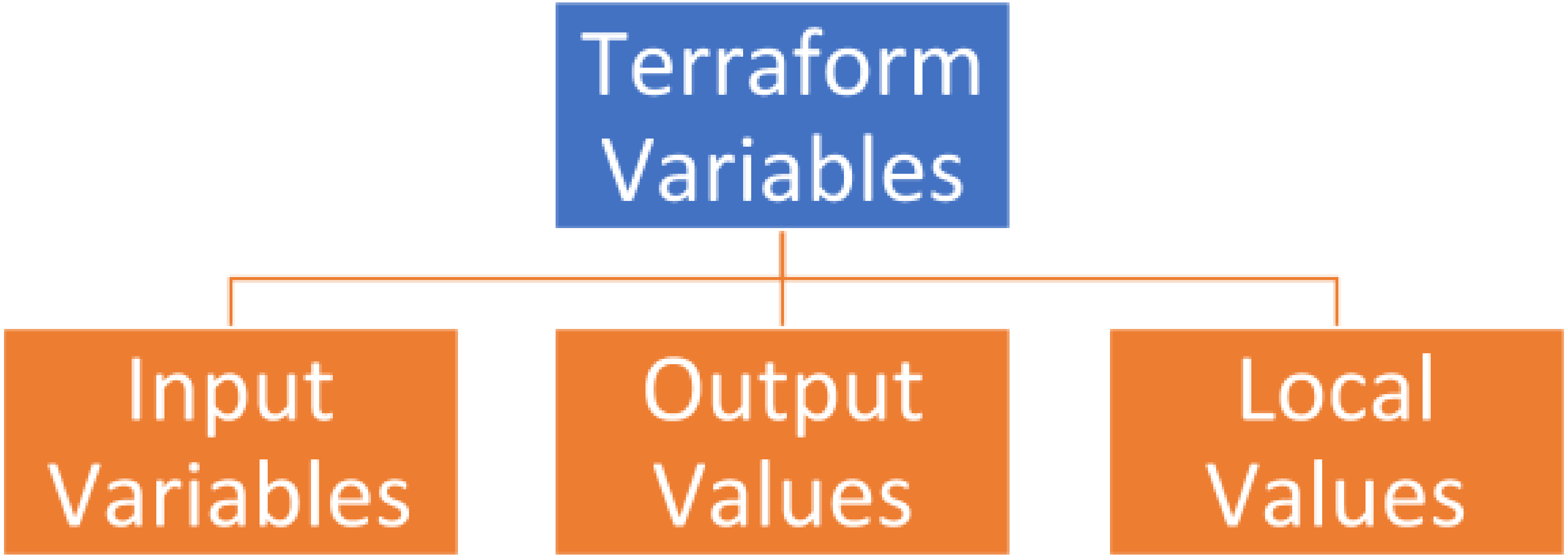
Agenda:

- Terraform Variables
- Terraform Input Variables
- Terraform Output Variables
- Terraform Local Variables
- Conditioning in Terraform
- Iterations in Terraform
- Resources Metadata
- Data Source in Terraform
- Operators in Terraform
- Interpolation in Terraform

Terraform Variables

Variable a symbolic name associated with a value and whose associated value may be changed

Variables in Terraform are a great way to define centrally controlled reusable values. The information in **Terraform variables** is saved independently from the deployment plans, which makes the values easy to read and edit from a single file.

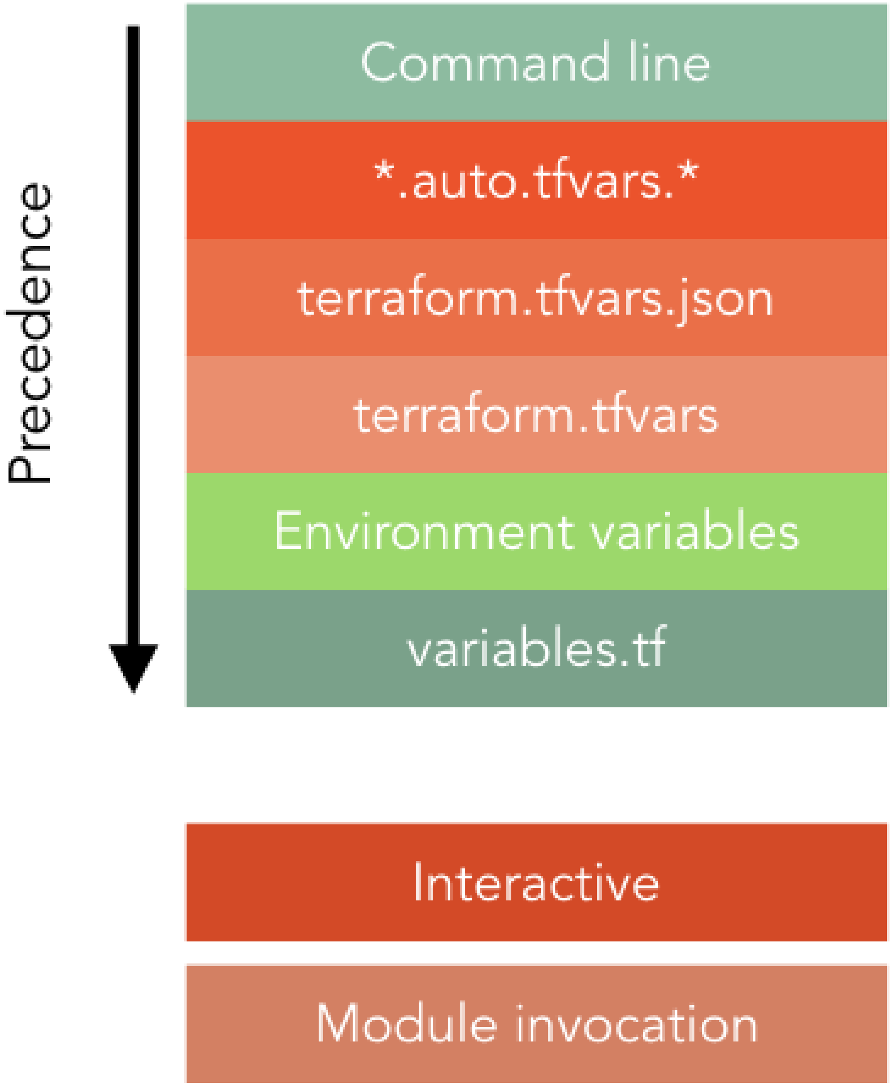


Terraform Variables

Input Variables

Output Values

Local Values



Command line

Assign with runtime flag such as `-var='my_var=value'`

.auto.tfvars.

terraform.tfvars.json

terraform.tfvars

Dedicated Terraform files just for assigning values

Environment variables

Easy to set, see, change; Use TF_VAR_<variable> pattern

variables.tf

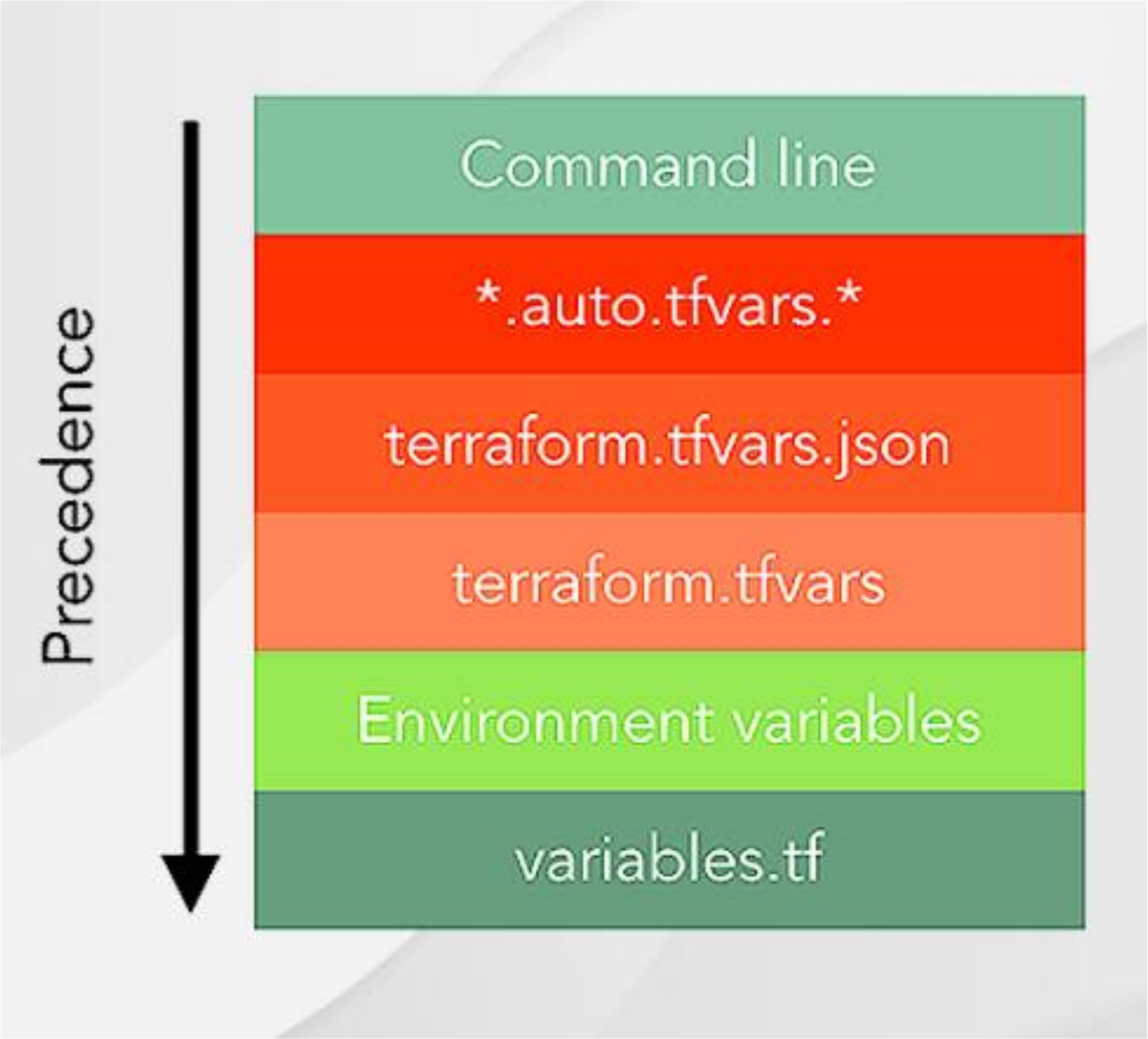
Good for providing non-sensitive values, reasonable defaults

Interactive

Terraform asks for required values not otherwise provided

Module invocation

Invoking a module provides opportunity to assign values


Precedence
Command line
.auto.tfvars.
terraform.tfvars.json
terraform.tfvars
Environment variables
variables.tf

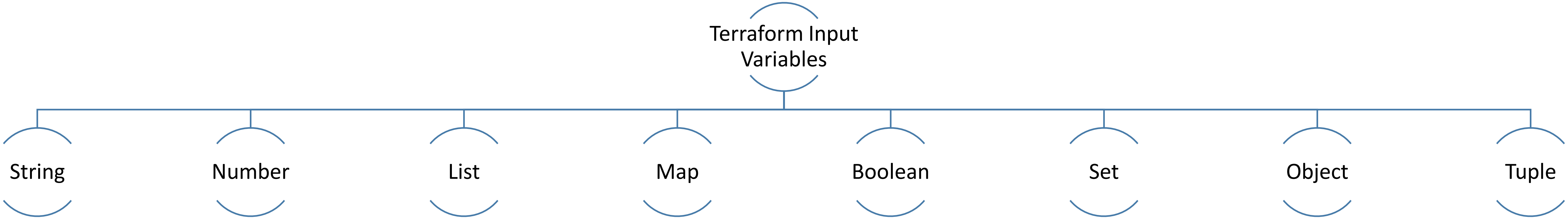
Terraform Input Variables

Terraform Input Variable

Input variables serve as parameters for a Terraform module/Terrform code, allowing aspects of the module/Terrform code to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

- When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables.
- When you declare them in child modules, the calling module should pass values in the module block.

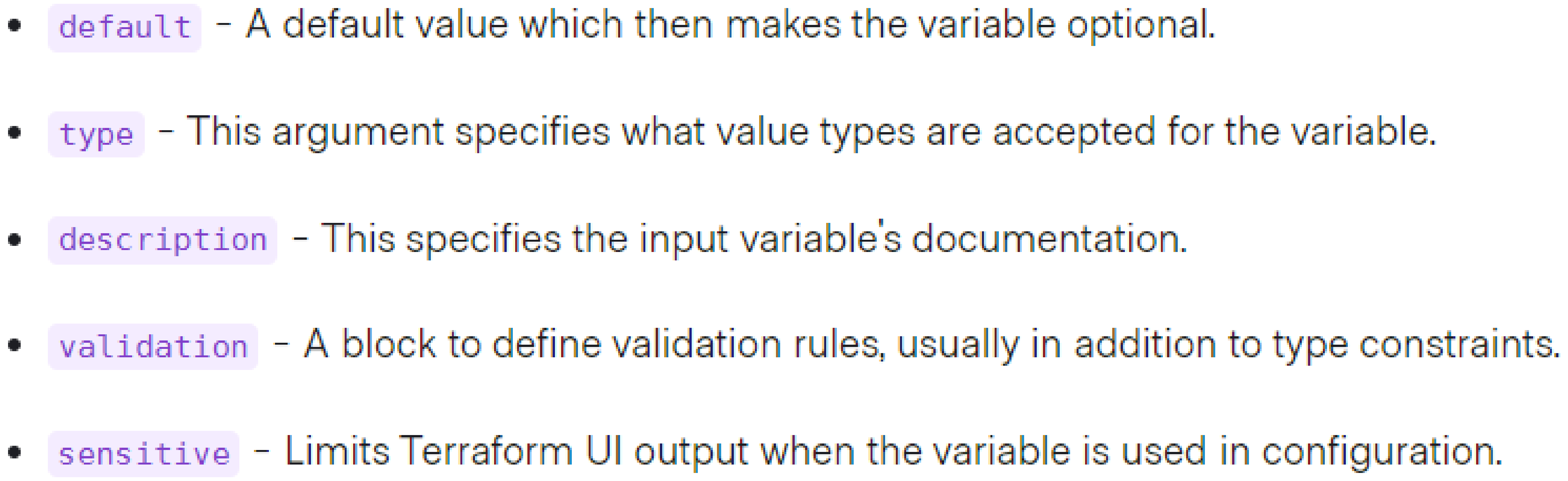
Terraform Input Variable Types



Terraform Input Variable

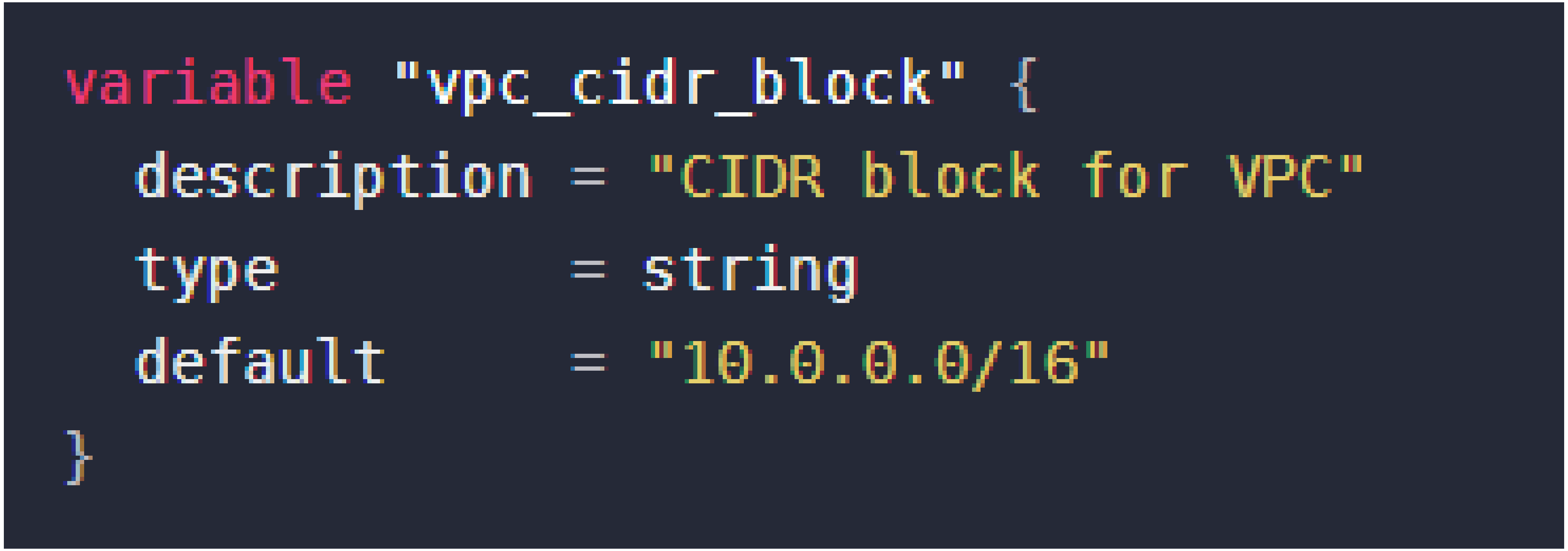
Variable blocks have 5 optional arguments.

We recommend setting a description and type for all variables, and setting a default value when practical.
If you do not set a default value for a variable, you must assign a value before Terraform can apply the configuration. Terraform does not support unassigned variables.



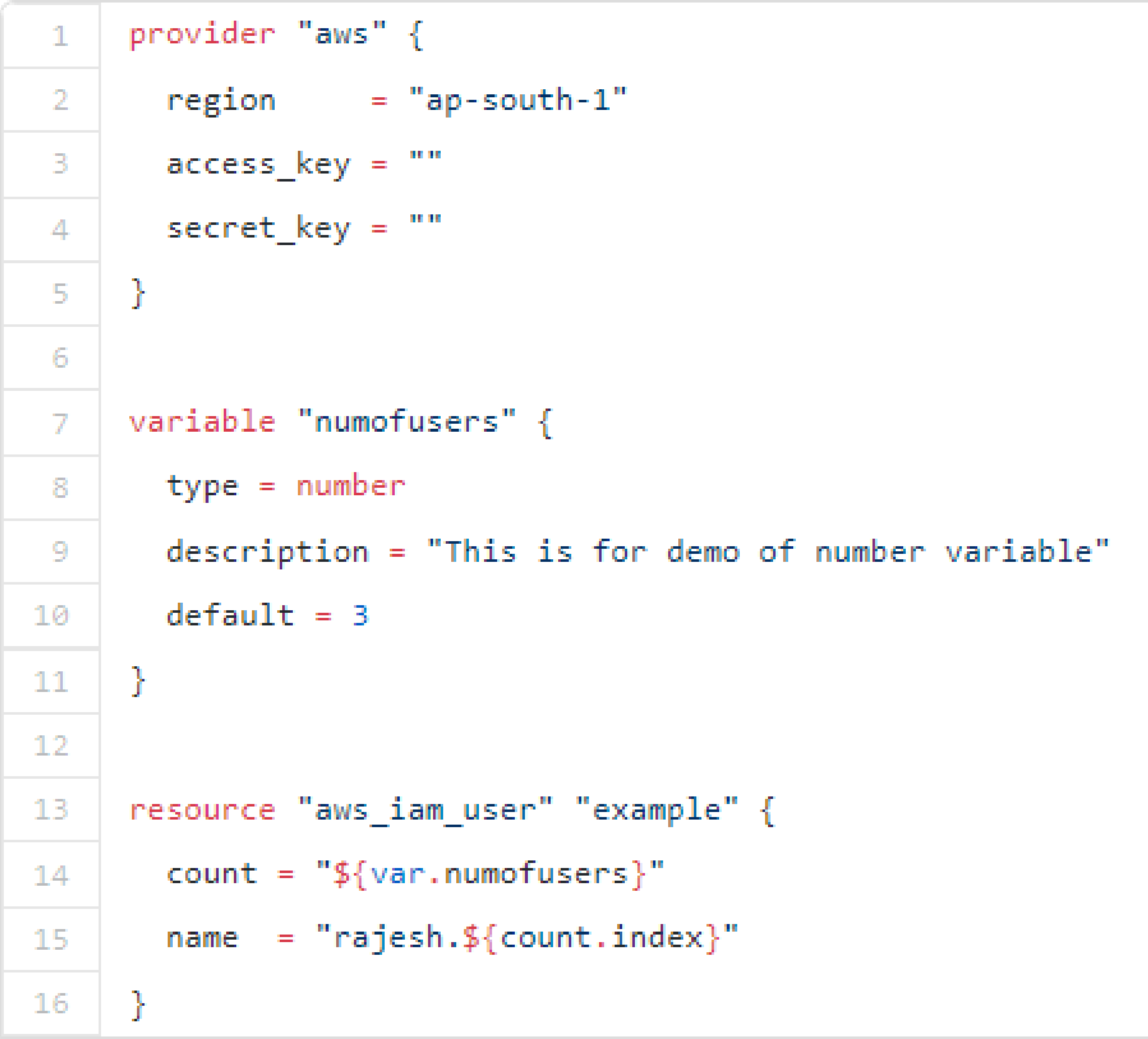
- `default` - A default value which then makes the variable optional.
- `type` - This argument specifies what value types are accepted for the variable.
- `description` - This specifies the input variable's documentation.
- `validation` - A block to define validation rules, usually in addition to type constraints.
- `sensitive` - Limits Terraform UI output when the variable is used in configuration.

How to declare?



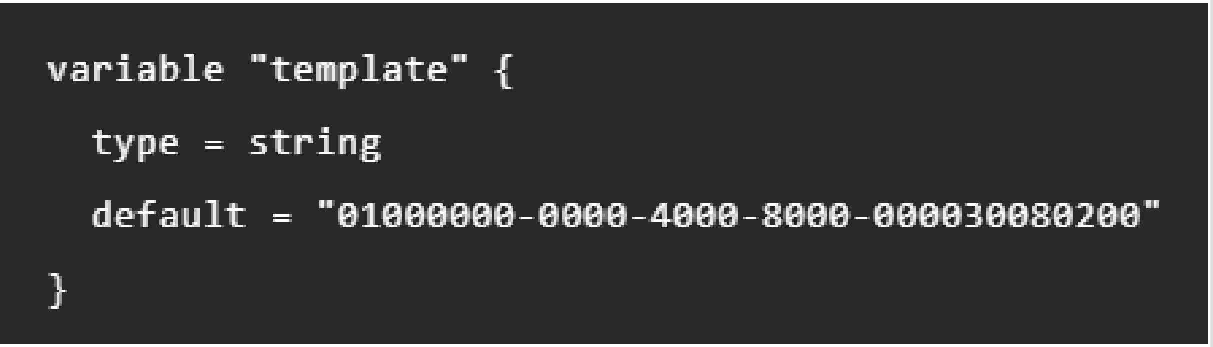
```
variable "vpc_cidr_block" {
  description = "CIDR block for VPC"
  type        = string
  default     = "10.0.0.0/16"
}
```

Terraform Variable Type: Number

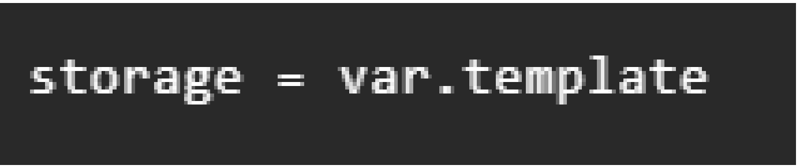


```
provider "aws" {
  region     = "ap-south-1"
  access_key = ""
  secret_key = ""
}

variable "numofusers" {
  type = number
  description = "This is for demo of number variable"
  default = 3
}

resource "aws_iam_user" "example" {
  count = "${var.numofusers}"
  name  = "rajesh.${count.index}"
}
```

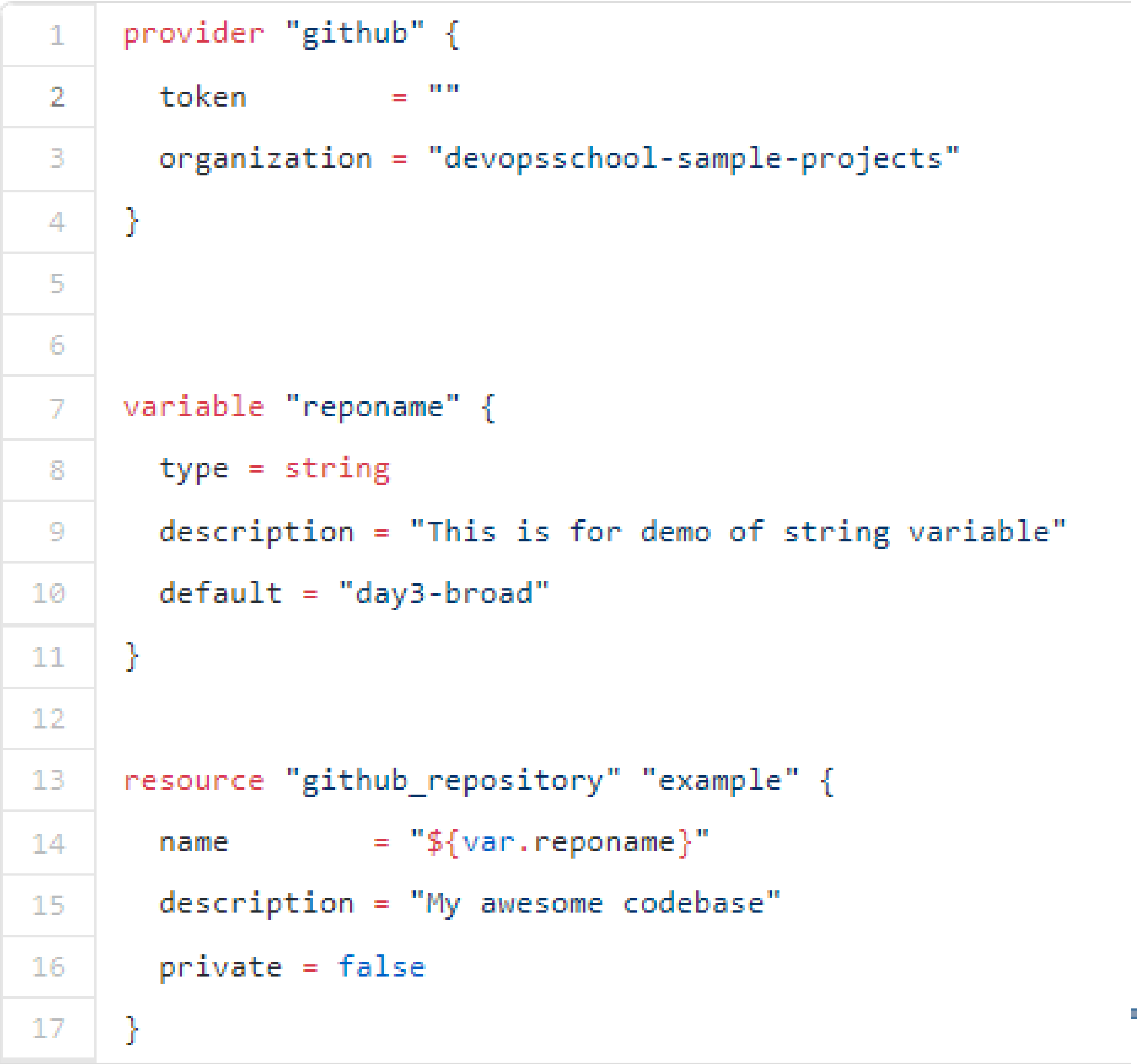
Terraform Variable Type: String



```
variable "template" {
  type = string
  default = "01000000-0000-4000-8000-000030080200"
}
```

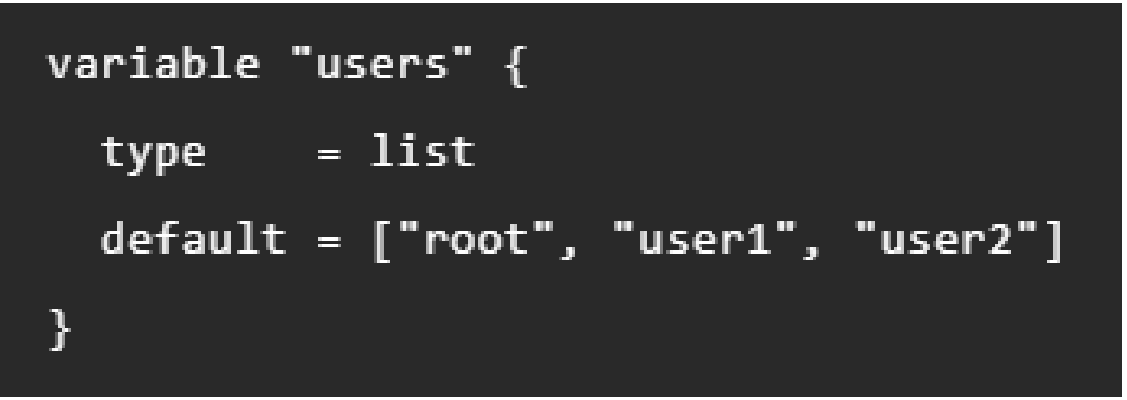


```
storage = var.template
```

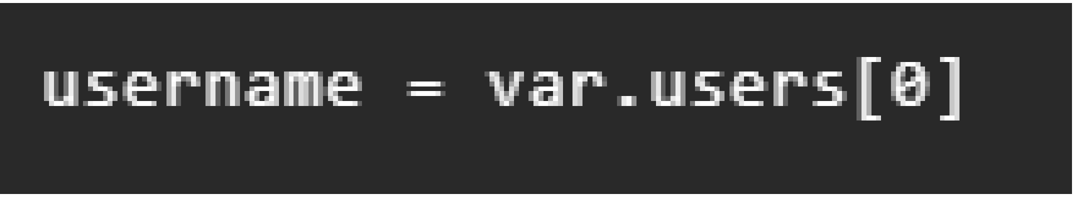


```
provider "github" {
  token        = ""
  organization = "devopsschool-sample-projects"
}


variable "reponame" {
  type = string
  description = "This is for demo of string variable"
  default = "day3-broad"
}

resource "github_repository" "example" {
  name        = "${var.reponame}"
  description = "My awesome codebase"
  private = false
}
```

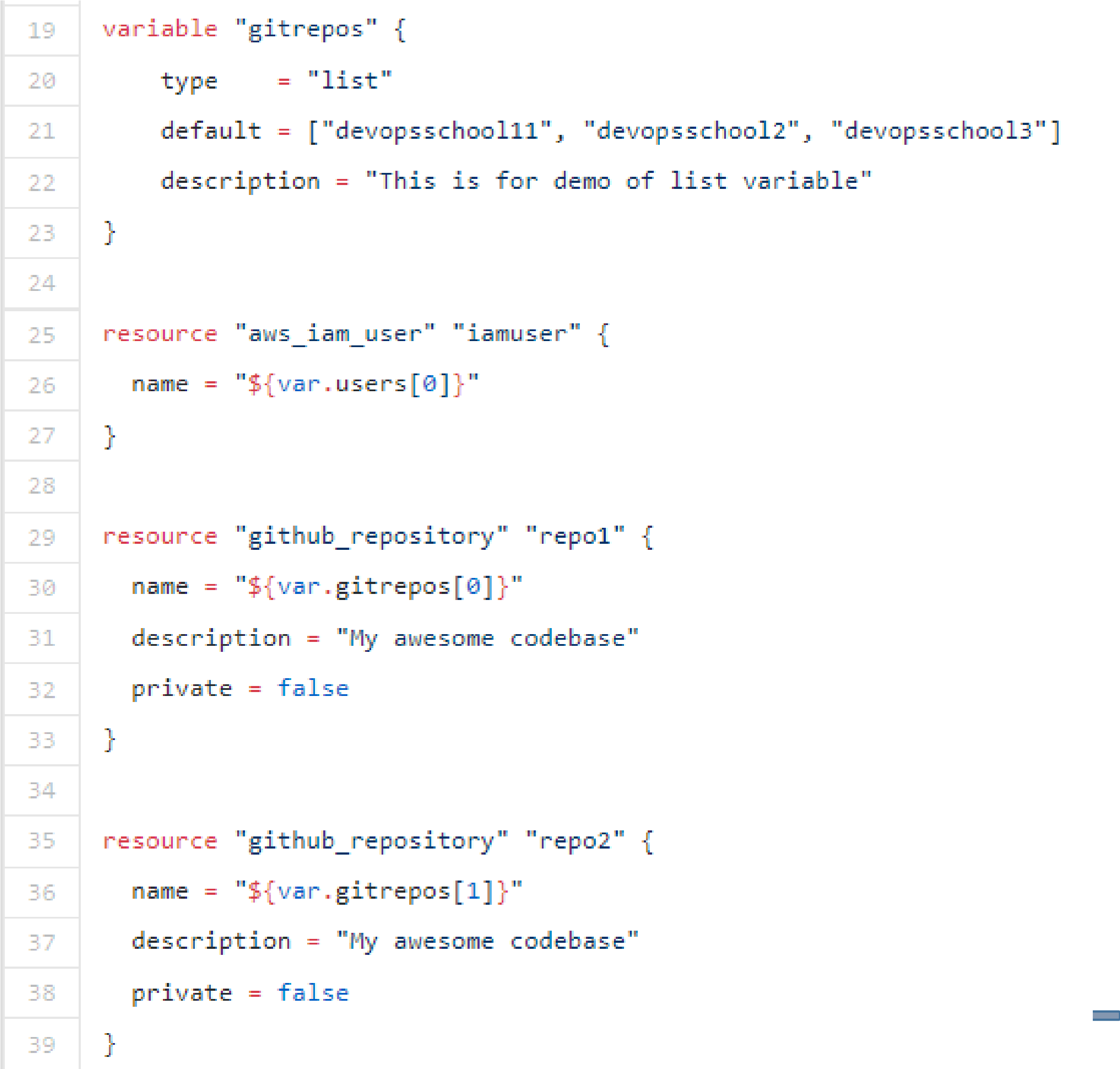
Terraform Variable Type: List



```
variable "users" {
  type     = list
  default = ["root", "user1", "user2"]
}
```

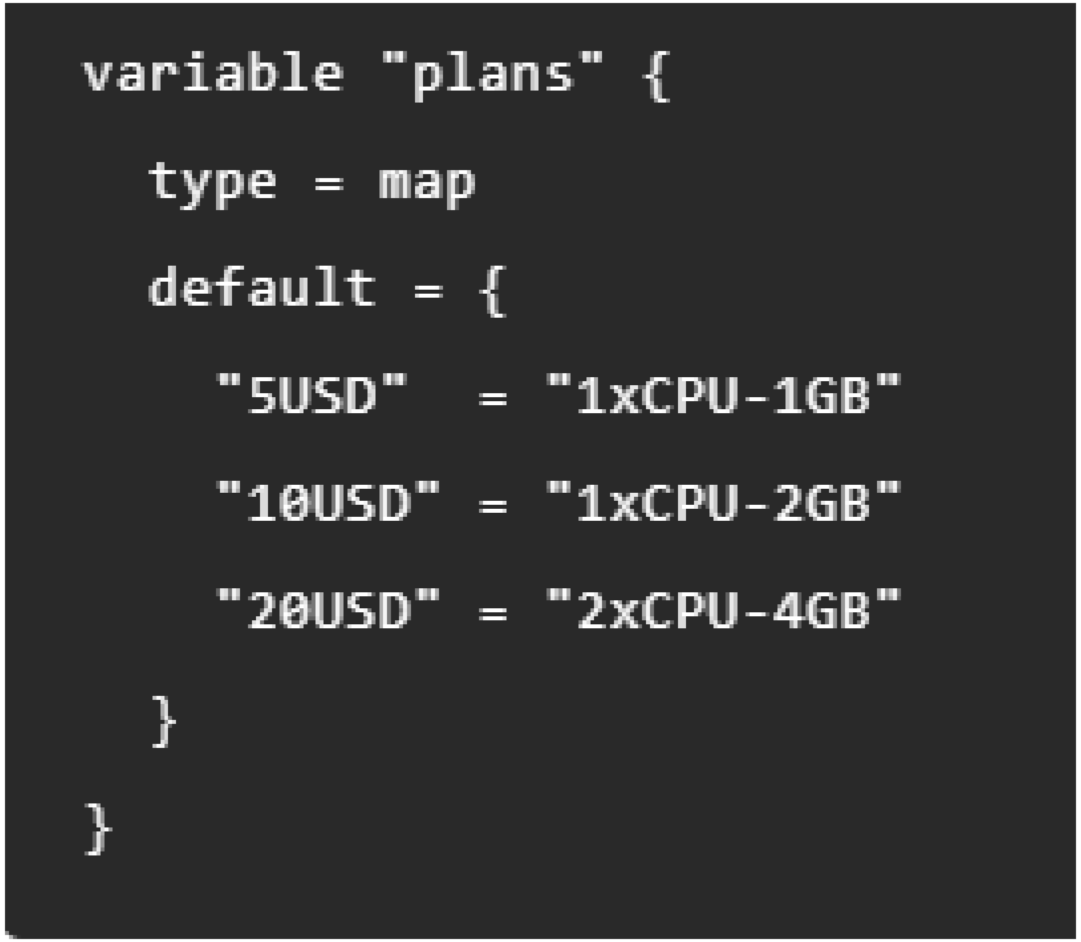


```
username = var.users[0]
```

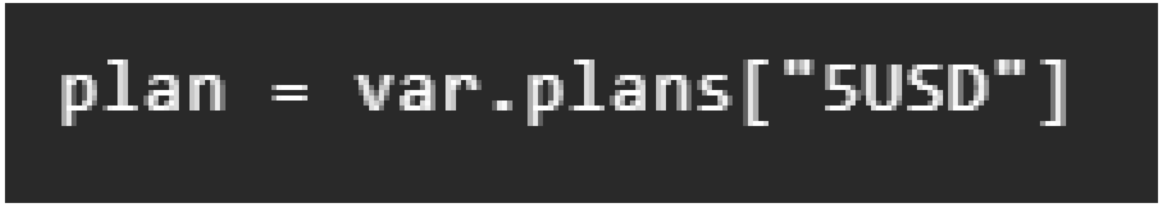


```
variable "gitrepos" {
    type     = "list"
    default = ["devopsschool11", "devopsschool2", "devopsschool3"]
    description = "This is for demo of list variable"
}

resource "aws_iam_user" "iamuser" {
  name = "${var.users[0]}"
}

resource "github_repository" "repo1" {
  name = "${var.gitrepos[0]}"
  description = "My awesome codebase"
  private = false
}

resource "github_repository" "repo2" {
  name = "${var.gitrepos[1]}"
  description = "My awesome codebase"
  private = false
}
```

Terraform Variable Type: Map

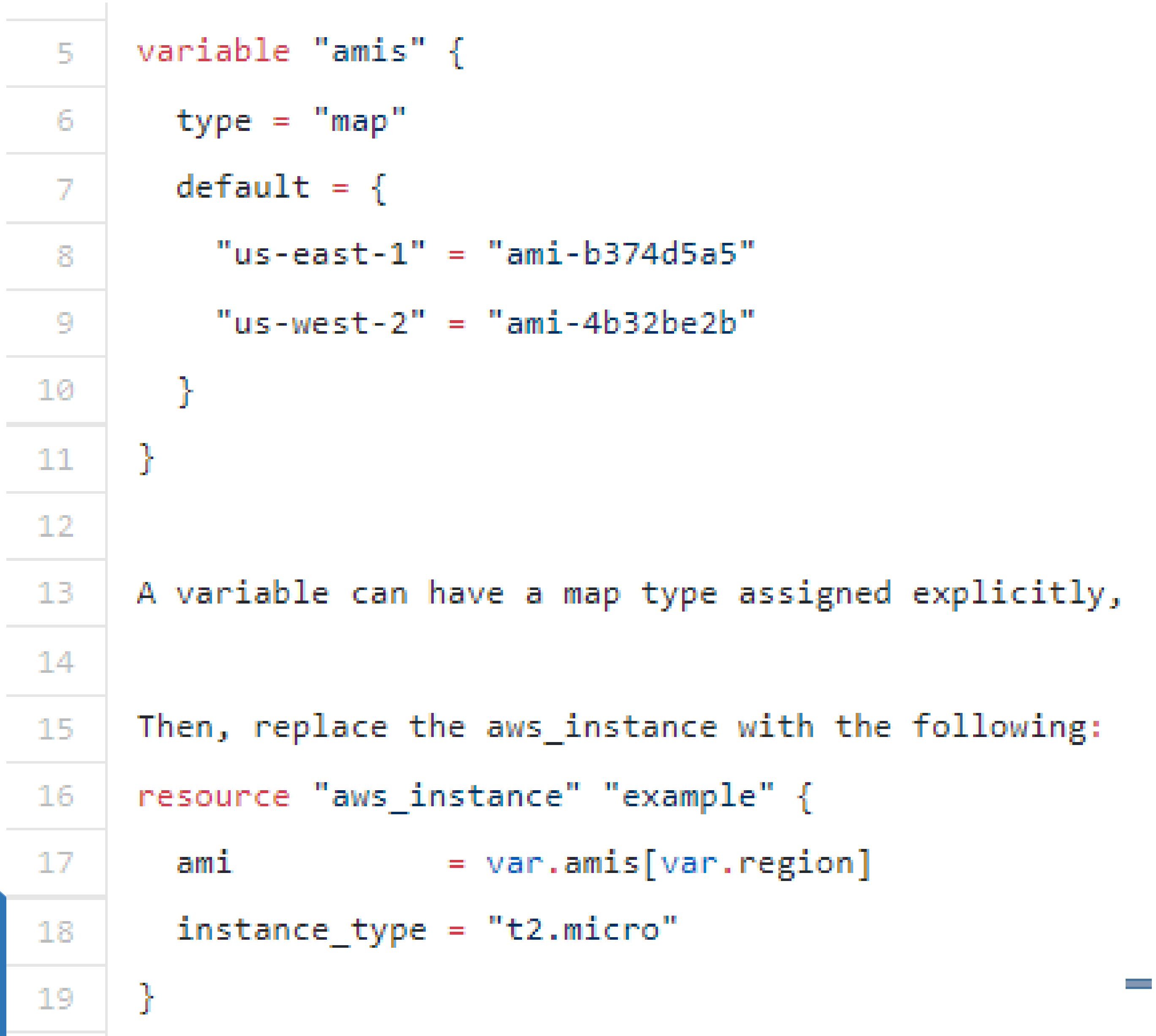


```
variable "plans" {
  type = map
  default = {
    "5USD"  = "1xCPU-1GB"
    "10USD" = "1xCPU-2GB"
    "20USD" = "2xCPU-4GB"
  }
}
```

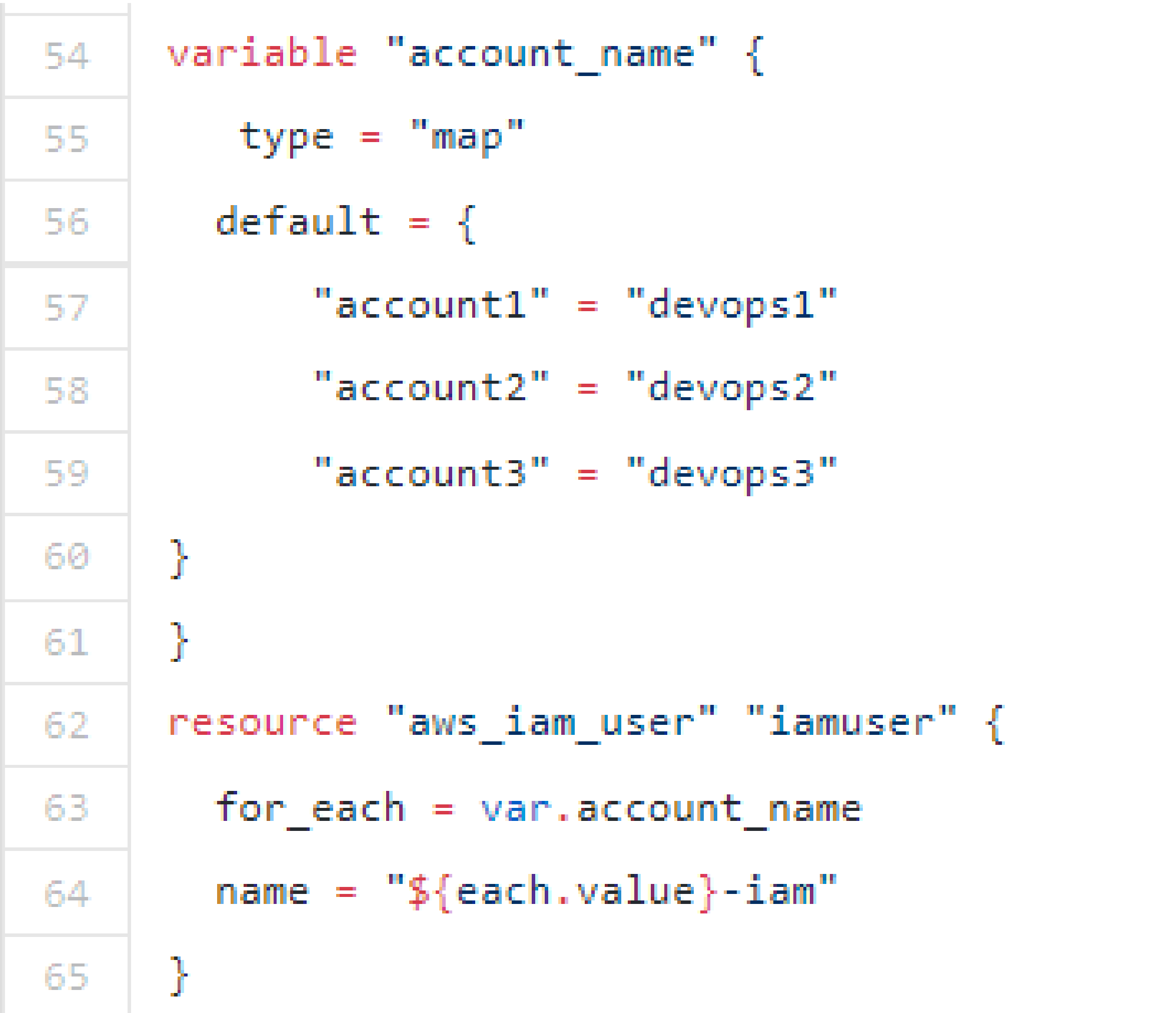


```
plan = var.plans["5USD"]
```

Terraform Variable Type: Map

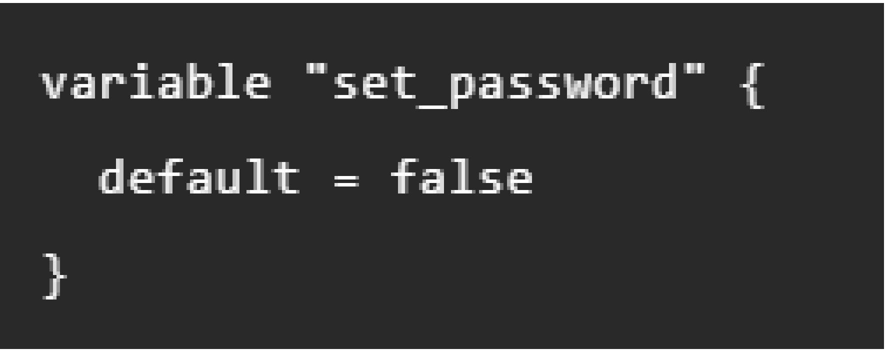


```
variable "amis" {
  type = "map"
  default = {
    "us-east-1" = "ami-b374d5a5"
    "us-west-2" = "ami-4b32be2b"
  }
}

A variable can have a map type assigned explicitly,

Then, replace the aws_instance with the following:
resource "aws_instance" "example" {
  ami           = var.amis[var.region]
  instance_type = "t2.micro"
}
```

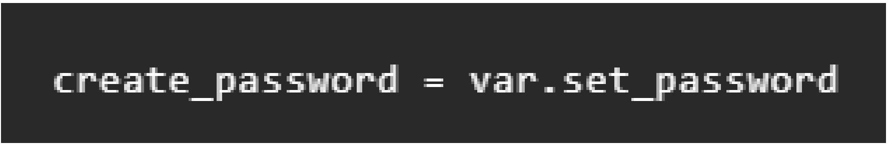


```
variable "account_name" {
   type = "map"
  default = {
      "account1" = "devops1"
      "account2" = "devops2"
      "account3" = "devops3"
}
}
resource "aws_iam_user" "iamuser" {
  for_each = var.account_name
  name = "${each.value}-iam"
}
```

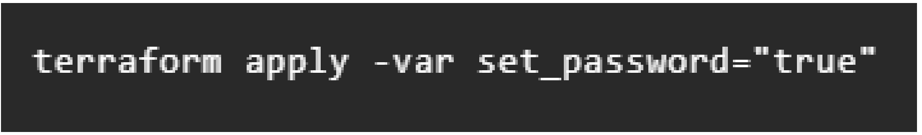
Terraform Variable Type: Bool



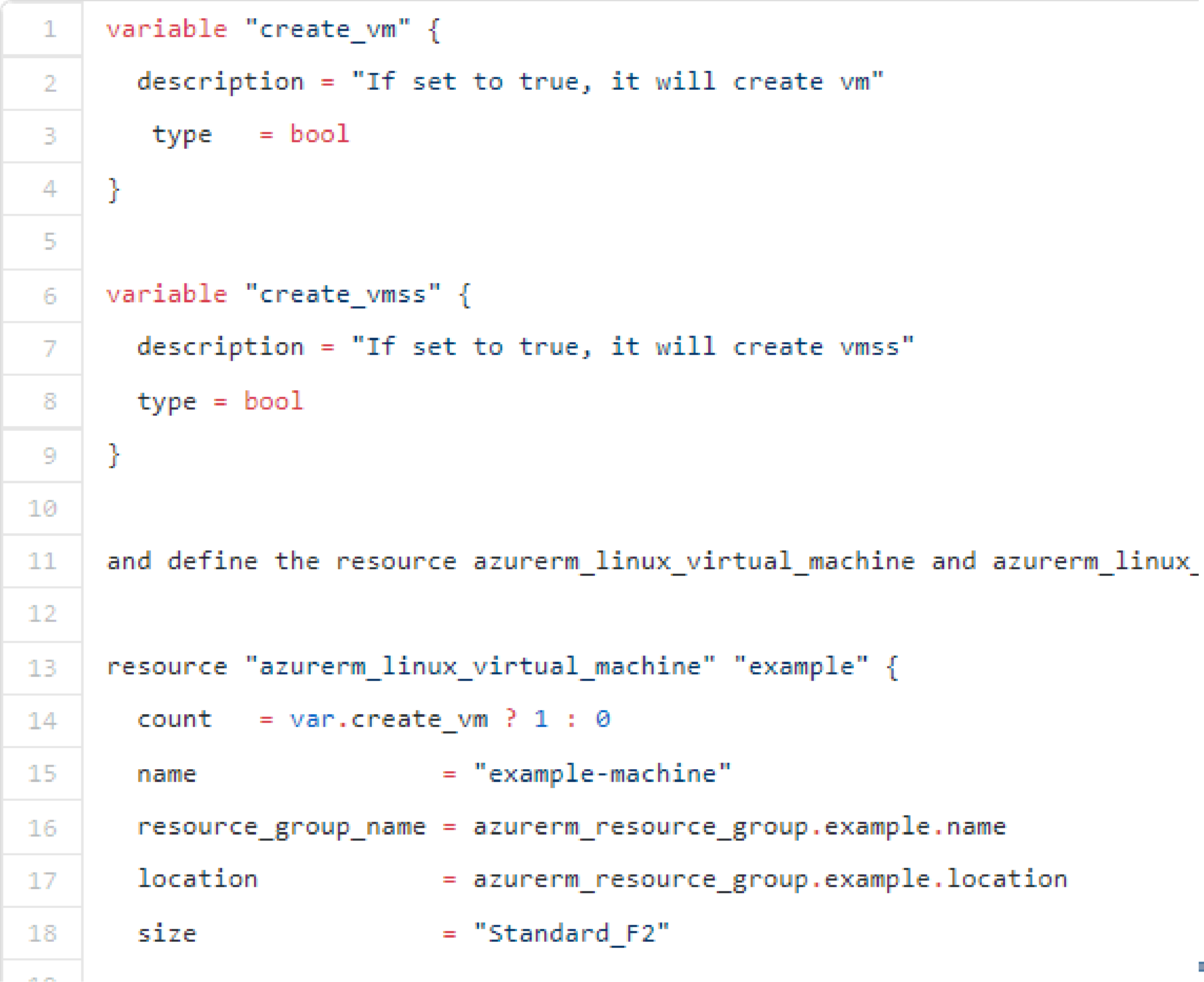
```
variable "set_password" {
  default = false
}
```



```
create_password = var.set_password
```



```
terraform apply -var set_password="true"
```



```
variable "create_vm" {
  description = "If set to true, it will create vm"
   type    = bool
}

variable "create_vmss" {
  description = "If set to true, it will create vmss"
  type = bool
}

and define the resource azurerm_linux_virtual_machine and azurerm_linux_

resource "azurerm_linux_virtual_machine" "example" {
  count    = var.create_vm ? 1 : 0
  name                = "example-machine"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  size                = "Standard_F2"
```

Terraform Output Variables

Terraform Variable: Output

Terraform will store hundreds or even thousands of attribute values for all the defined resources in our infrastructure in state file.

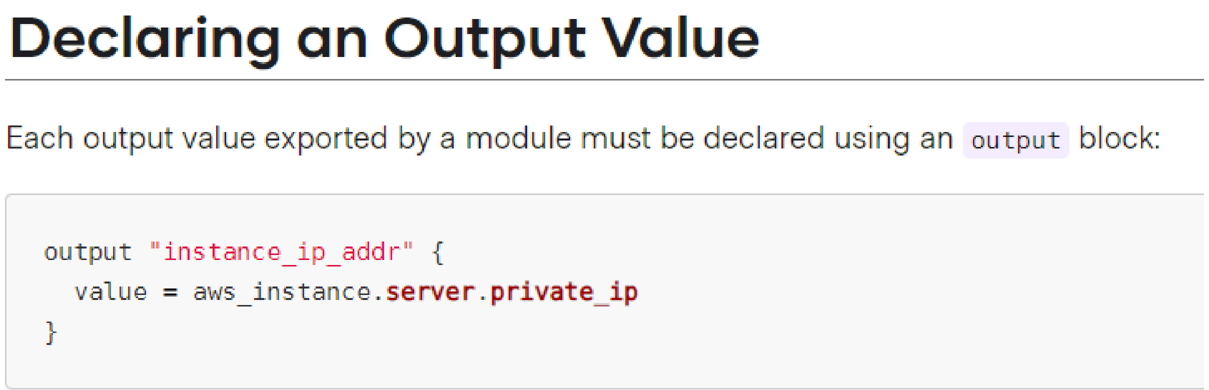
An outputted attributes can not only be used for the user reference but it can also act as an input to other resources being created via Terraform. We can use output variables to organize data to be easily queried and shown back to the Terraform user.

While Terraform stores hundreds or thousands of attribute values for all our resources, we are more likely to be interested in a few values of importance, such as a load balancer IP, VPN address, etc.

Output values are like the return values of a Terraform module, and have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running terraform apply.
- When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state data source.

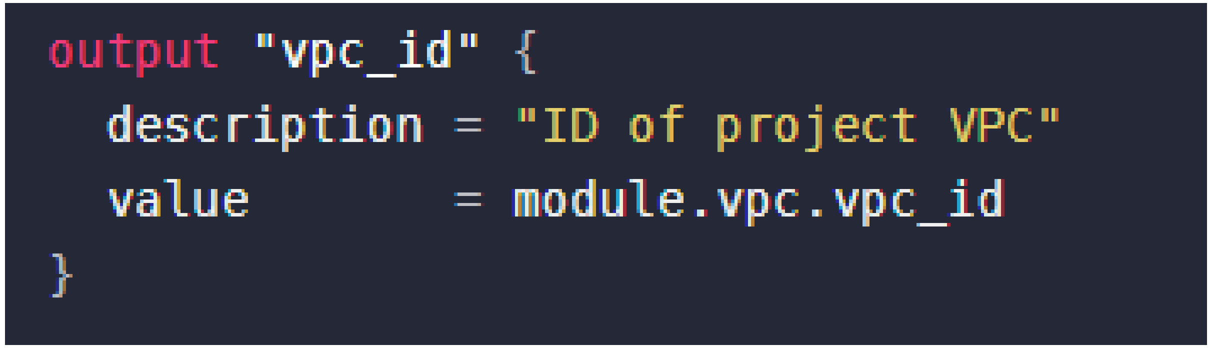
Terraform Variable: Output



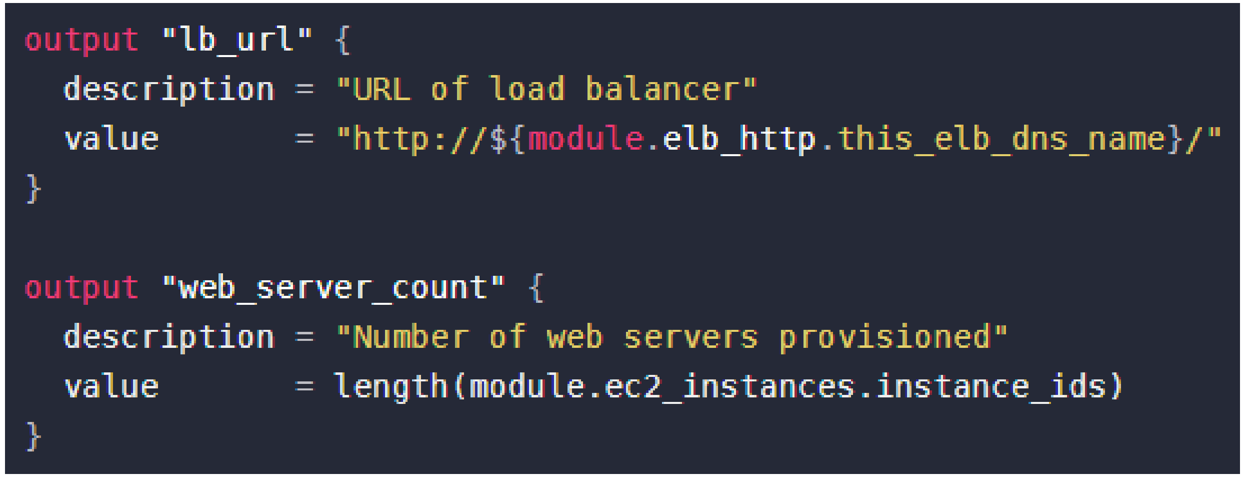
Declaring an Output Value

Each output value exported by a module must be declared using an `output` block:

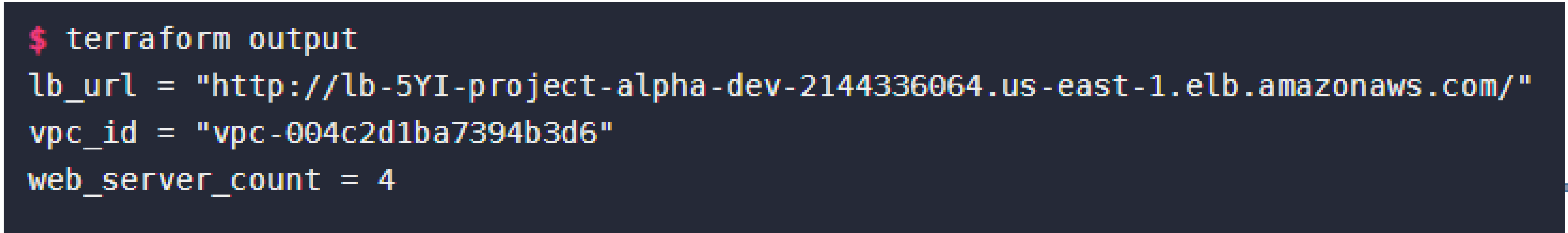
```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```



```
output "vpc_id" {
  description = "ID of project VPC"
  value       = module.vpc.vpc_id
}
```



```
output "lb_url" {
  description = "URL of load balancer"
  value       = "http://${module.elb_http.this_elb_dns_name}/"
}

output "web_server_count" {
  description = "Number of web servers provisioned"
  value       = length(module.ec2_instances.instance_ids)
}
```



```
$ terraform output
lb_url = "http://lb-5YI-project-alpha-dev-2144336064.us-east-1.elb.amazonaws.com/"
vpc_id = "vpc-004c2d1ba7394b3d6"
web_server_count = 4
```

Terraform Local Variables

What is local value in terraform?

These are variables that are local to a module. They are defined, assigned, and used in the same module, and defined in the “locals” block. Below is an example snippet on a local block:

Local variables can be declared once and used any number of times in the module. These can be accessed as objects by using the format of "local.Variable_Name".

Unlike variables found in programming languages, Terraform's locals don't change values during or between Terraform runs such as plan, apply, or destroy. You can use locals to give a name to the result of any Terraform expression, and re-use that name throughout your configuration. Unlike input variables, locals are not set directly by users of your configuration.

What is local value in terraform?

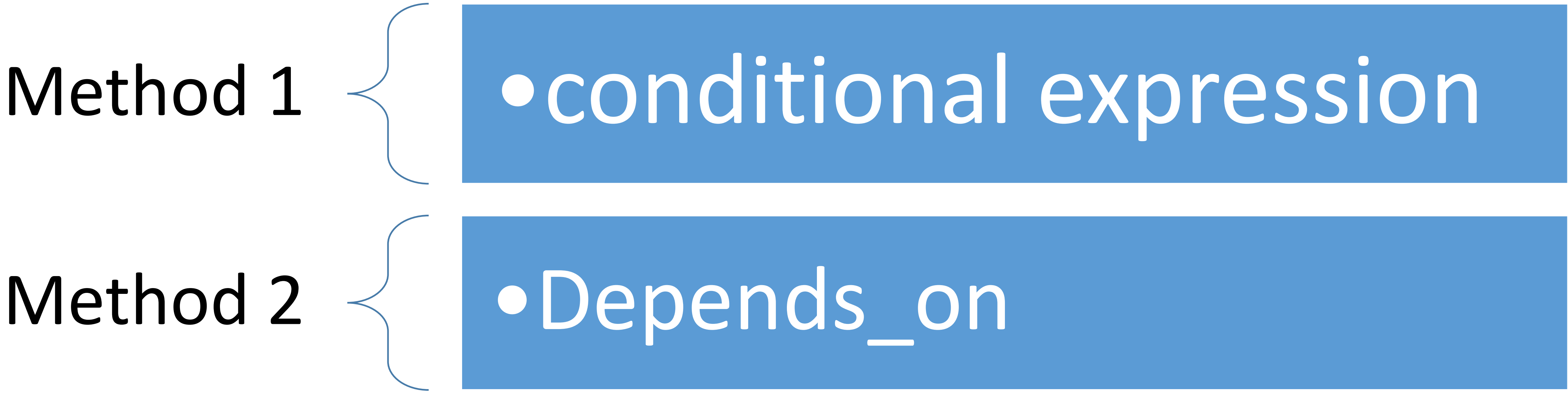
Comparing modules to functions in a traditional programming language:

- Input variables are analogous to function arguments and
- Outputs values are analogous to function return values, then
- local values are comparable to a function’s local temporary symbols.

When To Use Local Values?

- Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.
- Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future. The ability to easily change the value in a central place is the key advantage of local values.
- Each locals block can have as many locals as needed, and there can be any number of locals blocks within a module. The names given for the items in the local block must be unique throughout a module. The given value can be any expression that is valid within the current module.
- The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

Conditioning in Terraform



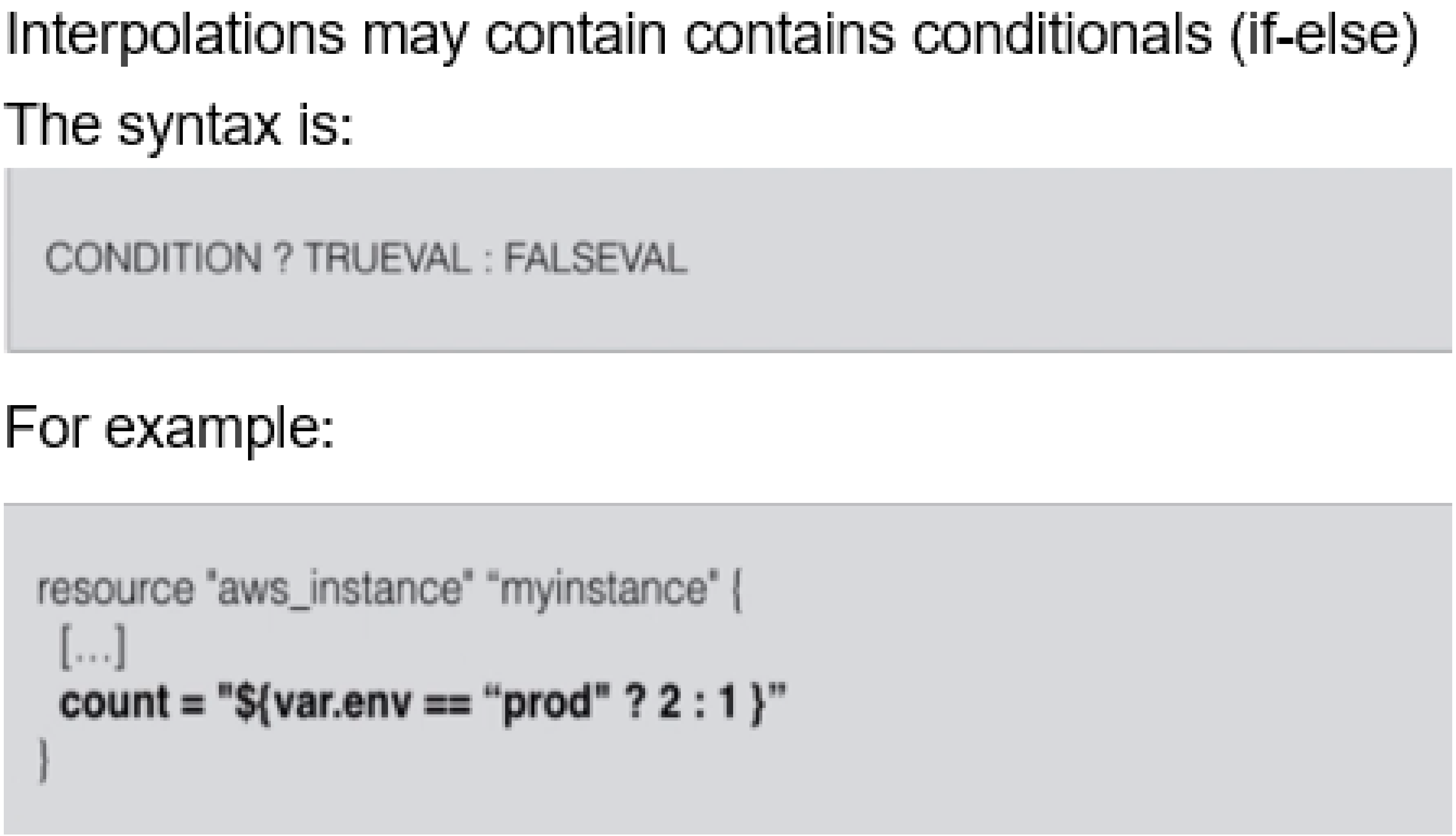
Method 1

- conditional expression

Method 2

- Depends_on

Conditions in Terraform



Interpolations may contain contains conditionals (if-else)

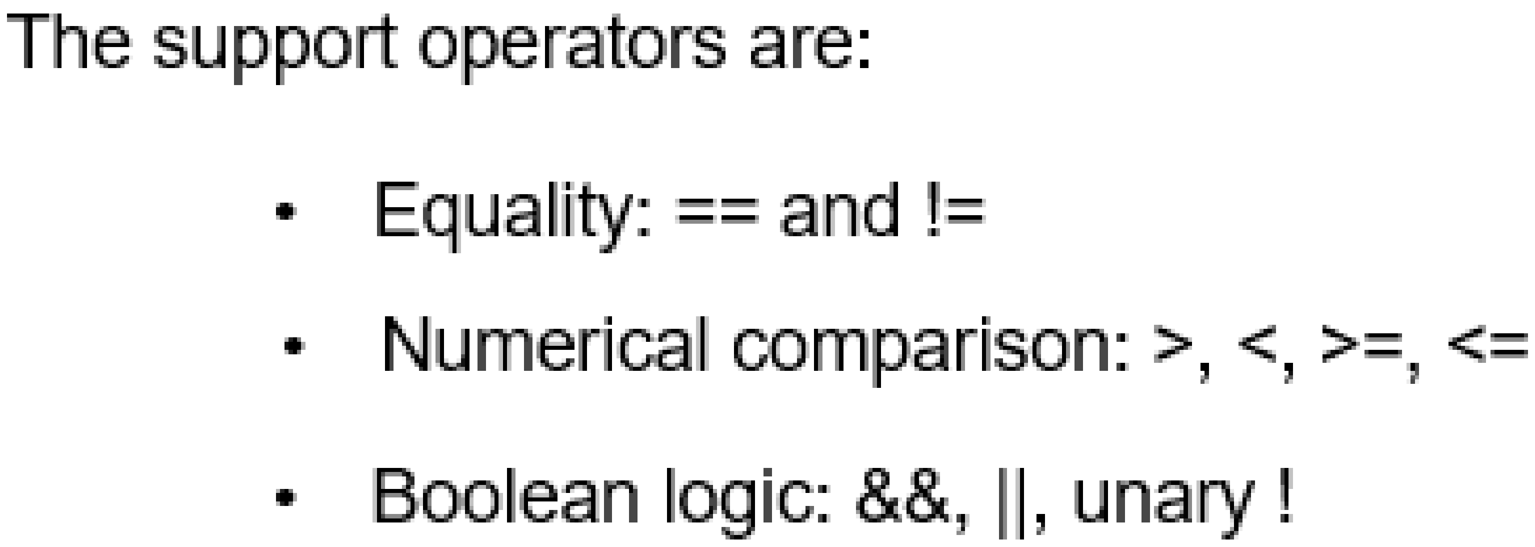
The syntax is:

```
CONDITION ? TRUEVAL : FALSEVAL
```

For example:

```
resource "aws_instance" "myinstance" {
  [...]
  count = "${var.env == “prod" ? 2 : 1 }"
}
```

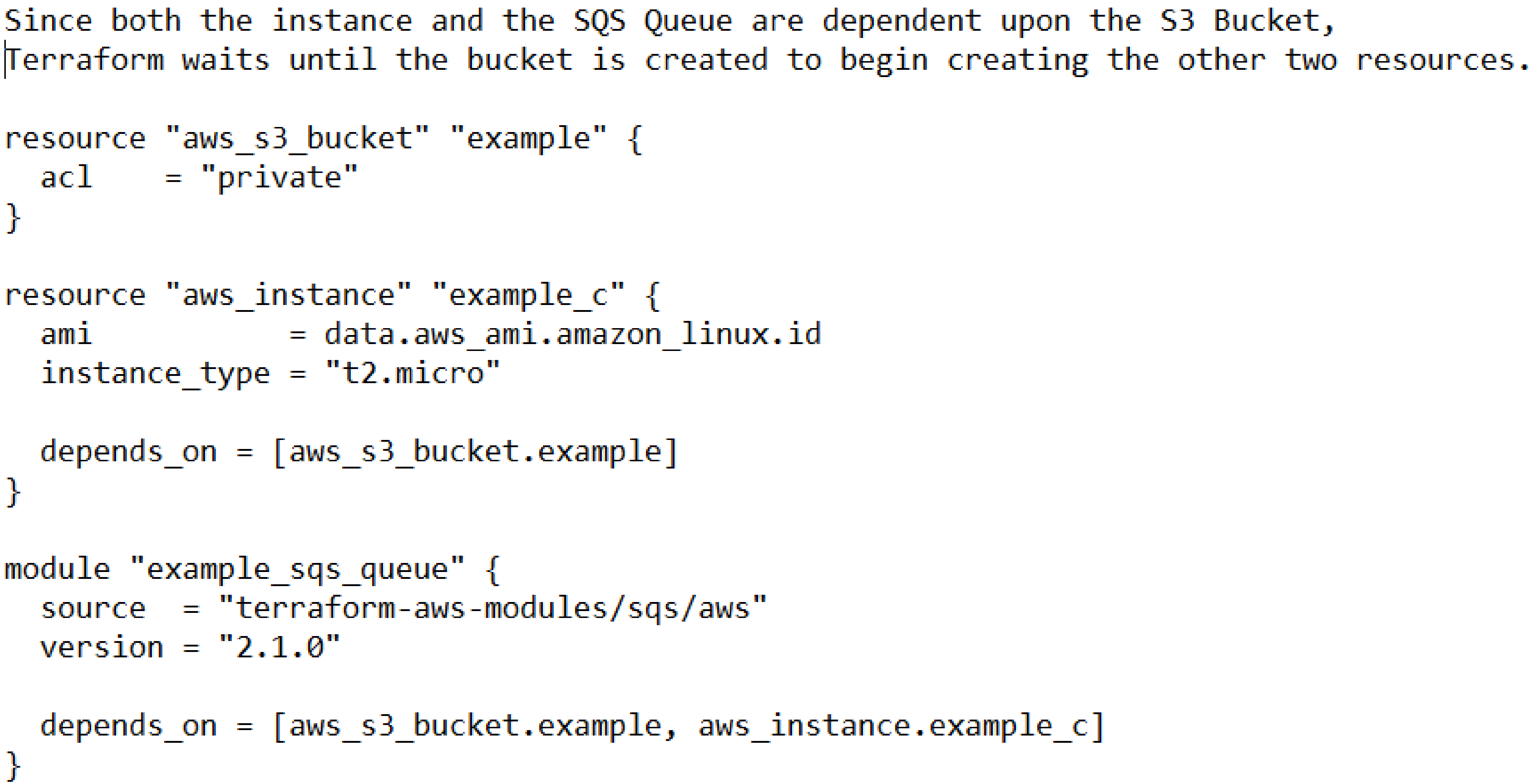
Conditions in Terraform



The support operators are:

- Equality: == and !=
- Numerical comparison: >, <, >=, <=
- Boolean logic: &&, ||, unary !

Conditions in Terraform



Since both the instance and the SQS Queue are dependent upon the S3 Bucket,
Terraform waits until the bucket is created to begin creating the other two resources.

```
resource "aws_s3_bucket" "example" {
  acl    = "private"
}

resource "aws_instance" "example_c" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t2.micro"

  depends_on = [aws_s3_bucket.example]
}

module "example_sqs_queue" {
  source  = "terraform-aws-modules/sqs/aws"
  version = "2.1.0"

  depends_on = [aws_s3_bucket.example, aws_instance.example_c]
}
```

Interpolation in Terraform

Interpolation

- In Terraform, you can Interpolate other values, Using ${...}
- You can use simple math functions, refer to other variables, or use conditionals (if-else)
- I have been using them already throughout the course, without naming them
    - Variables: ${var.VARIABLE-NAME} refers to a variable
    - Resources: ${aws_instance.name.id} (type.resource-name.attr)
    - Data Source: ${data.template_file.name.rendered} (data.type.resource-name.attr)

Interpolation: variables

Name	Syntax	Example
String variable	var.name	${var.SOMETHING}
Map variable	var.MAP["key"]	1) ${var.AMIS ["us-east-1"]} 2) 2) ${lookup(var.AMIS, var.AWS_REGION)}
List variable	var.LIST, var.LIST[i]	1) ${var.subnets [i]} 2) ${join(",", var.subnets)}

Interpolation: various

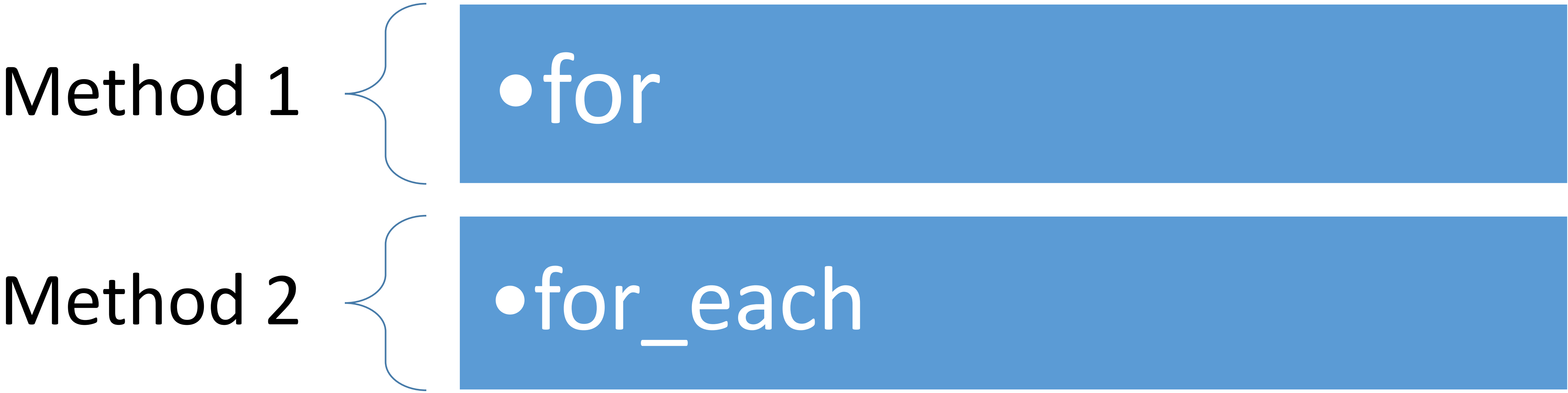
Name	Syntax	Example
Outputs of a module	module.Name.output	${module.aws_vpc.vpcid}
Count information	count.FIELD	When using the attribute count = number in a resource, you can use ${count.index}
Path information	Path.TYPE	path.cwd (current directory) path.module (module path) path.root (root module path)
Meta information	terraform.FIELD	terraform.env shows active workspace

Interpolation

Math:

- Add (+), Subtract (-), multiply (*), and Divide (/) for float types
- Add (+), Subtract (-), multiply (*), Divide (/) and Modulo (%) for float types for integer types
- For example: ${2+3*4} results in 14

Iterations in Terraform



Method 1

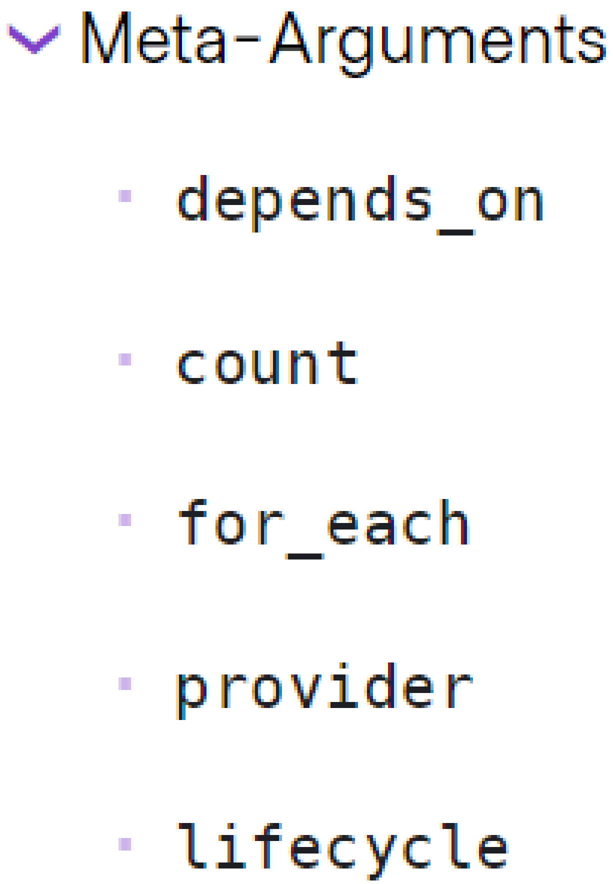
- for

Method 2

- for_each

https://www.devopsschool.com/blog/how-to-do-looping-iterations-in-terraform/

Resources Metadata



Meta-Arguments

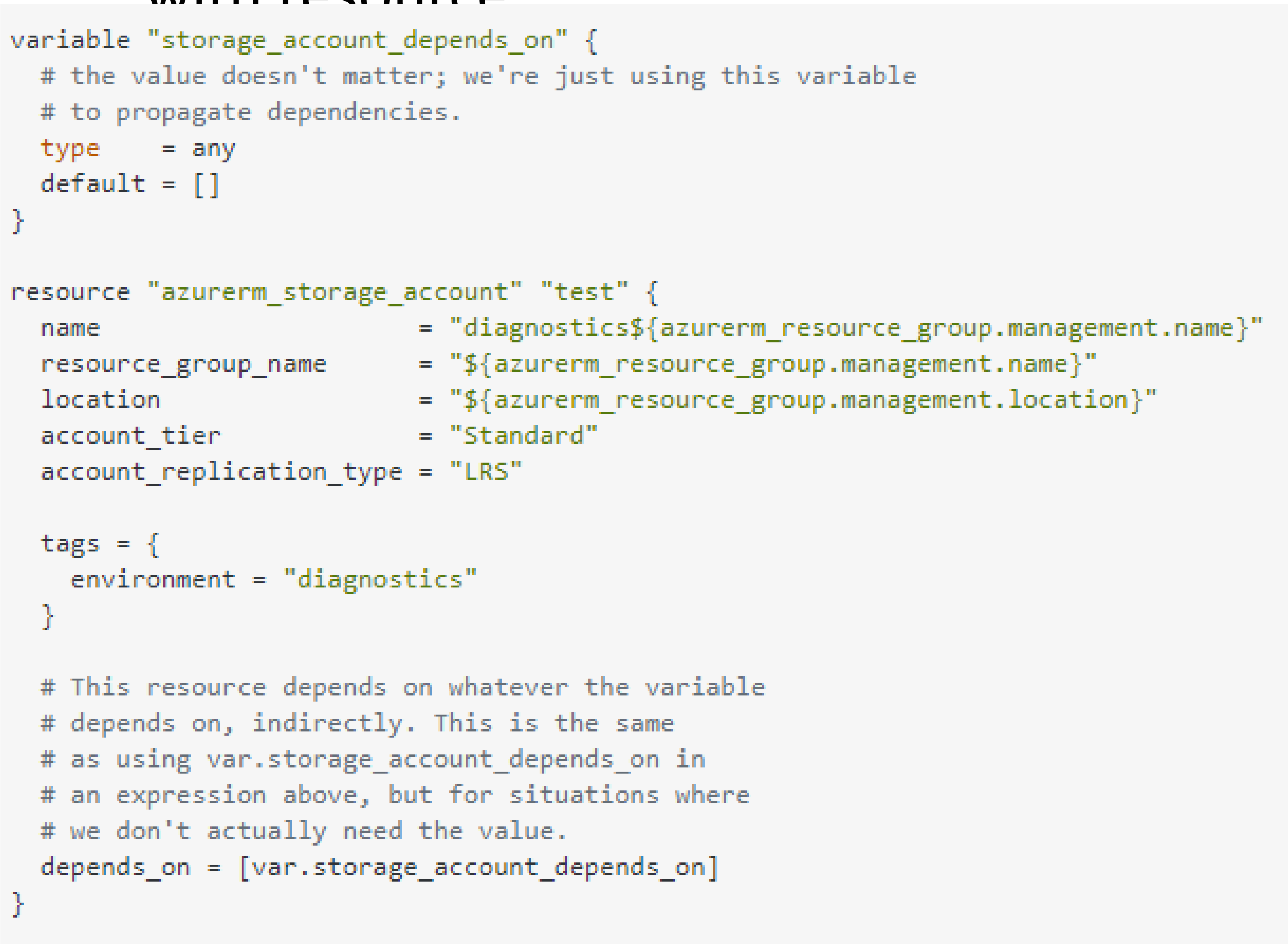
- `depends_on`
- `count`
- `for_each`
- `provider`
- `lifecycle`

Resource Meta Arg: depends_on

Use the depends_on meta-argument to handle hidden resource or module dependencies that Terraform can't automatically infer.

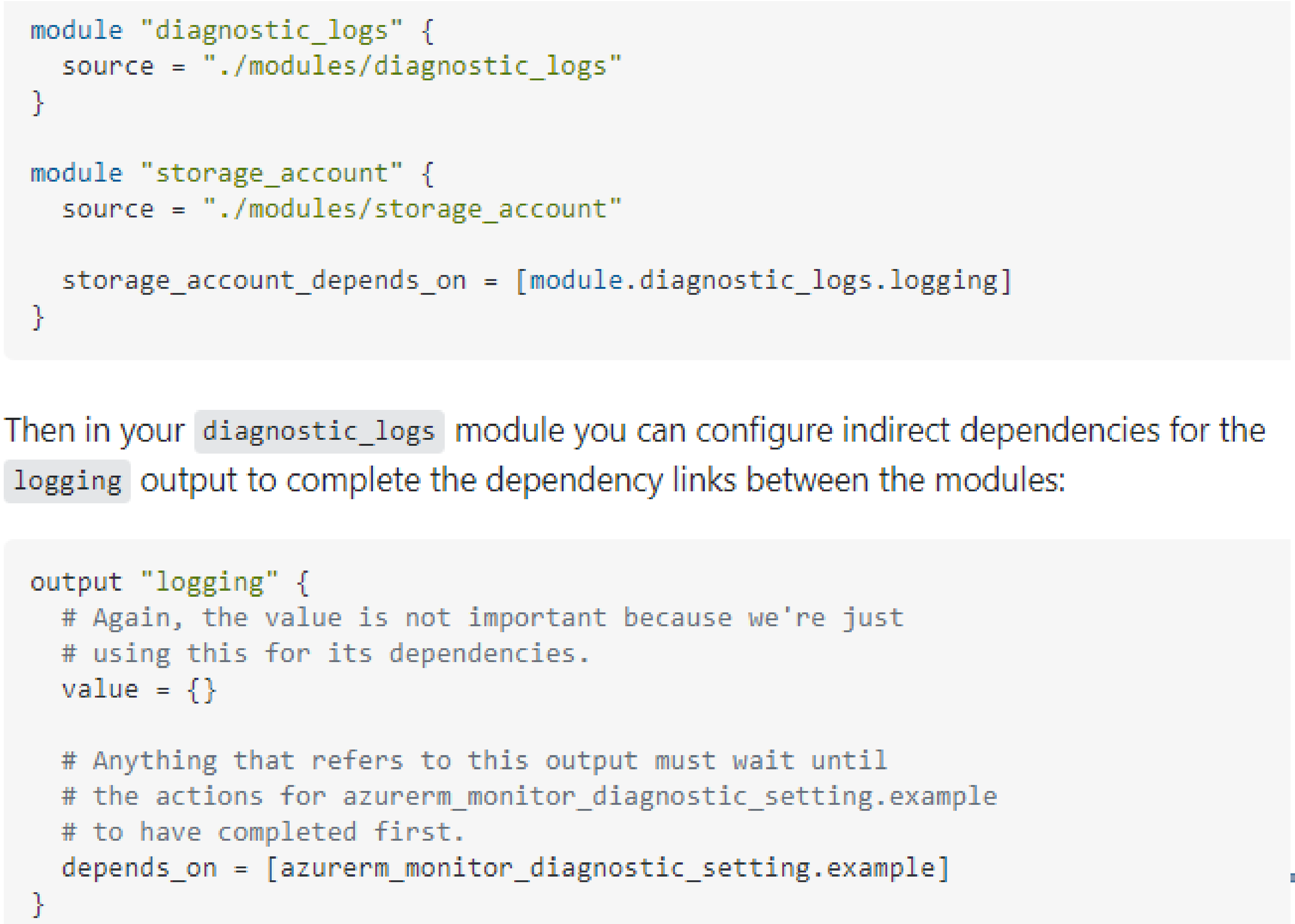
Explicitly specifying a dependency is only necessary when a resource or module relies on some other resource's behavior but doesn't access any of that resource's data in its arguments.

Resource Meta Arg: depends_on – with resource



```
variable "storage_account_depends_on" {
  # the value doesn't matter; we're just using this variable
  # to propagate dependencies.
  type    = any
  default = []
}

resource "azurerm_storage_account" "test" {
  name                     = "diagnostics${azurerm_resource_group.management.name}"
  resource_group_name      = "${azurerm_resource_group.management.name}"
  location                 = "${azurerm_resource_group.management.location}"
  account_tier             = "Standard"
  account_replication_type = "LRS"

  tags = {
    environment = "diagnostics"
  }

  # This resource depends on whatever the variable
  # depends on, indirectly. This is the same
  # as using var.storage_account_depends_on in
  # an expression above, but for situations where
  # we don't actually need the value.
  depends_on = [var.storage_account_depends_on]
}
```

Resource Meta Arg: depends_on – with Module



```
module "diagnostic_logs" {
  source = "./modules/diagnostic_logs"
}

module "storage_account" {
  source = "./modules/storage_account"

  storage_account_depends_on = [module.diagnostic_logs.logging]
}
```

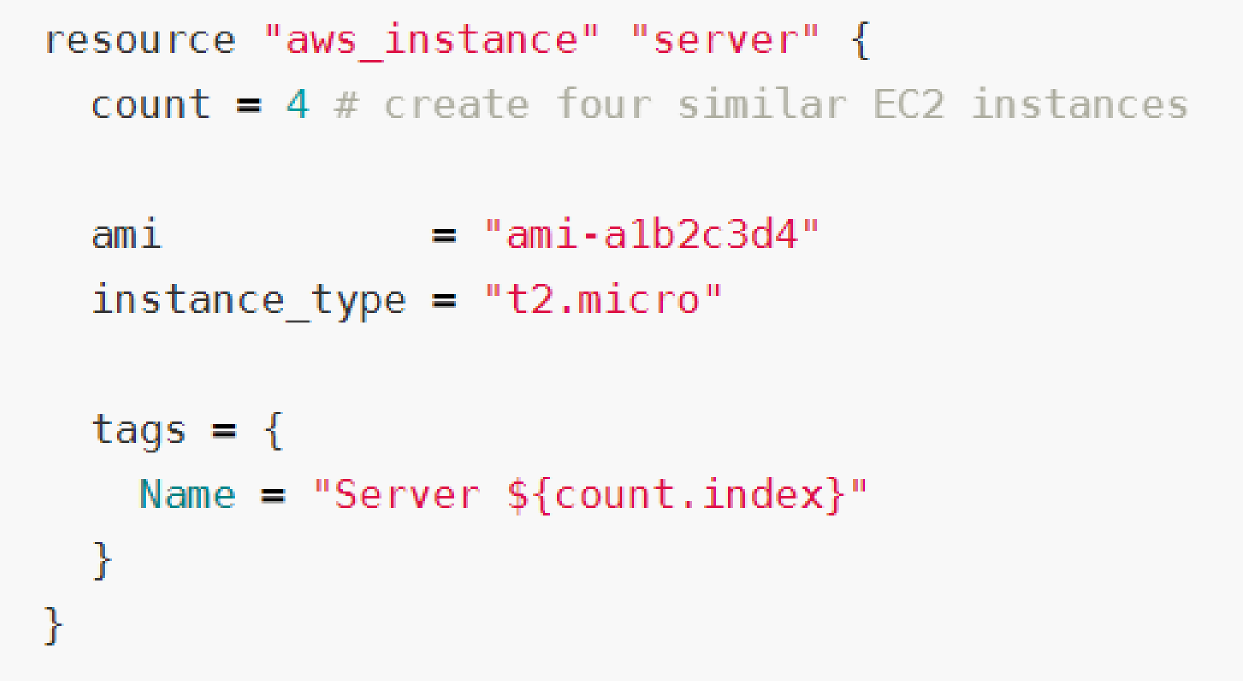
Then in your `diagnostic_logs` module you can configure indirect dependencies for the `logging` output to complete the dependency links between the modules:

```
output "logging" {
  # Again, the value is not important because we're just
  # using this for its dependencies.
  value = {}

  # Anything that refers to this output must wait until
  # the actions for azurerm_monitor_diagnostic_setting.example
  # to have completed first.
  depends_on = [azurerm_monitor_diagnostic_setting.example]
}
```

Resource Meta Arg: count

count is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type.

The count meta-argument accepts a whole number, and creates that many instances of the resource or module. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

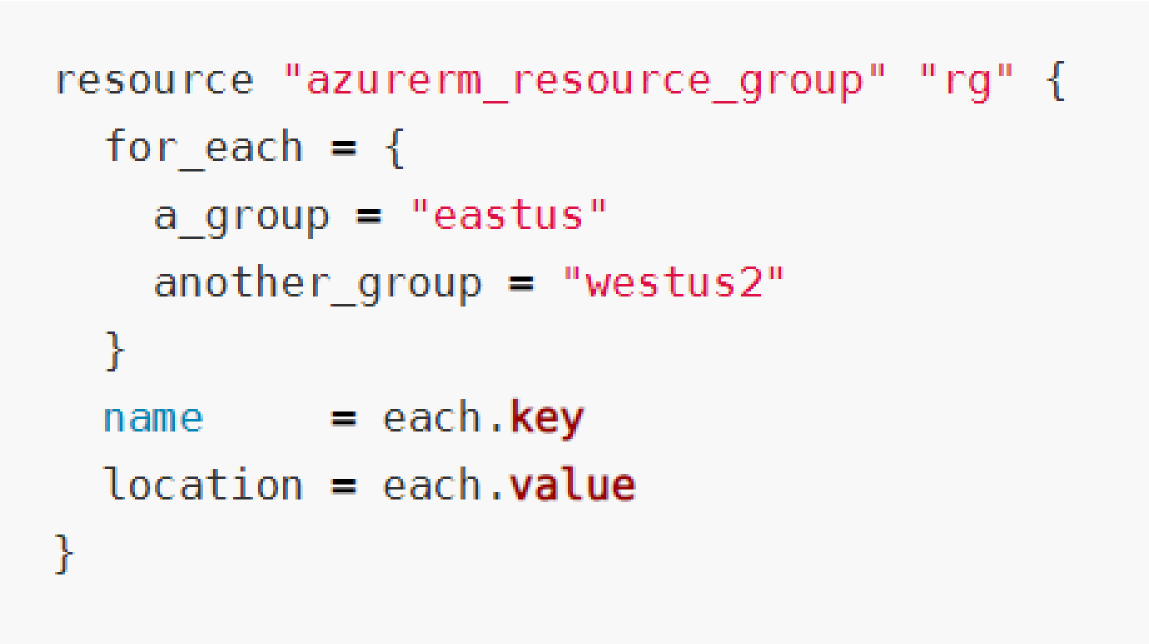


```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances

  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${count.index}"
  }
}
```

Resource Meta Arg: for_each

for_each is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type.

The for_each meta-argument accepts a map or a set of strings, and creates an instance for each item in that map or set. Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.



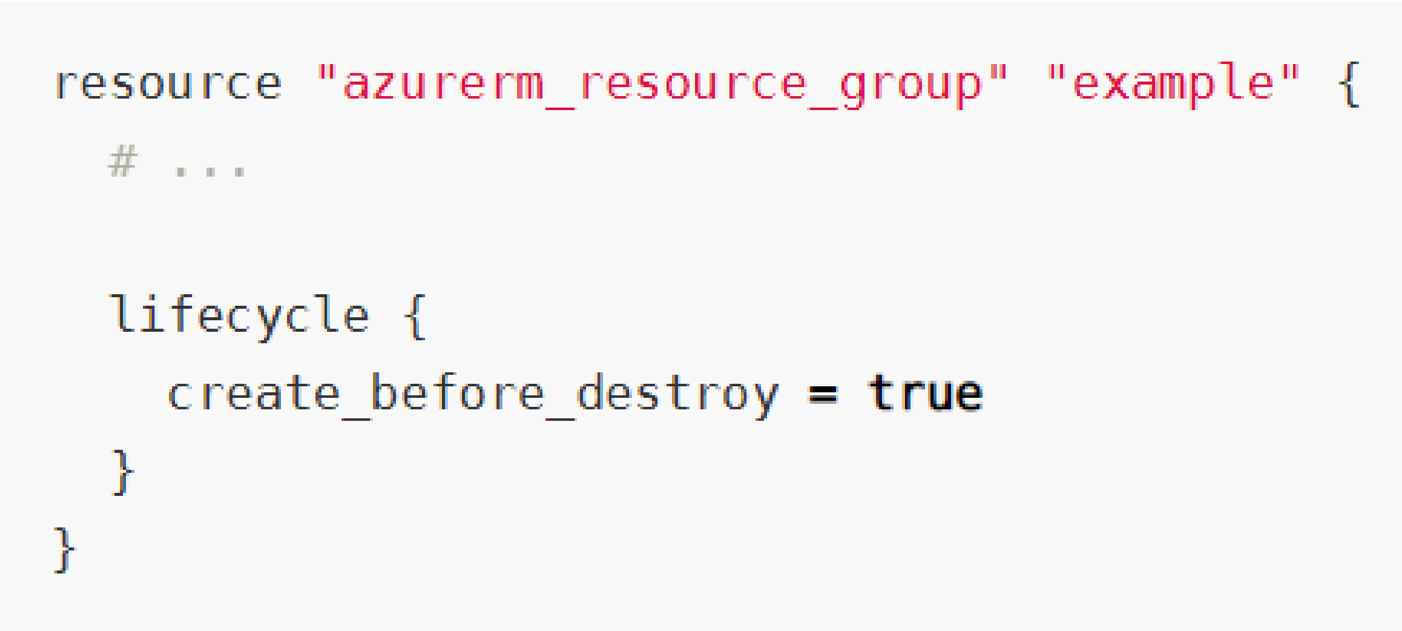
```
resource "azurerm_resource_group" "rg" {
  for_each = {
    a_group = "eastus"
    another_group = "westus2"
  }
  name     = each.key
  location = each.value
}
```

Resource Meta Arg: lifecycle

lifecycle is a nested block that can appear within a resource block. The lifecycle block and its contents are meta-arguments, available for all resource blocks regardless of type.

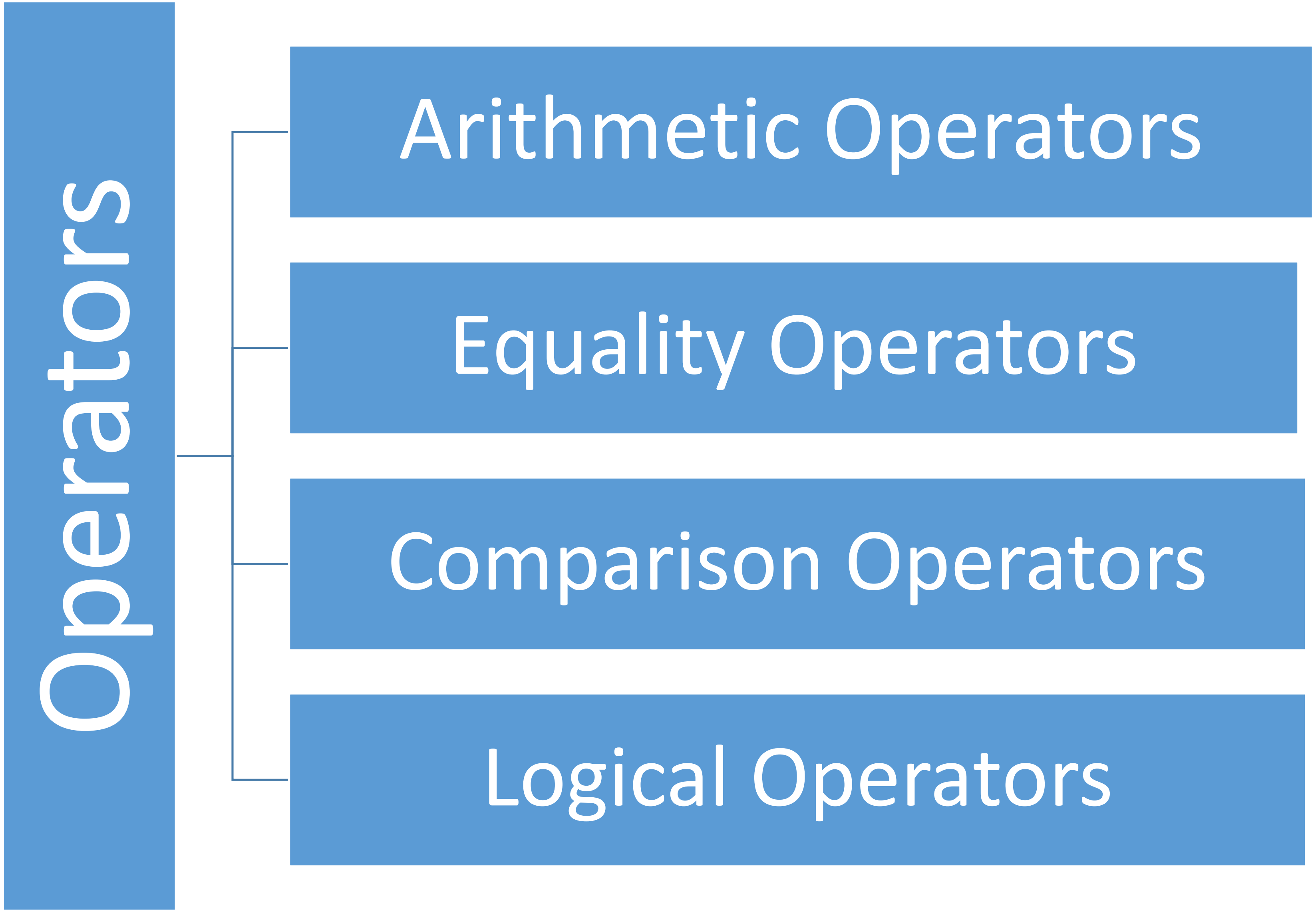
The following arguments can be used within a lifecycle block:

- create_before_destroy
- prevent_destroy
- ignore_changes



```
resource "azurerm_resource_group" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
  }
}
```

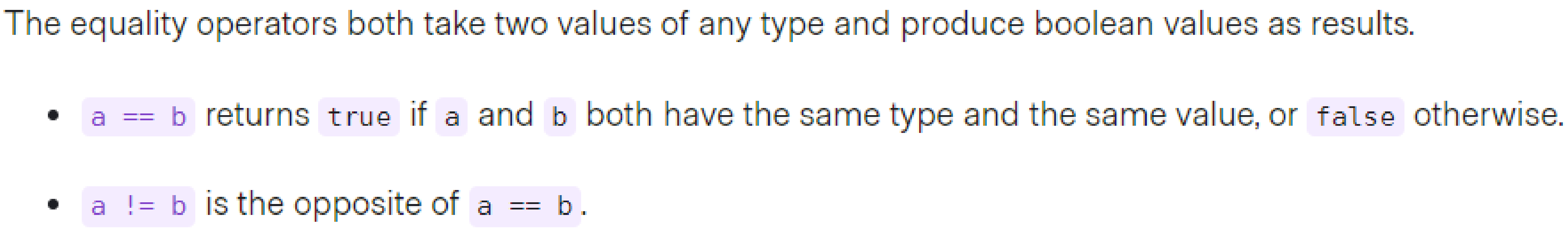
Operators in Terraform



Operators

- Arithmetic Operators
- Equality Operators
- Comparison Operators
- Logical Operators

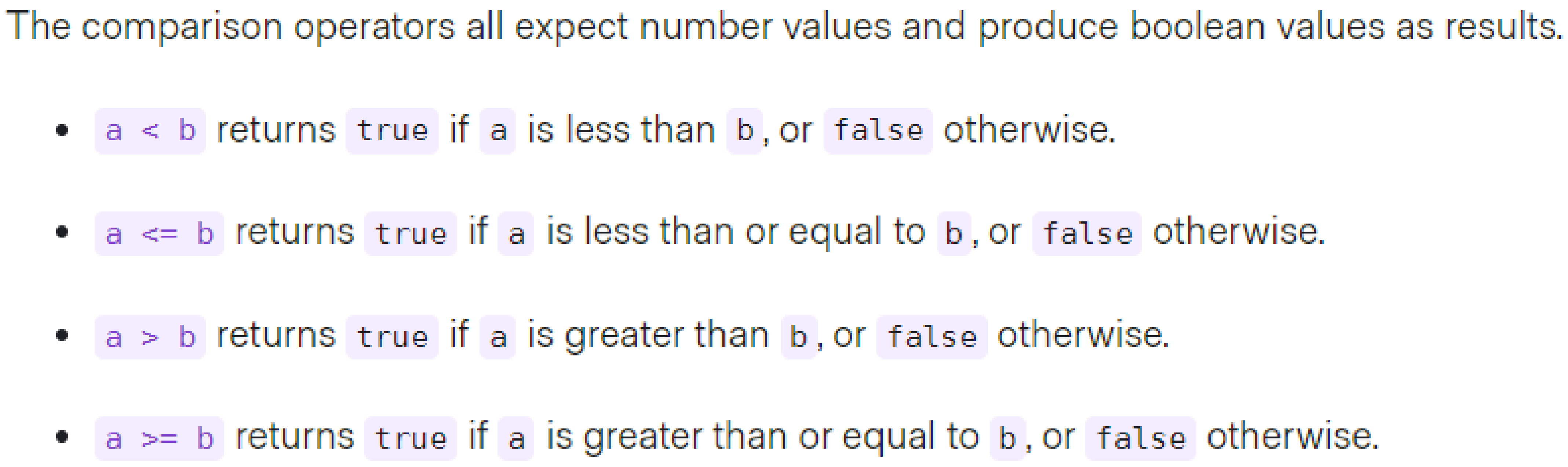
Terraform: **Equality Operators**



The equality operators both take two values of any type and produce boolean values as results.

- `a == b` returns `true` if `a` and `b` both have the same type and the same value, or `false` otherwise.
- `a != b` is the opposite of `a == b`.

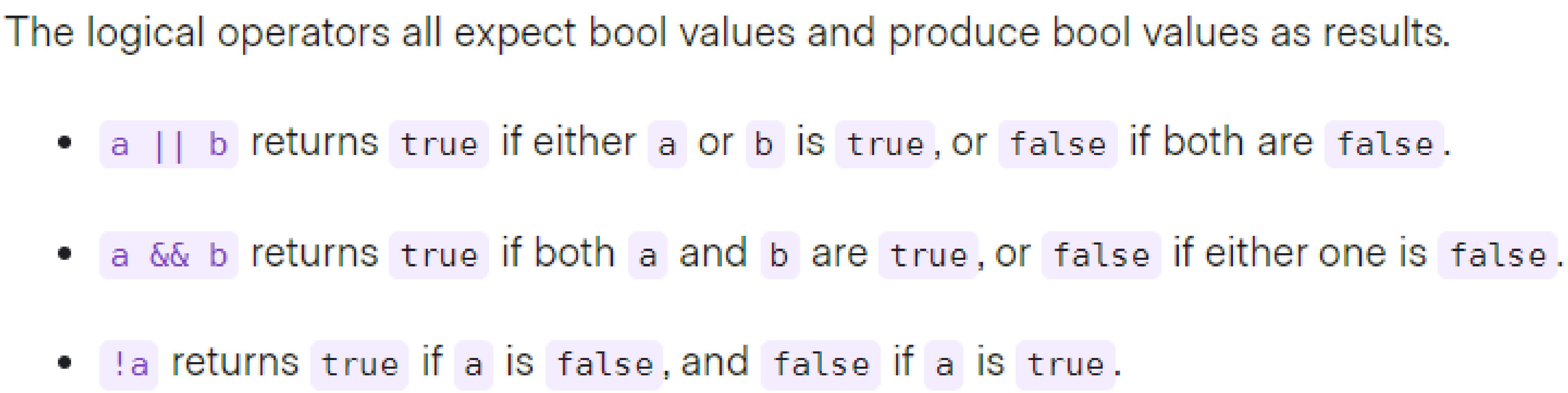
Terraform: **Comparison Operators**



The comparison operators all expect number values and produce boolean values as results.

- `a < b` returns `true` if `a` is less than `b`, or `false` otherwise.
- `a <= b` returns `true` if `a` is less than or equal to `b`, or `false` otherwise.
- `a > b` returns `true` if `a` is greater than `b`, or `false` otherwise.
- `a >= b` returns `true` if `a` is greater than or equal to `b`, or `false` otherwise.

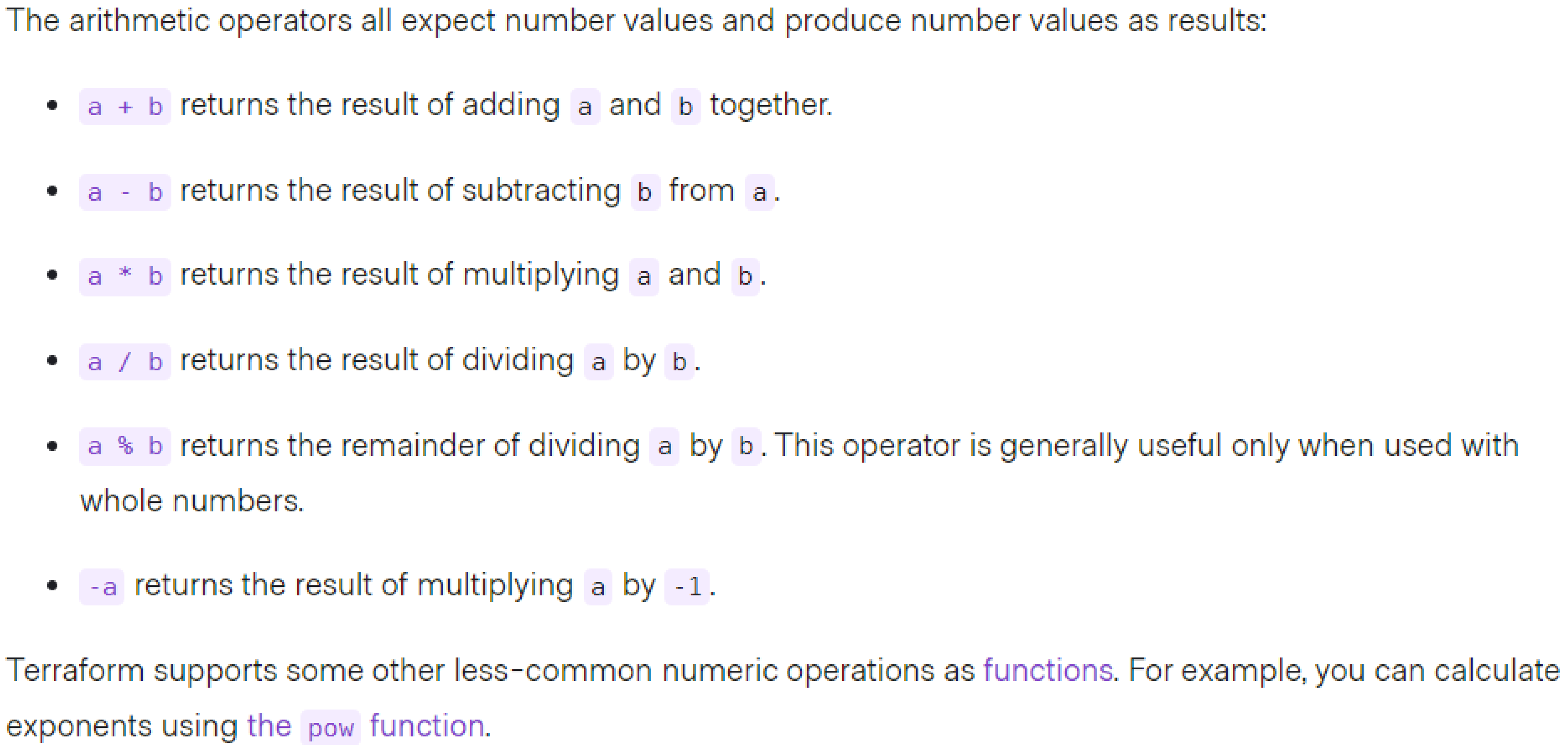
Terraform: **Logical Operators**



The logical operators all expect bool values and produce bool values as results.

- `a || b` returns `true` if either `a` or `b` is `true`, or `false` if both are `false`.
- `a && b` returns `true` if both `a` and `b` are `true`, or `false` if either one is `false`.
- `!a` returns `true` if `a` is `false`, and `false` if `a` is `true`.

Terraform: **Arithmetic Operators**



The arithmetic operators all expect number values and produce number values as results:

- `a + b` returns the result of adding `a` and `b` together.
- `a - b` returns the result of subtracting `b` from `a`.
- `a * b` returns the result of multiplying `a` and `b`.
- `a / b` returns the result of dividing `a` by `b`.
- `a % b` returns the remainder of dividing `a` by `b`. This operator is generally useful only when used with whole numbers.
- `-a` returns the result of multiplying `a` by `-1`.

Terraform supports some other less-common numeric operations as functions. For example, you can calculate exponents using the `pow` function.

Data Source in Terraform

Datasources

- For certain providers (like AWS), terraform provides datasources
- Datasources provide you with dynamic information

A lot of data is available by AWS in a structured format using their API

Terraform also exposes this information using data sources

- Examples:

list of AMIs

List of availability Zones

Datasources

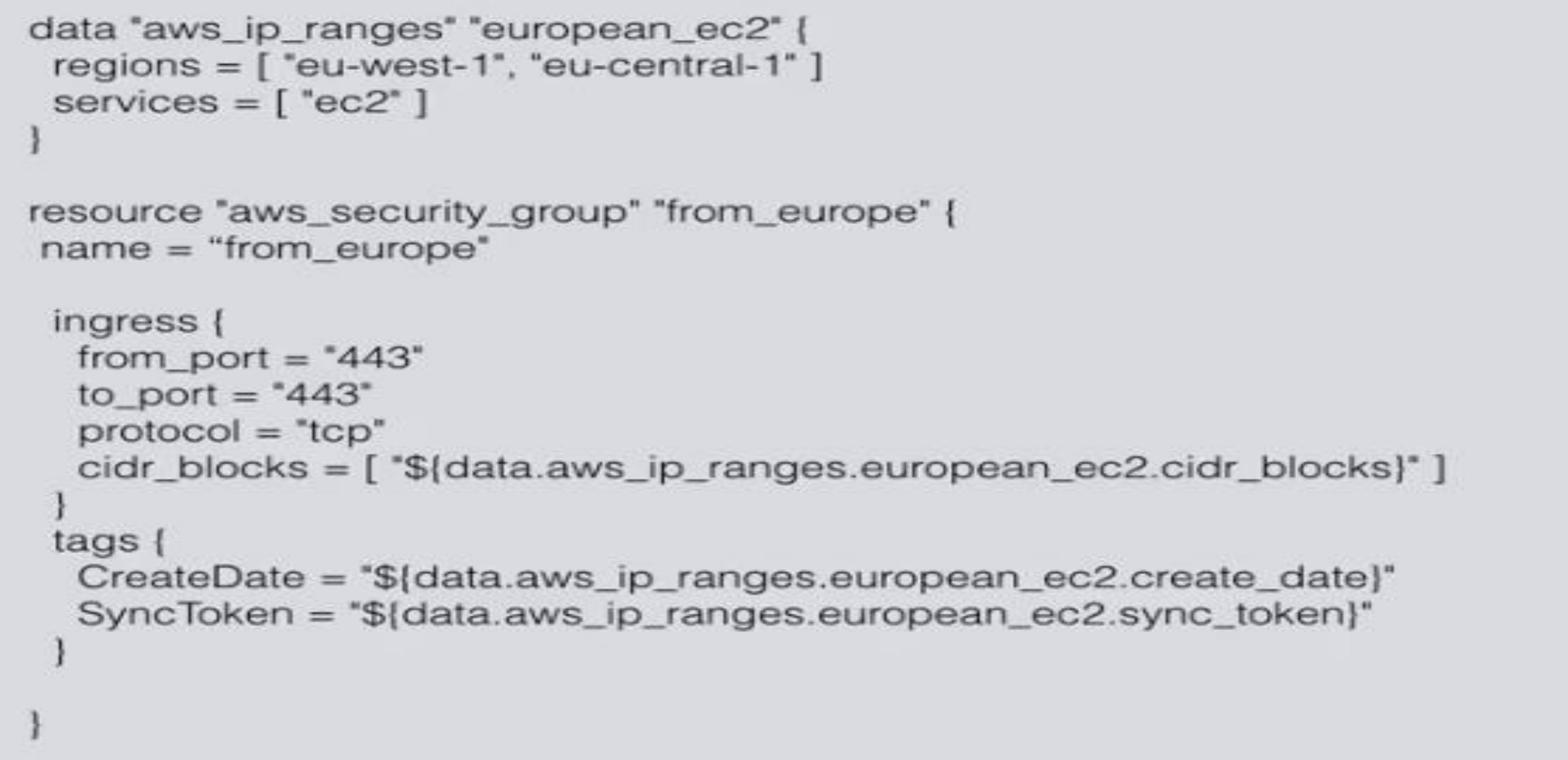
- Another great example is the datasource that gives you all IP addresses in use by AWS
- This is great if you want to filter traffic based on an AWS region

  e.g. allow all traffic from amazon instances in Europe
- Filtering traffic in AWS can be done using security groups

  Incoming and outgoing traffic can be filtered by protocol, IP range, and port

  Similar to iptables (Linux) or a firewall appliance

Datasources



```
data "aws_ip_ranges" "european_ec2" {
  regions = [ "eu-west-1", "eu-central-1" ]
  services = [ "ec2" ]
}

resource "aws_security_group" "from_europe" {
 name = “from_europe"

  ingress {
    from_port = "443"
    to_port = "443"
    protocol = "tcp"
    cidr_blocks = [ "${data.aws_ip_ranges.european_ec2.cidr_blocks}" ]
  }
  tags {
    CreateDate = "${data.aws_ip_ranges.european_ec2.create_date}"
    SyncToken = "${data.aws_ip_ranges.european_ec2.sync_token}"
  }

}
```

https://www.devopsschool.com/blog/data-sources-in-terraform-resources-explained-with-example/

Assignment

Output all the UBUNTU VM Image from Azure from one region but Use one of these image in Creating VM.

Debug

- TF_FORK=0

  add an environment variable to prevent forking #8795
- TF_LOG → TRACE, DEBUG, INFO, WARN or ERROR
- TF_LOG_PATH
- Found a bug? Report to the right place:
  - Check GitHub, it’s highly likely a known bug/”feature”
  - https://github.com/hashicorp/terraform - Core Issues
  - https://github.com/terraform-providers - Provider Plugins
  - Check Golang SDK’s bugs
  - Check Cloud provider documentation
- Don’t forget to obfuscate your crash log :)
- Use delve debugger to learn how Terraform core works!

Terraform tips

1. Use terraform console
   a. echo "random_string.new.result" | terraform console
2. Use workspaces for simple scenarios
3. Isolate state files and don’t use workspaces :)
4. To review output from terraform modules:

   terraform output -module=mymodule
5. My state is changed every time when I’m running terraform with different users! (binary files/lambda functions)
   a. substr("${path.module}"/, length(path.cwd) + 1, -1)
   b. ignore_changes = ["filename"]

Terraform: common workflow issues

1. Mess up with workspaces
2. Hard-coded values
3. Not following naming convention (tags)
4. TF can’t detect changes to computed values
5. Renaming modules, resources
6. Double references
7. Syntax problems
8. Variable “somevar” should be type map, got list
9. Timeouts
10. Permissions

Terraform: Sensitive information

1. terraform plan “-out plan-latest” - is not secured
2. terraform state - not secured.
   a. Encryption on backend at rest
   b. terraform pull - exposes sensitive
   c. use data sources - grant only what you need
3. Data remote state - Not possible to expose just single or few outputs
4. Sensitive output
5. Encrypt tfvars
6. terraform output sensitive = true
   a. seems ok? remote_secured = <sensitive>
   b. terraform refresh → exposed, remote_secured = 79e6

Terraform: Sensitive information

1. How to handle secrets in state file?
   a. Terrahelp - https://github.com/opencredo/terrahelp
   b. Don’t store secrets :) Use:
      i. AWS/Google Cloud/Azure Key Vault/ etc. KMS -like + user-data mechanisms
      ii. AWS System Manager Parameter store
      iii. AWS Secrets manager
      iv. Use resource Roles
      v. If set master-password for DB service - change it after creation.
2. Secure state at rest using backend built-in encryption
3. Secure tfvars and other project/module specific information with:
   a. pass - The password store - https://www.passwordstore.org/
   b. git-crypt - https://github.com/AGWA/git-crypt