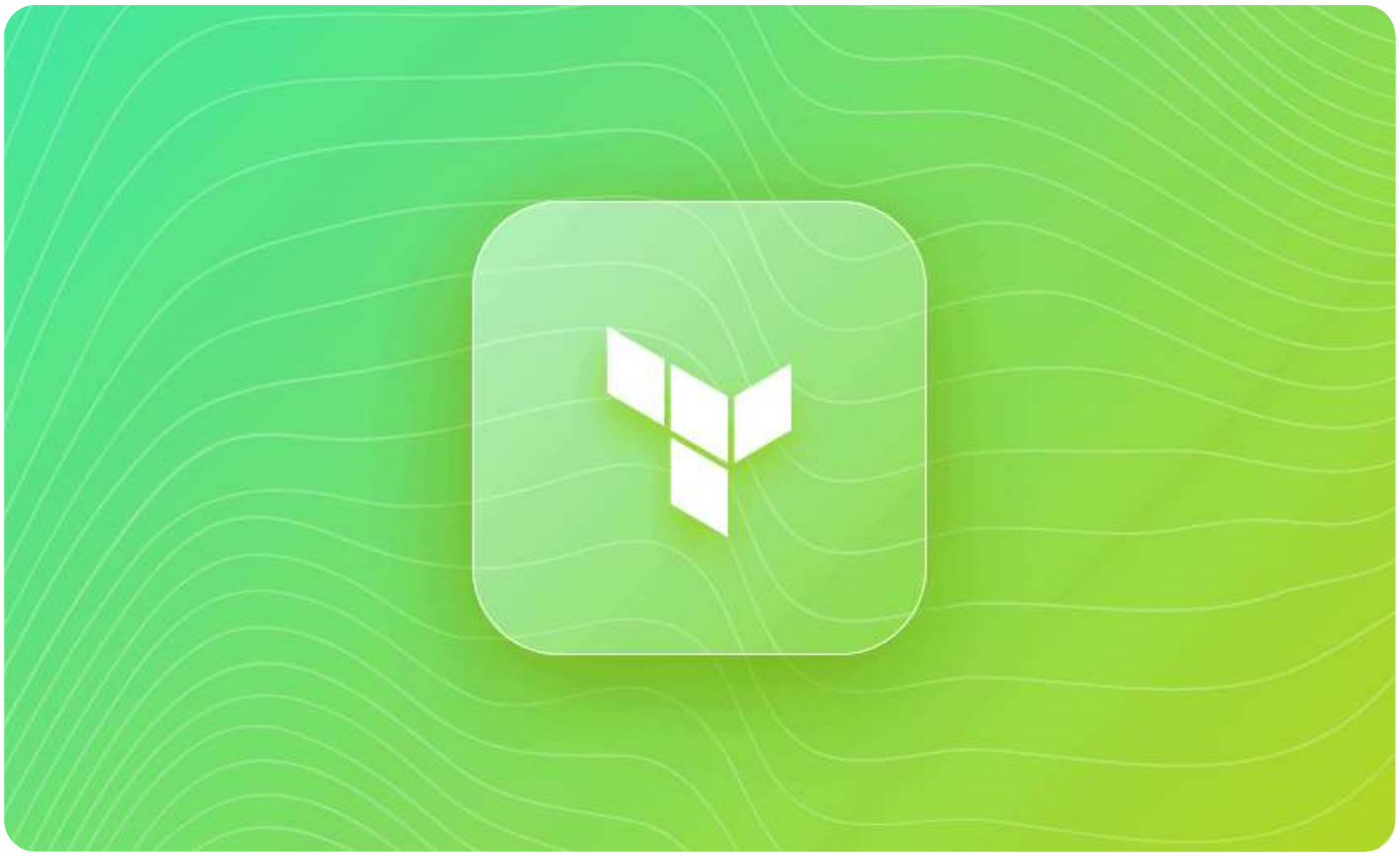
TERRAFORM

How to Use Terraform Variables (Locals, Input, Output, Environment)

Sumeet Ninawe

Updated 12 Oct 2023 · 18 min read



Variables are fundamental constructs in every programming language because they are inherently useful in building dynamic programs. We use variables to store temporary values so that they can assist programming logic in simple as well as complex programs.

In this post, we discuss how variables are used in Terraform. Terraform uses HCL (Hashicorp Configuration Language) to encode infrastructure. It is declarative in nature, meaning several blocks of code are declared to create a desired set of infrastructure.

with function in the form of arguments, return values, and local variables. These are analogous to input variables, output variables, and local variables in Terraform.

We will cover:

1. Local variables
2. Input variables
3. Variable Substitution using CLI and .tfvars
4. Environment variables
5. Precedence
6. Output variables
7. Using variables in for_each loop
8. Limitations

Note: New versions of Terraform will be placed under the BUSL license, but everything created before version 1.5.x stays open-source. OpenTofu is an open-source version of Terraform that will expand on Terraform's existing concepts and offerings. It is a viable alternative to HashiCorp's Terraform, being forked from Terraform version 1.5.6. OpenTofu retained all the features and functionalities that had made Terraform popular among developers while also introducing improvements and enhancements. OpenTofu is the future of the Terraform ecosystem, and having a truly open-source project to support all your IaC needs is the main priority.

be used in the configuration. The values can be hard-coded or be a reference to another variable or resource.

Local variables are accessible within the module/configuration where they are declared. Let us take an example of creating a configuration for an EC2 instance using local variables. Add this to a file named `main.tf`.

```
locals {
 ami  = "ami-0d26eb3972b7f8c96"
 type = "t2.micro"
 tags = {
   Name = "My Virtual Machine"
   Env  = "Dev"
 }
 subnet = "subnet-76a8163a"
 nic    = aws_network_interface.my_nic.id
}

resource "aws_instance" "myvm" {
 ami           = local.ami
 instance_type = local.type
 tags          = local.tags

 network_interface {
   network_interface_id = aws_network_interface.my_nic.id
   device_index         = 0
 }
}
```

In this example, we have declared all the local variables in the locals block. The variables represent the AMI ID (`ami`), Instance type (`type`), Subnet Id (`subnet`), Network Interface (`nic`) and Tags (`tags`) to be assigned for the given EC2 instance.

In the `aws_instance` resource block, we used these variables to provide appropriate values required for the given attribute. Notice how the local variables are being referenced using a `local` keyword (without 's').

Usage of local variables is similar to data sources. However, they have a completely different purpose. Data sources fetch valid values from the cloud provider based on the query filters we provide. Whereas we can set our desired values in local variables — they are **not dependent on the cloud providers.**

It is indeed possible to assign a value from a data source to a local variable. Similarly to how we have done it to create the `nic` local variable, it refers to the `id` argument in the `aws_network_interface` resource block.

As a best practice, try to keep the number of local variables to a minimum. Using many local variables can make the code hard to read.

If you want to know more about locals, see: Terraform Locals: What Are They, How to Use Them, Examples

f the configuration
s. The difference

Further, the main function of the input variables is to act as inputs to modules. Modules are self-contained pieces of code that perform certain predefined deployment tasks. Input variables declared within modules are used to accept values from the root directory.

Additionally, it is also possible to set certain attributes while declaring input variables, as below:

- `type` — to identify the type of the variable being declared.
- `default` — default value in case the value is not provided explicitly.
- `description` — a description of the variable. This description is also used to generate documentation for the module.
- `validation` — to define validation rules.
- `sensitive` — a boolean value. If true, Terraform masks the variable's value anywhere it displays the variable.

Input variables **support multiple data types**. They are broadly categorized as simple and complex. `String`, `number`, `bool` are simple data types, whereas `list`, `map`, `tuple`, `object`, and `set` are complex data types.

The following snippets provide examples for each of the types we listed.

String type

The string type input variables are used to accept values in the form of UNICODE

elow.

The string type input variables also support a heredoc style format where the value being accepted is a longer string separated by new line characters. The start and end of the value is indicated by “EOF” (End Of File) characters. An example of the same is shown below.

```
variable "string_heredoc_type" {
 description = "This is a variable of type string"
 type        = string
 default     = <<EOF
hello, this is Sumeet.
Do visit my website!
EOF
}
```

Number type

The number type input variable enables us to define and accept numerical values as inputs for their infrastructure deployments. For example, these numeric values can help define the desired number of instances to be created in an auto-scaling group. The code below defines a number type input variable in any given Terraform config.

The boolean type input variable is used to define and accept true/false values as inputs for infrastructure deployments to incorporate logic and conditional statements into the Terraform configurations. Boolean input variables are particularly useful for enabling or disabling certain features or behaviors in infrastructure deployments.

An example of a boolean variable is below.

```
variable "boolean_type" {
 description = "This is a variable of type bool"
 type        = bool
 default     = true
}
```

Terraform list variable

Terraform list variables allow to define and accept a collection of values as inputs for infrastructure deployments. A list is an ordered sequence of elements, and it can contain any data type, such as strings, numbers, or even other complex data structures. However, a single list cannot have multiple data types.

List type input variables are particularly useful in scenarios where we need to provide multiple values of the same type, such as a list of IP addresses, a set of ports, or a collection

ngs.

Map type

The map type input variable enables us to define and accept a collection of key-value pairs as inputs for our infrastructure deployments. A map is a complex data structure that associates values with unique keys, similar to a dictionary or an object in other programming languages. For example, a map can be used to specify resource tags, environment-specific settings, or configuration parameters for different modules.

The example below shows how a map of string type values is defined in Terraform.

```
variable "map_type" {
 description = "This is a variable of type map"
 type        = map(string)
 default     = {
   key1 = "value1"
   key2 = "value2"
 }
}
```

Learn how to use the Terraform map variable.

key-value pairs,
ing value. The
t of properties or

The variable below demonstrates how an object type input variable is defined with multi-typed properties.

```
variable "object_type" {
 description = "This is a variable of type object"
 type        = object({
   name    = string
   age     = number
   enabled = bool
 })
 default = {
   name    = "John Doe"
   age     = 30
   enabled = true
 }
}
```

Tuple type

A tuple is a fixed-length collection that can contain values of different data types. The key difference between tuples and lists are:

1. Tuples have a fixed length, as against lists.

types. Whereas lists

es, it is possible to

```
 description = "This is a variable of type tuple"
 type        = tuple([string, number, bool])
 default     = ["item1", 42, true]
}
```

Set type

A set is an unordered collection of distinct values, meaning each element appears only once within the set. As against lists, sets enforce uniqueness – each element can appear only once within the set. Sets support various inbuilt operations such as union, intersection, and difference, which are used to combine or compare sets.

An example of a set type input variable is below.

```
variable "set_example" {
 description = "This is a variable of type set"
 type        = set(string)
 default     = ["item1", "item2", "item3"]
}
```

Map of objects

One of the widely used complex input variable types is map(object). It is a data type that

e objects with
we define the
sponding types
own set of

```
variable "map_of_objects" {
  description = "This is a variable of type Map of objects"
  type = map(object({
    name = string,
    cidr = string
  }))
  default = {
    "subnet_a" = {
    name = "Subnet A",
    cidr = "10.10.1.0/24"
    },
   "subnet_b" = {
    name = "Subnet B",
    cidr = "10.10.2.0/24"
    },
   "subnet_c" = {
    name = "Subnet C",
    cidr = "10.10.3.0/24"
    }
  }
}
```

List of objects

This type of variable is similar to the Map of objects, except that the objects are not referred

presented in the

key value.

```
variable "list_of_objects" {
  description = "This is a variable of type List of objects"
  type = list(object({
    name = string,
    cidr = string
  }))
  default = [
    {
      name = "Subnet A",
      cidr = "10.10.1.0/24"
    },
    {
      name = "Subnet B",
      cidr = "10.10.2.0/24"
    },
    {
      name = "Subnet C",
      cidr = "10.10.3.0/24"
    }
  ]
}
```

Terraform Input Variables Example

e variables instead

les.tf and add the

```
 validation {
   condition     = length(var.ami) > 4 && substr(var.ami, 0, 4) == "ami-"
   error_message = "Please provide a valid value for variable AMI."
 }
}

variable "type" {
 type        = string
 description = "Instance type for the EC2 instance"
 default     = "t2.micro"
 sensitive   = true
}

variable "tags" {
 type = object({
   name = string
   env  = string
 })
 description = "Tags for the EC2 instance"
 default = {
   name = "My Virtual Machine"
   env  = "Dev"
 }
}

variable "subnet" {
 type        = string
 description = "Subnet ID for network interface"
 default     = "subnet-76a8163a"
```


h the simple data
alue pairs with
ion and default.

Let us now modify `main.tf` to use variables declared above.

```
resource "aws_instance" "myvm" {
 ami           = var.ami
 instance_type = var.type
 tags          = var.tags

 network_interface {
   network_interface_id = aws_network_interface.my_nic.id
   device_index         = 0
 }
}

resource "aws_network_interface" "my_nic" {
 description = "My NIC"
 subnet_id   = var.subnet

 tags = {
   Name = "My NIC"
 }
}
```

Within the *resource* blocks, we have simply used these variables by using `var.<variable name>` format. When you proceed to plan and apply this configuration, the variable values

mple plan output.

To check how validation works, modify the default value provided to the `ami` variable. Make sure to change the `ami-` part since validation rules are validating the same. Run the `plan` command, and see the output. You should see the error message thrown on the console as below.

```
╷
│ Error: Invalid value for variable
│
│   on variables.tf line 1:
│    1: variable "ami" {
│
│ Please provide a valid value for variable AMI.
│
│ This was checked by the validation rule at variables.tf:6,3-13.
```

Also, notice how the `type` value is represented in the plan output. Since we have marked it as `sensitive`, its value is not shown. Instead, it just displays `sensitive`.

```
  + id                                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_state                                   = (known after apply)
                                                         nsitive)
                                                         fter apply)
                                                         n after apply)
```

💡 You might also like:

- 5 Ways to Manage Terraform at Scale
- How to Automate Terraform Deployments and Infrastructure Provisioning
- How to Improve Your Infrastructure as Code using Terraform

Variable Substitution using CLI and .tfvars

In the previous example, we relied on the default values of the variables. However, variables are generally used to substitute values during runtime. The default values can be overridden in two ways —

- Passing the values in CLI as `-var` argument.
- Using .tfvars file to set variable values explicitly.

If we want to initialize the variables using the CLI argument, we can do so as below. Running this command results in Terraform using these values instead of defaults

```
terraform plan -var "ami=test" -var "type=t2.nano" -var "tags={\"name\":\"My Virtual M
```


d be used for every
plex data type with

ation. Passing the
vars files come into

Create a file with the `.tfvars` extension and add the below content to it. I have used the name `values.tfvars` as the file name. This way we can organize and manage variable values easily.

```
ami  = "ami-0d26eb3972b7f8c96"
type = "t2.nano"
tags = {
 "name" : "My Virtual Machine"
 "env" : "Dev"
}
```

This time, we should ask Terraform to use the `values.tfvars` file by providing its path to `-var-file` CLI argument. The final `plan` command should look as such:

```
terraform plan -var-file values.tfvars
```

The `-var-file` argument is great if you have multiple `.tfvars` files with variations in values. However, if you do not wish to provide the file path every time you run `plan` or `apply`, simply name the file as `<filename>.auto.tfvars`. This file is then automatically chosen to supply input variable values.

onment variables. To

ble name>.

corresponding value.

```
export TF_VAR_ami=ami-0d26eb3972b7f8c96
```

Apart from the above environment variable, it is important to note that Terraform also uses a few other environment variables like `TF_LOG`, `TF_CLI_ARGS`, `TF_DATA_DIR`, etc. These environment variables are used for various purposes like logging, setting default behavior with respect to workspaces, CLI arguments, etc.

Precedence

As we have seen till now, there are three ways of providing input values to Terraform configuration using variables. Namely—default values, CLI arguments, and .tfvars file. The **precedence is given to values passed via CLI arguments.** This is followed by values passed using the `.tfvars` file and lastly, the default values are considered.

In the current example, now that we have the `values.tfvars` file saved, try to run a `plan` command by passing values via CLI `-var` arguments. Make sure to provide different values as that of .tfvars and defaults. Terraform ignores the values provided via .tfvars and defaults.

If the values are not provided in the .tfvars file, or as defaults, or as CLI arguments, it falls

d above, Terraform
ands are run.

information in
environment

This is where Spacelift shines. It makes use of these Terraform native environment variables to manage secrets as well as other attributes that make the most sense. Managing the environment in the Spacelift console is easy, thanks to a dedicated tab where values can be edited on the go.

Output Variables

For situations where you deploy a large web application infrastructure using Terraform, you often need certain endpoints, IP addresses, database user credentials, and so forth. This information is most useful for passing the values to modules along with other scenarios.

This information is also available in Terraform state files. But state files are large, and normally we would have to perform an intricate search for this kind of information.

Output variables in Terraform are used to display the required information in the console output after a successful application of configuration for the root module. To declare an output variable, write the following configuration block into the Terraform configuration files.

```
output "instance_id" {
 value       = aws_instance.myvm.id
 description = "AWS EC2 instance ID"
 sensitive   = false
```


e ID of the EC2

_id — this could be

to use its `id` attribute.

Optionally, we can use the `description` and `sensitive` flags. We have discussed the purpose of these attributes in previous sections. When a `plan` command is run, the plan output acknowledges the output variable being declared as below.

```
Changes to Outputs:
  + instance_id = (sensitive value)
```

Similarly, when we run the `apply` command, upon successful creation of EC2 instance, we would know the instance ID of the same. Once the deployment is successful, output variables can also be accessed using the `output` command:

```
terraform output
```

Output:

```
instance_id = "i-xxxxxxxx"
```


the root module.
eated by the child
module, output
child module.

Using Variables in for_each loop

This section goes through an example where we want to **create multiple subnets** for a given VPC. Without variables, we would write the configurations for the number of subnets specified. The other way is to use the Terraform `for` loop and **create separate subnets** based on the index value of the iteration.

The *for loop* thus implemented is not very useful. Consider a case where subnets need to be in selected availability zones, and each of them has a different CIDR range specified. For situations like these, we can make use of complex variable type `map(object())` along with `for` loop to iterate over this variable.

Below is the input variable declaration we need to define attributes of multiple subnets. This block specifies the `type` of variable `my_subnets` to be `map(object())`. It means we want to define a variable that has a map of objects, with a couple of attributes in each object. We have used `cidr` and `az` attributes for each object with string type.

```
variable "my_subnets" {
 type = map(object({
   cidr = string
   az   = string
 }))
```


. The `.tfvars` file

```
    a  = {
      cidr = "10.0.1.0/26"
      az   = "eu-central-1a"
    },
    "b" = {
      cidr = "10.0.2.0/26"
      az   = "eu-central-1a"
    },
    "c" = {
      cidr = "10.0.3.0/26"
      az   = "eu-central-1b"
    },
    "d" = {
      cidr = "10.0.4.0/26"
      az   = "eu-central-1c"
    },
    "e" = {
      cidr = "10.0.5.0/26"
      az   = "eu-central-1b"
    }
  }
```

If you compare the input variable declaration and initialization, you will see we have aligned the attributes. Also, the initialized value consists of 6 objects mapped by a string key. Each object has a unique CIDR and Availability Zone (az) value.

Lastly, let's define the configuration for the subnets themselves. Note: The following code

figuration.

```
    Name = "Subnet - ${each.value.az}"
  }
}
```

In this resource block, we have used the `my_subnets` variable to iterate over in a `for_each` loop. `for_each` loop comes along with a keyword `each`, which helps us identify the value to be assigned in each iteration. This single block of code is capable of creating 6 unique subnets.

Read more about using Terraform for_each meta-argument.

Limitations

It is important to note that Terraform does not allow the usage of variables in provider configuration blocks. This is mainly to adhere to best practices of using Terraform.

Variables usually make the developer's life easier by **improving the maintainability of your code**. Especially in larger Terraform codebases, variables should be used. Larger codebases are often worked upon by a team of developers. Variables make it very easy to read and modify certain aspects, like infrastructure changes via IaC.

Working in a team of Terraform developers can be challenging, especially when it comes to

Written by

loud and DevOps
nd TOGAF® 9. He
ains a blog at

Read also

GENERAL

13 min read

Who Is DevOps? Is Becoming a DevOps Engineer Worth It?

ANSIBLE

10 min read

Terraform vs. Ansible : Key Differences and Comparison of Tools

Product

Documentation

How it works

Spacelift Tutorial

Pricing

Customer Case Studies

Integrations

Security

System Status

Product Updates

Company

About Us

Careers

Contact Sales

Partners

Learn

Blog

Atlantis Alternative

Terraform Cloud Alternative

Terraform Enterprise Alternative

Spacelift for AWS

Terraform Automation

Get our newsletter

Subscribe

Terraform CLI Commands Cheatsheet

Initialize/ plan/ apply your IaC, manage modules, state, and more.