Infrastructure can be developed in code using Hashicorp Configuration Language (HCL), which Terraform then interprets for provisioning purposes. There are a lot of advantages to using Infrastructure as Code (IaC) - consistency, low risk, disaster recovery - to name but a few.

Nonetheless, as physical infrastructure grows, the challenge of managing it via Terraform IaC also grows. State files, which hold the mapping between the real-world infrastructure and the configurations and resources defined by Terraform, become even more critical and even though infrastructure is managed as “code”, it still differs from conventional programming.

To tackle the complexities involved in large Terraform codebases, Terraform provides built-in functions that enable developers to produce more maintainable code. This post aims to shed light on some of the more common Terraform functions.

**Terraform Function Overview**

As of version 1.1.9, functions have been categorized into types. Before we move on, let’s summarize them.

|  |  |
| --- | --- |
| **Function type** | **Description** |
| [String](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.9tayt2bc3mea) | functions to perform string manipulation |
| [Collection](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.y26mvwawao9o) | functions to manipulate collections - arrays, objects, lists, and maps |
| [Encoding](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.jis1zycsxisk) | encoding and decoding functions that deal with various formats - base64, text, JSON, YAML, etc |
| [File System](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.n42k544k2uhh) | functions to perform essential file operations |
| [Numeric](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.w1gvkex5rz7n) | functions to perform mathematical operations on numeric values |
| [Date and Time](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.ipt02qh37y6o) | functions for date and time manipulation |
| [Hash and Crypto](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.4prjyy6ecf4d) | functions that work with hashing and cryptographic mechanisms |
| [IP Network](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.sx8tbnyv2uic) | functions to work with CIDR ranges |
| [Type Conversion](https://docs.google.com/document/d/1Q_rQ2zX3LmwM5id_IxW1xleKrcQlANquC6X0HkKk7wk/edit#heading=h.mety4wse473d) | functions to convert data types |

We will look at examples relevant to each category in the upcoming sections.

**About Terraform Functions**

Before we continue, it might be worthwhile highlighting a few things about Terraform functions that will help us later on.

Terraform functions are used in the expression of an argument, and return a value of a specified type. The built-in functions can be generalized using the syntax below:

|  |
| --- |
| <function\_name>(arg 1, arg 2). |

The number and type of arguments accepted by Terraform functions are predefined. In some instances however, the built-in functions may accept undefined arguments - for example, min() and max() functions of the numeric type.

Inputs to these arguments are of the same type. Instead of passing a long list of arguments by value, we use a **type list** or **tuple variable**. The syntax for the function call will now look as below.

|  |
| --- |
| <function\_name>(<list/tuple variable> ...) |

The three dots “...” syntax above is known as **expansion syntax** and is only available inside functions.

Terraform functions handle sensitive data conservatively. When you pass an object containing sensitive information to the function, Terraform marks the output of the entire function as sensitive.

It is impossible for users to create custom functions that add further processing logic beyond what is already baked-in. Terraform is a provisioning tool and not a logic processing compiler/interpreter. The built-in functions represent  all available logic, which engineers must leverage creatively.

**Exploring Terraform functions**

Terraform functions can be used to address very complex requirements and their capabilities are limited only by your imagination. Furthermore, these functions can perform fundamental tasks that would take considerable effort if using HCL alone. Let’s take a look at some of the most commonly used Terraform functions:

**String**

String manipulation is an onion fundamental to almost all forms of programming. You can use string functions to extract and format information from a source, so that it is suitable for the next step in processing.

When provisioning cloud resources, cloud providers typically impose formatting restrictions in their APIs for certain attributes. Accordingly, Terraform string functions can be used to create attribute values.

As well as this, . Terraform string functions offer the ability to trim, slice, replace, apply regex and manage case sensitivities.

In the example below, we use the **join function** to convert a given list of characters into a single string. Join functions accept a separator as the first argument and a list as the second.

|  |
| --- |
| variable "chars" {  type = list(string)  default     = ["HELLO", "WORLD,", "LET'S", "DO", "THIS!"]  description = "Characters" } |

|  |
| --- |
| > join(" ", var.chars) "HELLO WORLD, LET'S DO THIS!" |

We can also use Terraform functions in sequence - where the return value/output of one function can be the input to another. This is similar to the pipe operator often used in a shell script.

In the example below we will get the sub-string and convert it to lower case.

|  |
| --- |
| lower(**substr**(**join**(" ", var.chars), 13, 26)) "let's do this!" |

**Collection**

A good deal of Terraform code, written to perform repetitive actions, makes use of collections. **Lists**, **Tuples**, **Maps**, and **Objects** can all store information in a way that, when used together, provide desired configuration values.

Since collections repetitively store values, it is easy to run loops through them. Within Terraform, we use the **splat** operator to identify the collection object in each iteration.

We can use collection functions on data to sort, slice, fetch and validate keys and corresponding values. We can also perform operations similar to string operations such as concat, length, coalesce and so on.

In the example below we receive some data that looks like a list. However, it has irregularities in its nesting, which makes it impossible for us to predict the pattern of the text.

In this instance, we can use the **flatten function** to remove the nesting and create a new list containing the individual elements.

|  |
| --- |
| > flatten([["banana", "grape", ["apple", "orange"]], ["banana"], [], ["guava"]]) [   "banana",   "grape",   "apple",   "orange",   "banana",   "guava", ] |

In the resulting list, “banana” appears twice. To ensure this list contains only unique items, we can use another collection function - **distinct** - to remove duplicates. See below.

|  |
| --- |
| > distinct(flatten([["banana", "grape", ["apple", "orange"]], ["banana"], [], ["guava"]])) tolist([   "banana",   "grape",   "apple",   "orange",   "guava", ]) |

**Encoding**

Although it’s not recommended to encode or decode large volumes of data, encoding and decoding functions can convert strings into different formats. For example, Base64 is a standard format typically used to encode text.

Terraform functions of this type help encode and decode characters using formats like base64, JSON, YAML, CSV, URL and so forth.

In the example below, we encode a string of Unicode characters into the “ISO\_8859-1:1987” character set. You can find a whole list of character sets [here](https://www.iana.org/assignments/character-sets/character-sets.xhtml).

|  |
| --- |
| > textencodebase64("Hello World", "ISO\_8859-1:1987") "SGVsbG8gV29ybGQ=" |

Similarly, we can decode by using the corresponding decoding function. For the sake of this example, we will use it in sequence and expect the output to be the same as the fororiginal input.

|  |
| --- |
| > textdecodebase64(**textencodebase64**("Hello World", "ISO\_8859-1:1987"), ,) "Hello World" |

Using “UTF-8” as the character set above, is the same as using a dedicated base64encode function.

**File System**

Configuring and setting up applications, web servers, network configs and so on are tasks commonly performed by Terraform after object deployment. We may need to read a file from a source path and modify this to replace variable values before pushing them to a newly provisioned VM’s target path. This requires good file manipulation (read/write) functionality and we have this via Terraform Filesystem functions.

The example below demonstrates how a **templatefile** **function** can be used to create an NGINX config file on a target EC2 instance.

File and templatefile functions are the most used functions when handling filesystems. These functions read files from the Terraform host machine and data from files specified by a path. It takes a single argument - the path to the file, as we can see in the example below. If the complete path is not specified, it looks for the file in the current directory.

|  |
| --- |
| > file("file.txt") "Hello World!" |

The **templatefile function** can also handle template files (.tftpl), which could contain additional logic used to create output via templated content. Note the template file (file.tftpl) below - it is a simple shell script that creates a text file within a folder, before adding an IP address into the file.

|  |
| --- |
| #!/bin/sh sudo mkdir ${request\_id} cd ${request\_id} sudo touch ${name}.txt echo ${IP} >> ${name}.txt |

In the example we use Request ID, Name, and IP address as variables that will be replaced dynamically. The templatefile() function used in the terraform code below, implements the first argument as a file path. It then accepts an object as a second argument, allowing us to provide values as JSON key-value pairs in the template file.

|  |
| --- |
| resource "aws\_instance" "my\_vm" {  ami           = var.ami  instance\_type = var.type  user\_data     = templatefile("file.tftpl", { IP = "x.x.x.x", name = "Web Server", request\_id = "REQ001232" })  key\_name      = "functiontpl"   tags = {    name = "My VM"    type = "Function experiment"  } } |

**Numeric**

Most numeric functions take numbers (integers and floats) as their input arguments. The exception to this is the **parseint function**, which accepts a string argument and

parses the numeric value from it, for a given base.

|  |
| --- |
| > parseint("AD", 16) 173 |

The numeric function can be used to identify the minimum, maximum, nearest whole number, signature, exponents, and absolute value.

Terraform modules accept a set of inputs - these could be provided by an application that triggers cloud provisioning. In most cases, the parent module carries the inputs to the child module.

The supplied information may therefore generate multiple choices, in various formats. In cases where this is expected, you can leverage Terraform numeric functions to calculate the correct number. You can also leverage these functions to handle exceptions resulting from unexpected inputs.

The example below shows how we can use the **max function** to determine the number of virtual machines that should be created, based on the input received from a variable.

|  |
| --- |
| variable "no\_of\_vms" {  type = list(number)  default     = [2, 3, 4, 5, 6, 7, 8]  description = "Region" } resource "aws\_instance" "demo\_vm" {  ami                     = var.ami  instance\_type           = var.type  count                   = max(var.no\_of\_vms ...) } |

Notice how we have leveraged **expansion syntax** in the above example. If we run the plan command now, observe how Terraform correctly plans to create 8 EC2 instances.

|  |
| --- |
| + root\_block\_device {           + delete\_on\_termination = (known after apply)           + device\_name           = (known after apply)           + encrypted             = (known after apply)           + iops                  = (known after apply)           + kms\_key\_id            = (known after apply)           + tags                  = (known after apply)           + throughput            = (known after apply)           + volume\_id             = (known after apply)           + volume\_size           = (known after apply)           + volume\_type           = (known after apply)         }     }  Plan: 8 to add, 0 to change, 0 to destroy. |

**Date and Time**

Like strings, dates and times may also require manipulation or alteration. Terraform currently supports three Date and Time functions, listed below, that can be utilized to do this. In our demonstration below, we use these functions to obtain the current timestamp, increase the time and then reformat it.

|  |
| --- |
| > timestamp() "2022-04-29T12:29:56Z"  > timeadd(**timestamp**(), "1.5h") "2022-04-29T14:00:50Z"   > formatdate("E"1.5h")) "Friday, 29-Apr-22 14:01:38 UTC" |

**Hash and Crypto**

Terraform is already good at masking sensitive data, but hash and cryptographic functions can be used to anonymize data even further. Testing the integrity of files used by Terraform for operations is also a crucial feature that you can implement via hashing functions.

A range of hashing functions exists for creating hash values from text and files. In the example below we hash a string using SHA256.

|  |
| --- |
| > base64sha256("testing terraform functions") "dVVvXLTTrrykK2zjWOd0DbSprXWXbX4j+toeTB/C91E=" |

Similarly, the **filesha256 function** takes a  file path as an argument to encrypt its content.

|  |
| --- |
| > filesha256("file.txt") "75556f5cb4d3aebca42b6ce358e7740db4a9ad75976d7e23fada1e4c1fc2f751" |

**IP Network**

Working with virtual networks, subnets, route tables and other aspects of cloud networking typically involves handling IP addresses. For a given range, determining the available IP addresses requires some calculation. IP Network-related functions can help in this regard, where the supplied IP address, network prefix, CIDRs and so on are dynamic.

For example, if you know the network prefix, you can find the IP address of the nth host using the function **cidrhost,** as demonstrated below. Here we find the IP address of the 7th host.

|  |
| --- |
| > cidrhost("10.12.112.0/20", 7) "10.12.112.7" |

A problem with using this function though, is that it does not throw an error if the requested host number is outside of the provided CIDR range.

**Type Conversion**

As the name suggests, type conversion functions convert the data type of the variable. This can help structure data consumed or produced by our Terraform code.

When we want to unmask and read information that is marked or declared sensitive, use the **nonsensitive** **function**.

|  |
| --- |
| > var.data1 (sensitive) > nonsensitive(var.data1) tolist([   tolist([     "banana",     "grape",   ]),   tolist([     "banana",   ]),   tolist([]),   tolist([     "guava",   ]), ]) |

Similarly, to convert any non-sensitive data to sensitive, use the **sensitive** **function** as below.

|  |
| --- |
| > var.data1 tolist([   tolist([     "banana",     "grape",   ]),   tolist([     "banana",   ]),   tolist([]),   tolist([     "guava",   ]), ]) > sensitive(var.data1) (sensitive) |

**Conclusion**

Since Terraform is a declarative configuration language, rather than a fully-fledged programming language, it can sometimes be challenging to perform some of the more basic programming tasks. But when used correctly, Terraform functions can be used to apply dynamic logic that helps create more maintainable and repeatable infrastructure code.

Developers often miss these basic functions, as they understandably gravitate more towards the core tasks of developing IaC and managing state files. It’s a shame though, because with a little bit of practice and study (there is a wealth of documentation),  they can be used to reduce configuration effort and to address some quite complex issues.