io/docs/language/expressions/types.html) in the following way:

| **YAML type** | **Terraform type** |
| --- | --- |
| !!str | string |
| !!float | number |
| !!int | number |
| !!bool | bool |
| !!map | object(...) with attribute types determined per this table |
| !!seq | tuple(...) with element types determined per this table |
| !!null | The Terraform language null value |
| !!timestamp | string in [RFC 3339](https://tools.ietf.org/html/rfc3339) format |
| !!binary | string containing base64-encoded representation |

YAML is a complex language and it supports a number of possibilities that the Terraform language's type system cannot represent. Therefore this YAML decoder supports only a subset of YAML 1.2, with restrictions including the following:

* Although aliases to earlier anchors are supported, cyclic data structures (where a reference to a collection appears inside that collection) are not. If yamldecode detects such a structure then it will return an error.
* Only the type tags shown in the above table (or equivalent alternative representations of those same tags) are supported. Any other tags will result in an error.
* Only one YAML document is permitted. If multiple documents are present in the given string then this function will return an error.

 **YamlenCode**:

yamlencode encodes a given value to a string using [YAML 1.2](https://yaml.org/spec/1.2/spec.html) block syntax.

This function maps [Terraform language values](https://www.terraform.io/docs/language/expressions/types.html) to YAML tags in the following way:

| **Terraform type** | **YAML type** |
| --- | --- |
| string | !!str |
| number | !!float or !!int |
| bool | !!bool |
| list(...) | !!seq |
| set(...) | !!seq |
| tuple(...) | !!seq |
| map(...) | !!map |
| object(...) | !!map |
| Null value | !!null |

## Examples

> yamlencode({"a":"b", "c":"d"})

"a": "b"

"c": "d"

> yamlencode({"foo":[1, 2, 3], "bar": "baz"})

"bar": "baz"

"foo":

- 1

- 2

- 3

> yamlencode({"foo":[1, {"a":"b","c":"d"}, 3], "bar": "baz"})

"bar": "baz"

"foo":

- 1

- "a": "b"

"c": "d"

- 3

File System Function:

file reads the contents of a file at the given path and returns them as a string.

## Examples

> file("${path.module}/hello.txt")

Hello World

**Fileexist**s:

fileexists determines whether a file exists at a given path.

## Examples

> fileexists("${path.module}/hello.txt")

true

**Fileset**:

fileset enumerates a set of regular file names given a path and pattern. The path is automatically removed from the resulting set of file names and any result still containing path separators always returns forward slash (/) as the path separator for cross-system compatibility.

## Examples

> fileset(path.module, "files/\*.txt")

[

"files/hello.txt",

"files/world.txt",

]

> fileset(path.module, "files/{hello,world}.txt")

[

"files/hello.txt",

"files/world.txt",

]

> fileset("${path.module}/files", "\*")

[

"hello.txt",

"world.txt",

]

> fileset("${path.module}/files", "\*\*")

[

"hello.txt",

"world.txt",

"subdirectory/anotherfile.txt",

]

**Templatefile**:

templatefile reads the file at the given path and renders its content as a template using a supplied set of template variables.

## Examples

### [»](https://www.terraform.io/docs/language/functions/templatefile.html" \l "lists)Lists

Given a template file backends.tpl with the following content:

%{ for addr in ip\_addrs ~}

backend ${addr}:${port}

%{ endfor ~}

The templatefile function renders the template:

> templatefile("${path.module}/backends.tpl", { port = 8080, ip\_addrs = ["10.0.0.1", "10.0.0.2"] })

backend 10.0.0.1:8080

backend 10.0.0.2:8080

**Formatdate**:

formatdate converts a timestamp into a different time format.

## Examples

> formatdate("DD MMM YYYY hh:mm ZZZ", "2018-01-02T23:12:01Z")

02 Jan 2018 23:12 UTC

> formatdate("EEEE, DD-MMM-YY hh:mm:ss ZZZ", "2018-01-02T23:12:01Z")

Tuesday, 02-Jan-18 23:12:01 UTC

> formatdate("EEE, DD MMM YYYY hh:mm:ss ZZZ", "2018-01-02T23:12:01-08:00")

Tue, 02 Jan 2018 23:12:01 -0800

> formatdate("MMM DD, YYYY", "2018-01-02T23:12:01Z")

Jan 02, 2018

> formatdate("HH:mmaa", "2018-01-02T23:12:01Z")

11:12pm

**Timeadd**:

timeadd adds a duration to a timestamp, returning a new timestamp.

## Examples

> timeadd("2017-11-22T00:00:00Z", "10m")

2017-11-22T00:10:00Z

**Timestamp**:

timestamp returns a UTC timestamp string in [RFC 3339](https://tools.ietf.org/html/rfc3339) format.

## Examples

> timestamp()

2018-05-13T07:44:12Z

**IP Network Function**:

**Cidrsubnet**:

cidrsubnet calculates a subnet address within given IP network address prefix.

cidrsubnet(prefix, newbits, netnum)

## Examples

> cidrsubnet("172.16.0.0/12", 4, 2)

172.18.0.0/16

> cidrsubnet("10.1.2.0/24", 4, 15)

10.1.2.240/28

> cidrsubnet("fd00:fd12:3456:7890::/56", 16, 162)

fd00:fd12:3456:7800:a200::/72

**Type Conversions Function**:

**Can**:

can evaluates the given expression and returns a boolean value indicating whether the expression produced a result without any errors.

## Examples

> local.foo

{

"bar" = "baz"

}

> can(local.foo.bar)

true

> can(local.foo.boop)

false

**Defaults**:

The defaults function is a specialized function intended for use with input variables whose type constraints are object types or collections of object types that include optional attributes.

defaults(input\_value, defaults)

**Tolist**:

tolist converts its argument to a list value.

## Examples

> tolist(["a", "b", "c"])

[

"a",

"b",

"c",

]

**Tomap**:

tomap converts its argument to a map value.

## Examples

> tomap({"a" = 1, "b" = 2})

{

"a" = 1

"b" = 2

}

**Tryfunction**:

try evaluates all of its argument expressions in turn and returns the result of the first one that does not produce any errors.

## Examples

> local.foo

{

"bar" = "baz"

}

> try(local.foo.bar, "fallback")

baz

> try(local.foo.boop, "fallback")

fallback