Terraform Interview Questions

1. **What are all the components that you have created through Terraform?**

   So we have worked with AWS cloud so we have worked on various resources like Ec2, S3,VPC, RDS Etc

```
Eg- resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags = {
    Name = "HelloWorld"
  }
}
resource "aws_s3_bucket" "example" {
  bucket = "my-tf-test-bucket"

  tags = {
    Name        = "My bucket"
    Environment = "Dev"
  }
}
```

2. **How to do changes in the configuration of already resources using Terraform?**

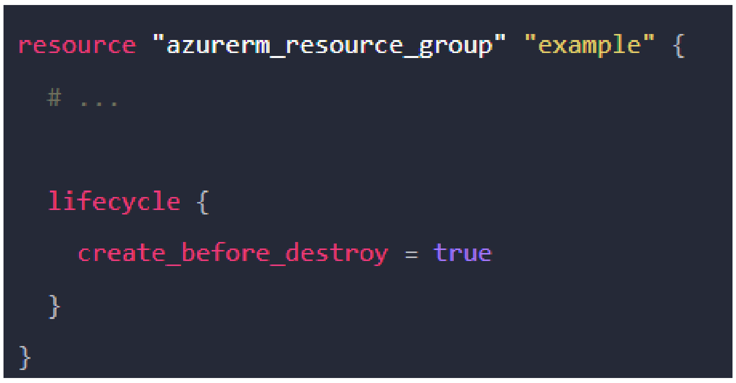
   Using Terraform Import commands

   Synx - terraform import aws_instance.foo i-abcd1234

3. **When terraform run state file creates what is the purpose of it and where do u store it?**
   So terraform maintains statefile that maps the current status of infra with the configuration and it will be stored wither locally or s3 in AWS
   It will be stored with terraform.tfstatefile
4. **If we lose the statefile how to get back it?**
   The way to fix that is to run the terraform import command, where you pass in the Terraform resource ID and the corresponding AWS resource ID.
5. **Terraform - Delete all resources except one?**
   So follow the below steps
   # list all resources
   terraform state list
   # remove that resource you don't want to destroy
   # you can add more to be excluded if required
   terraform state rm <resource_to_be_deleted>
   # destroy the whole stack except above excluded resource(s)
   terraform destroy
   terraform import
6. **Main Features of Terraform?**
   Its Cloud agonistic, Use HCL(Hashicorp config language)
7. **What does Terraform Validate do?**

Validates the syntax changes in the terraform files foreg-HCL syntax, if resources declared multiple times, module declared multiple times etc

8. **Lifecycle of Terraform?**
   Lifecycle is a nested block within resource block



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

  lifecycle {
    create_before_destroy = true
  }
}
```

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 arguments available within a lifecycle block are create_before_destroy, prevent_destroy, ignore_changes, and replace_triggered_by

9. **What to choose Terraform or Ansible?**

   Terraform – Declarative approach it works best for maintaining series state without much intervention. Building Infra from scratch

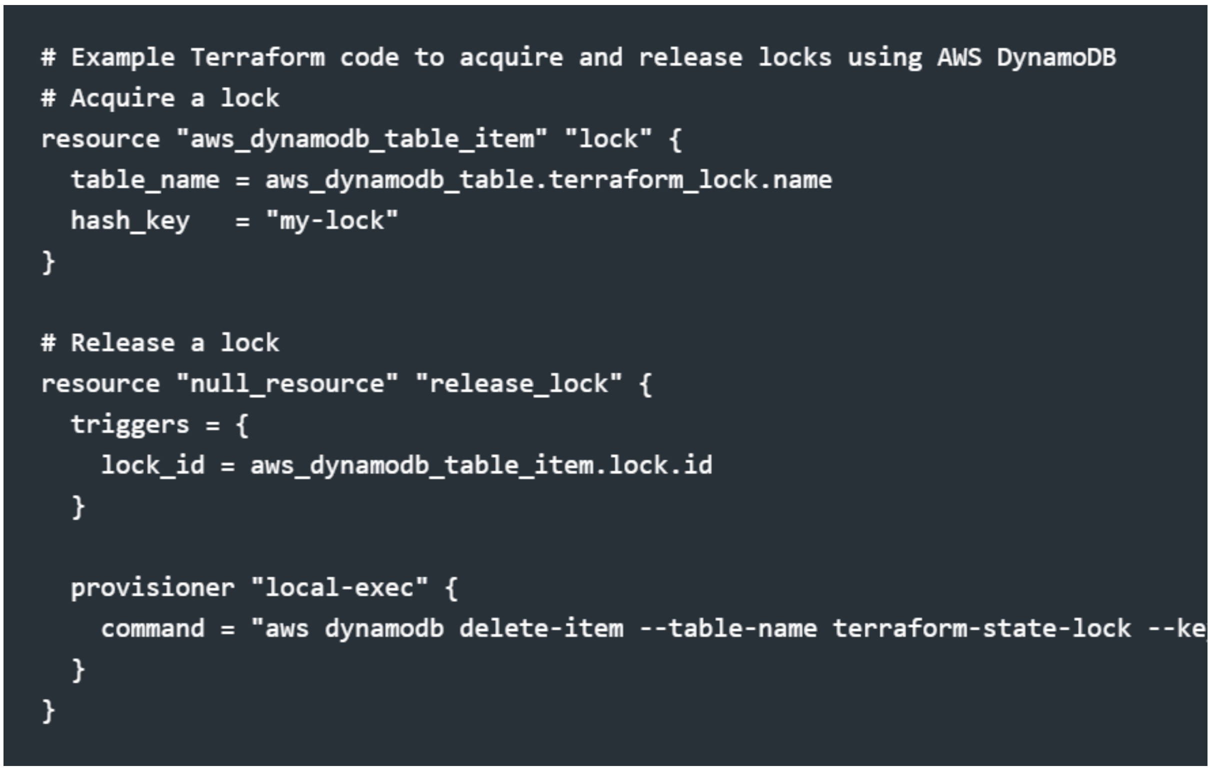
   Ansible – Procedural approach, it is for those who needs to manage and configure their infra in the way its data is changing regularly like adding softwares etc

   Both are fully featured and secured.

10. **How to destroy one specific resource in the terraform?**
    Using Terraform destroy -target <resourcetype>.<resourcename>
11. **What are diff kind of modules in Terraform?**
    Root modules – all resources defined
    Child – which call other modules (root module)
    Publish—public/private modules
12. The module that gets called – Parent module/Root module
13. **Remote Backend?**
    Makes easier for team to work collaboratively where we can store the statefile and share will all of them.
14. **How do you provide variable at the runtime?**
    Remove the variable from variable.tf file and use the variable from tfvars
15. **How do you use variables in the Command Line?**
    terraform apply -var="image_id=ami-abc123"
    terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.micro"
    terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
16. **How to manage terraform code in the multiple environments?**
    Terraform Workspaces – Separate environments(Separate statefiles for diff environments)
17. **Drawbacks of Terraform what ever we faced?**
    Error handling in the Terraform

We have to use Only HCL(hashicorp language)
It does not support script generation

18. **What is statefile Locking?**
    If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state. State locking happens automatically on all operations that could write state.
19. **How do you implement Statefile locking?**
    In this scenario, the team sets up a DynamoDB table specifically for state locking. Each Terraform run checks for a lock in DynamoDB before proceeding. If the lock is acquired, the Terraform run proceeds; otherwise, it waits for the lock to be released
    DynamoDB is an excellent choice for state locking in Terraform due to its high availability, scalability, and low-latency performance. It offers fine-grained control over access, which is crucial for preventing simultaneous writes to the state.



```
# Example Terraform code to acquire and release locks using AWS DynamoDB
# Acquire a lock
resource "aws_dynamodb_table_item" "lock" {
  table_name = aws_dynamodb_table.terraform_lock.name
  hash_key   = "my-lock"
}

# Release a lock
resource "null_resource" "release_lock" {
  triggers = {
    lock_id = aws_dynamodb_table_item.lock.id
  }

  provisioner "local-exec" {
    command = "aws dynamodb delete-item --table-name terraform-state-lock --ke
  }
}
```

20. **What is a null resource?**
    A null resource is basically something that doesn't create anything on its own, but you can use it to define provisioners blocks. They also have a “trigger” attribute, which can be used to recreate the resource, hence to rerun the provisioner block if the trigger is hit.
21. **What is a tainted Terraform Resource?**
    Terraform Taint command Informs terraform that particular objects is degraded or damaged so that terraform marks this resourcs as tained in the statefile and next time we check plan it shows replace the object.
22. **Difference between Count and ForEach Meta arguments?**
    f your instances are almost identical, count is appropriate. If some of their arguments need distinct values that can't be directly derived from an integer, it's safer to use for_each . This was fragile, because the resource instances were still identified by their index instead of the string values in the list.

Terraform Commands:

1. **terraform version** - Show the current version of your Terraform and notifies you if there is a newer version available for download.
2. **terraform fmt** — Format your Terraform configuration files using the HCL language standard.
3. **terraform fmt --recursive** — Also format files in subdirectories
4. **terraform init** — In order to prepare the working directory for use with Terraform, the terraform init command performs Backend Initialization, Child Module Installation, and Plugin Installation.
5. **terraform init -get-plugins=false** — Initialize the working directory, do not download plugins.
6. **terraform init -lock=false** — Initialize the working directory, don’t hold a state lock during backend migration.
7. **terraform init -verify-plugins=false** — Initialize the working directory, do not verify plugins for Hashicorp signature
8. **terraform validate** — Validate the configuration files in your directory and does not access any remote state or services. terraform init should be run before this command.
9. **terraform plan** — Plan will generate an execution plan, showing you what actions will be taken without actually performing the planned actions.
10. **terraform apply** — Create or update infrastructure depending on the configuration files. By default, a plan will be generated first and will need to be approved before it is applied.
11. **terraform apply -auto-approve** — Apply changes without having to interactively type ‘yes’ to the plan. Useful in automation CI/CD pipelines.
12. **terraform apply -lock=false** — Do not hold a state lock during the Terraform apply operation. Use with caution if other engineers might run concurrent commands against the same workspace.
13. **terraform apply -var="environment=dev"** — Pass in a variable value.
14. **terraform apply -var-file="varfile.tfvars"** — Pass in variables contained in a file.
15. **terraform destroy** — Destroy the infrastructure managed by Terraform.
16. **terraform destroy --auto-approve** — Destroy the infrastructure without having to interactively type ‘yes’ to the plan. Useful in automation CI/CD pipelines.
17. **terraform refresh** — Modify the state file with updated metadata containing information on the resources being managed in Terraform. Will not modify your infrastructure.
18. **terraform show** — Show the state file in a human-readable format.
19. **terraform state list** — Lists out all the resources that are tracked in the current state file.
20. **terraform state mv** — Move an item in the state, for example, this is useful when you need to tell Terraform that an item has been renamed, e.g. terraform state mv vm1.oldname vm1.newname
21. **terraform state pull > state.tfstate** — Get the current state and outputs it to a local file.
22. **terraform state push** — Update remote state from the local state file.
23. **terraform state rm** — Remove the specified instance from the state file. Useful when a resource has been manually deleted outside of Terraform.
24. **terraform state show <resourcename>** — Show the specified resource in the state file.
25. **terraform import vm1.name -i id123** — Import a VM with id123 into the configuration defined in the configuration files under vm1.name.
26. **terraform workspace show** — Show the name of the current workspace.

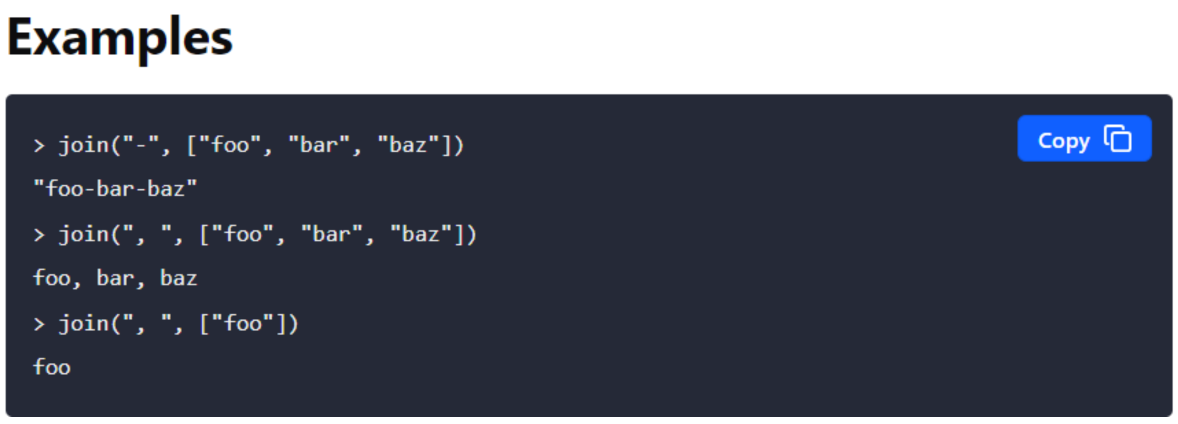
27. **terraform workspace list** — List your workspaces.
28. **terraform workspace select <workspace name>** — Select a specified workspace.
29. **terraform workspace new <workspace name>** — Create a new workspace with a specified name.
30. **terraform workspace delete <workspace name>** — Delete a specified workspace
31. **terraform output** — List all the outputs currently held in your state file. These are displayed by default at the end of a terraform apply, this command can be useful if you want to view them independently.
32. **terraform output vm1_public_ip** — List a specific output held in your state file
33. **terraform graph** — Produce a graph in DOT language showing the dependencies between objects in the state file. This can then be rendered by a program called Graphwiz (amongst others).
34. **terraform console** — Allow testing and exploration of expressions on the interactive console using the command line. e.g. 1+2

Terraform Functions:

1. **Join Function:**

   Syntax - join(separator, list)

   Examples –

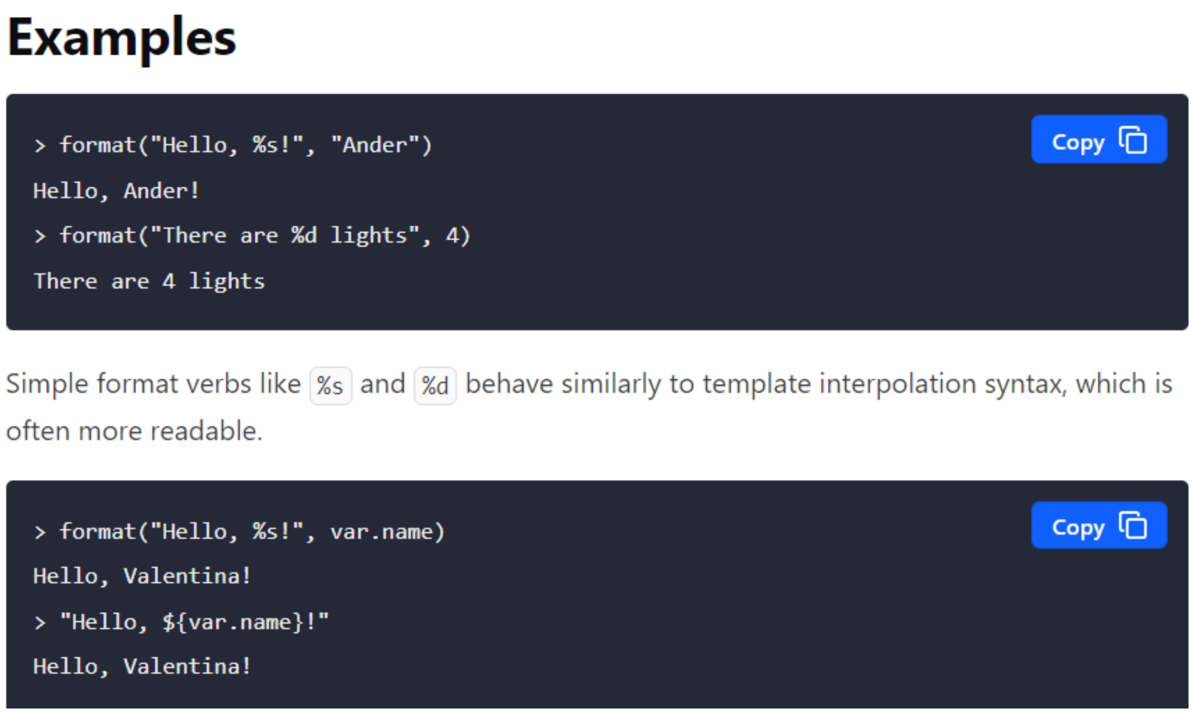


Examples

```
> join("-", ["foo", "bar", "baz"])
"foo-bar-baz"
> join(", ", ["foo", "bar", "baz"])
foo, bar, baz
> join(", ", ["foo"])
foo
```

2. **Format Function:**

   Syntax - format(spec, values...)



Examples

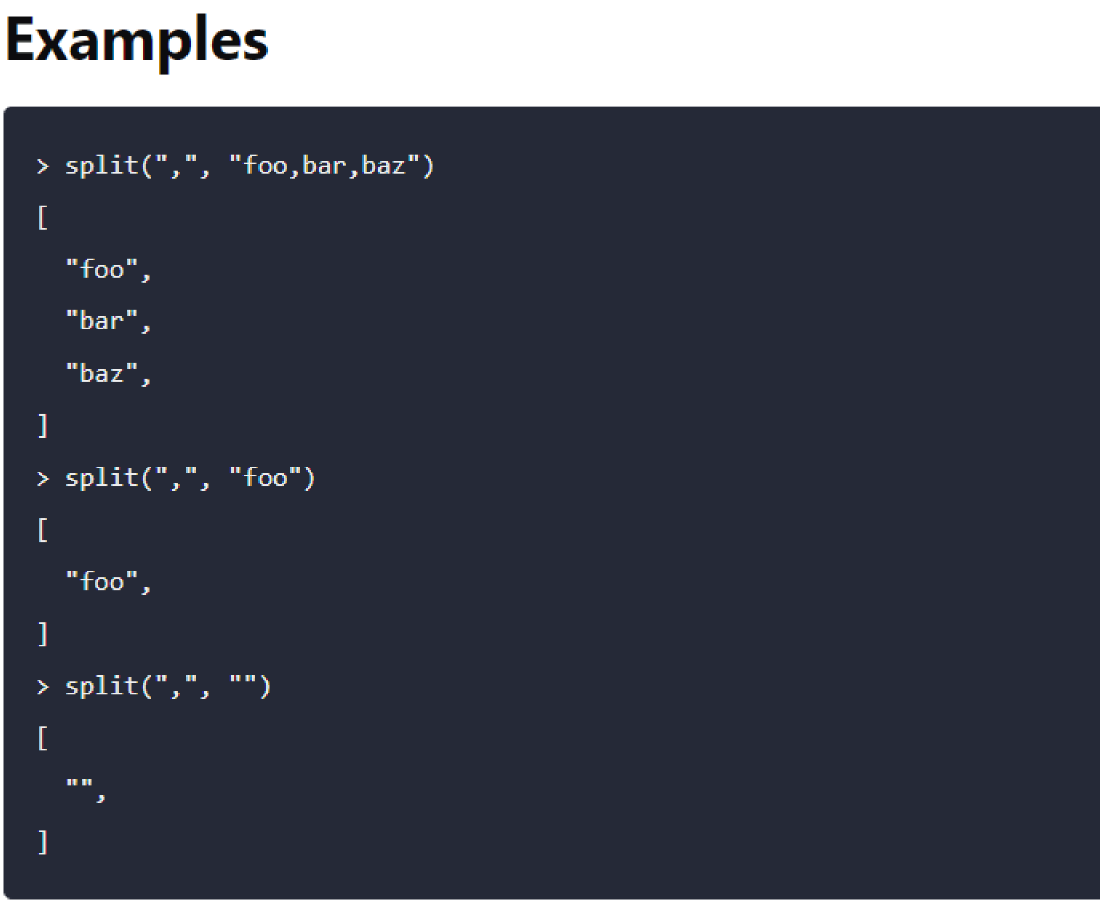
```
> format("Hello, %s!", "Ander")
Hello, Ander!
> format("There are %d lights", 4)
There are 4 lights
```

Simple format verbs like `%s` and `%d` behave similarly to template interpolation syntax, which is often more readable.

```
> format("Hello, %s!", var.name)
Hello, Valentina!
> "Hello, ${var.name}!"
Hello, Valentina!
```

3. **Split Function:**

Syntax – split (separator, string)



Examples

```
> split(",", "foo,bar,baz")
[
  "foo",
  "bar",
  "baz",
]
> split(",", "foo")
[
  "foo",
]
> split(",", "")
[
  "",
]
```

4. **Contains Function:**

   Syntax – contains (list, value)

Examples

```
> contains(["a", "b", "c"], "a")
true
> contains(["a", "b", "c"], "d")
false
```

5. **Element Function:**

   Syntax – element (list, index)

Examples

```
> element(["a", "b", "c"], 1)
b
```

6. **Length Function:**

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

Examples

```
> length([])
0
> length(["a", "b"])
2
> length({"a" = "b"})
1
> length("hello")
5
```

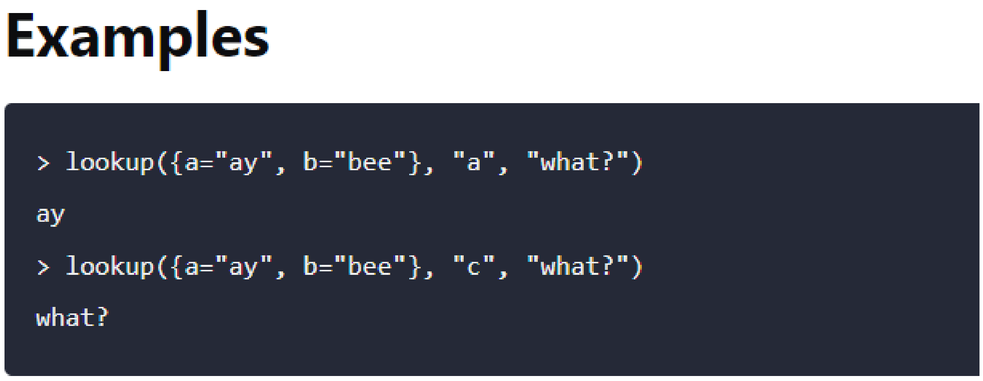
7. **Tolist Function:**

   tolist([a, b, c])

   The [...] brackets construct a tuple value, and then the tolist function then converts it to a list.

8. **Lookup Function:**

   Syntax:- lookup(map, key, default)



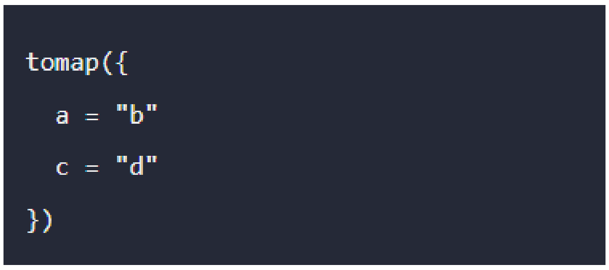
Examples

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

9. **Map Function**:

   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.

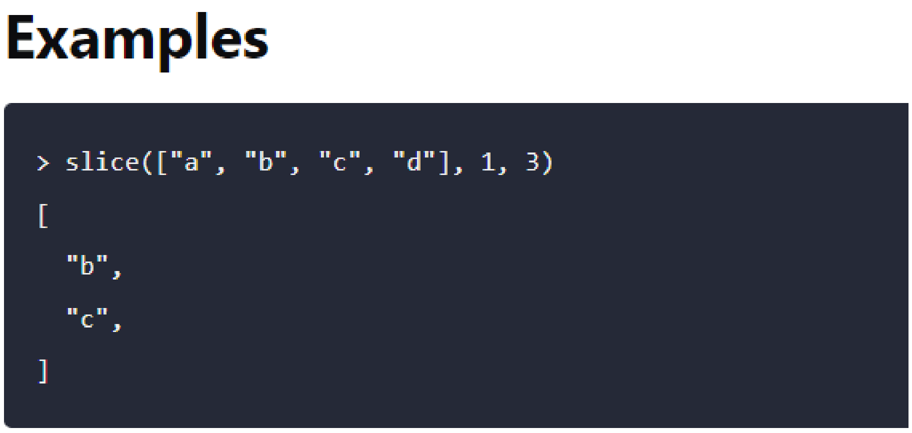
   To update an expression like map("a", "b", "c", "d"), write the following instead:



```
tomap({
  a = "b"
  c = "d"
})
```

10. **Slice Function:**

    Syntax – slice (list, startindex, endindex)



Examples

```
> slice(["a", "b", "c", "d"], 1, 3)
[
  "b",
  "c",
]
```