Lab: Terraform Built-in Functions

Duration: 15 minutes

As you continue to work with data inside of Terraform, there might be times where you want to modify or manipulate data based on your needs. For example, you may have multiple lists or strings that you want to combine. In contrast, you might have a list where you want to extract data, such as returning the first two values.

The Terraform language has many built-in functions that can be used in expressions to transform and combine values. Functions are available for actions on numeric values, string values, collections, date and time, filesystem, and many others.

- Task 1: Use basic numerical functions to select data
- Task 2: Manipulate strings using Terraform functions
- Task 3: View the use of cidrsubnet function to create subnets

Task 1: Use basic numerical functions to select data

Terraform includes many different functions that work directly with numbers. To learn how they work, let's add some code and check it out. In your `variables.tf` file, let's add some variables. Feel free to use any number as the default value.

```
variable "num_1" {
  type = number
  description = "Numbers for function labs"
  default = 88
}

variable "num_2" {
  type = number
  description = "Numbers for function labs"
  default = 73
}

variable "num_3" {
  type = number
  description = "Numbers for function labs"
  default = 52
}
```

In the `main.tf`, let's add a new local variable that uses a numercial function:

```
locals {
  maximum = max(var.num_1, var.num_2, var.num_3)
  minimum = min(var.num_1, var.num_2, var.num_3, 44, 20)
}

output "max_value" {
  value = local.maximum
}

output "min_value" {
  value = local.minimum
}
```

Go ahead and run a `terraform apply -auto-approve` so we can see the result of our numerical functions by way of outputs.

Task 2: Manipulate strings using Terraform functions

Now that we know how to use functions with numbers, let's play around with strings. Many of the resources we deploy with Terraform and the related aruments require a string for input, such as a subnet ID, security group, or instance size.

Let's modify our VPC to make use of a string function. Update your VPC resource in the `main.tf` file to look something like this:

```
#Define the VPC
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name        = upper(var.vpc_name)
    Environment = upper(var.environment)
    Terraform   = upper("true")
  }

  enable_dns_hostnames = true
}
```

Go ahead and run a `terraform apply -auto-approve` so we can see the result of our string functions by way of the changes that are applied to our tags.

Task 2.1

Now, let's assume that we have set standards for our tags across AWS, and one of the requirements is that all tags are lower case. Rather than bothering our users with variable validations, we can simply take care of it for them with a simply function.

In your `main.tf` file, update the locals block to the following:

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Name      = lower(local.server_name)
    Owner     = lower(local.team)
    App       = lower(local.application)
    Service   = lower(local.service_name)
    AppTeam   = lower(local.app_team)
    CreatedBy = lower(local.createdby)
  }
}
```

Before we test it out, let's set the value of a variable using a `.tfvars` file. Create a new file called `terraform.auto.tfvars` in the same working directory and add the following:

```
environment = "PROD_Environment"
```

Let's test it out. Run `terraform plan` and let's take a look at the proposed changes. Notice that our string manipulations are causing some of the resource tags to be updated.

Go and apply the changes using `terraform apply -auto-approve`.

Task 2.2

When deploying workloads in Terraform, it's common practice to use functions or expressions to dynamically generate name and tag values based on input variables. This makes your modules reuseable without worrying about providing values for more and more tags.

In your `main.tf` file, let's update our locals block again, but this time we'll use a `join` to dynamically generate the value for Name based upon data we're already providing or getting from data blocks.

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Name = join("-",
      [local.application,
```


```
        data.aws_region.current.name,
    local.createdby])
    Owner     = lower(local.team)
    App       = lower(local.application)
    Service   = lower(local.service_name)
    AppTeam   = lower(local.app_team)
    CreatedBy = lower(local.createdby)
  }
}
```

Let's test it out. Run `terraform plan` and let's take a look at the proposed changes. Notice that our string function is dynamically creating a Name for our resource based on other data we've provided or obtained.

Go and apply the changes using `terraform apply -auto-approve`.

Task 3: View the use of cidrsubnet function to create subnets

There are many different specialized functions that come in handy when deploying resources in a public or private cloud. One of these special functions can help us automatically generate subnets based on a CIDR block that we provided it. Since the very first time you ran `terraform apply` in this course, you've been using the `cidrsubnet` function to create the subnets.

In your `main.tf` file, view the resource blocks that are creating our initial subnets:

```
#Deploy the private subnets
resource "aws_subnet" "private_subnets" {
  for_each          = var.private_subnets
  vpc_id            = aws_vpc.vpc.id
  cidr_block        = cidrsubnet(var.vpc_cidr, 8, each.value)
  availability_zone = tolist(data.aws_availability_zones.available.names)[
     each.value]

  tags = {
    Name      = each.key
    Terraform = "true"
  }
}
```

Note how this block (and the one to create the public subnets) is creating multiple subnets. The resulting subnets were created based on the initial CIDR block, which was our VPC CIDR block. The second value is the number of bits added to the original prefix, so in our case, it was `/16`, so resulting subnets will be `/24` since we're adding `8` in our function. The last value needed is derived from the

value obtained from the `value` of each key in `private_subnets` since we're using a for_each in this resource block.

Free free to create other subnets using the cidrsubnet function and play around with the values to see how it could best fit your requirements and scenarios.