📌 Basic Terraform Commands

Command	Description
terraform init	Initializes a new or existing Terraform configuration.
terraform plan	Creates an execution plan to preview changes.
terraform apply	Applies the desired infrastructure changes.
terraform destroy	Destroys all resources managed by Terraform.
terraform show	Displays details about Terraform's state or execution plan.
terraform output	Shows values of defined output variables.

📌 Intermediate Terraform Commands

Command	Description
terraform state list	Lists all resources managed in Terraform state.
terraform state show <resource>	Shows details of a specific resource.
terraform state mv <old> <new>	Moves a resource within the state file.
terraform state rm <resource>	Removes a resource from Terraform state.
terraform import <resource> <id>	Imports an existing resource into Terraform.
terraform taint <resource>	Marks a resource for replacement on the next apply.
terraform untaint <resource>	Removes the taint from a resource.

📌 Advanced Terraform Commands

Command	Description
terraform fmt	Formats Terraform files to follow best practices.
terraform validate	Validates the syntax and configuration.
terraform graph	Generates a graph of the Terraform dependency structure.
terraform workspace new <name>	Creates a new Terraform workspace.
terraform workspace list	Lists available workspaces.

Command	Description
terraform workspace select <name>	Switches to a different workspace.
terraform refresh	Updates Terraform state with real infrastructure status.
terraform force-unlock <lock-id>	Forces Terraform to unlock the state.
terraform apply -target=<resource>	Applies changes to a specific resource.
terraform plan -destroy	Shows a destruction plan without applying it.
terraform apply -var="name=value"	Overrides variable values during execution.
terraform apply -parallelism=N	Limits concurrent operations.
terraform console	Opens an interactive Terraform console.
terraform debug	Enables debug mode for troubleshooting.
terraform test	Runs Terraform test cases (for Terraform 1.6+).
terraform login	Authenticates to Terraform Cloud.
terraform logout	Removes stored credentials for Terraform Cloud.

📌 Less Common Terraform Commands

Command	Description
terraform providers	Lists required providers.
terraform force-unlock <lock-id>	Unlocks Terraform state manually.
terraform providers lock	Locks provider versions in .terraform.lock..
terraform providers mirror <dir>	Mirrors provider binaries to a directory.
terraform workspace delete <name>	Deletes a Terraform workspace.
terraform providers schema	Outputs the JSON schema of providers.
terraform show -json	Outputs Terraform state in JSON format.
terraform output -json	Shows Terraform output in JSON format.
terraform state pull	Retrieves the current state as a JSON file.
terraform state push <statefile>	Overwrites the current Terraform state.

📌 Terraform Enterprise & Cloud Commands

Command	Description
terraform cloud	Manages Terraform Cloud settings.
terraform login	Logs into Terraform Cloud.
terraform logout	Logs out of Terraform Cloud.
terraform state pull	Retrieves the latest Terraform Cloud state.
terraform force-unlock	Unlocks Terraform Cloud’s state.

📌 Terraform Commands with Flags

🌟 Execution Options

Flag	Description
-auto-approve	Skips confirmation prompt.
-lock=false	Disables state locking (use with caution).
-parallelism=N	Sets parallel execution limits.

🌟 State Management

Flag	Description
-state=<path>	Specifies an alternative state file.
-state-out=<path>	Writes state to a new file.

🌟 Variable Management

Flag	Description
-var="key=value"	Assigns a variable value.
-var-file=<file>	Loads variable values from a file.

🚀 Terraform Workflow Example

```
terraform init
terraform plan -out=tfplan
terraform apply tfplan
terraform output
terraform destroy -auto-approve
```

Terraform Input Variables (variable)

📌 What are Input Variables?

Input variables allow users to **parameterize Terraform configurations** to make them reusable and dynamic.

🔹 Defining Input Variables

```
variable "instance_type" {
  description = "The type of AWS EC2 instance"
  type        = string
  default     = "t2.micro"
}
```

🔹 Using Input Variables in Resources

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
}
```

🔹 Passing Variables

✅ **Via CLI:**

```
terraform apply -var="instance_type=t3.micro"
```

✅ **Via .tfvars file:**

```
instance_type = "t3.medium"
terraform apply -var-file="terraform.tfvars"
```

🔹 Input Variable Attributes

Attribute	Description	Example
type	Defines the expected data type (string, number, bool, list, map, object)	type = list(string)
default	Sets a default value	default = "t2.micro"
description	Provides documentation for the variable	description = "Instance type"
sensitive	Hides sensitive values in Terraform output	sensitive = true

---

Terraform Output Variables (output)

📌 What are Output Variables?

Outputs allow Terraform to display **important values** after execution.

🔹 Defining an Output Variable

```
output "instance_public_ip" {
  description = "The public IP of the instance"
  value       = aws_instance.example.public_ip
}
```

🔹 Viewing Outputs

```
terraform output instance_public_ip
```

🔹 Output Variable Attributes

Attribute	Description	Example
value	Defines the value to return	value = aws_instance.example.public_ip
description	Describes the output	description = "Public IP"
sensitive	Hides output from logs	sensitive = true

Terraform Resources (resource)

📌 What are Resources?

A **resource** defines a **real-world infrastructure component** like an EC2 instance, an S3 bucket, or a database.

🔹 Defining a Resource

```
resource "aws_instance" "example" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
}
```

🔹 Common Resource Attributes

Attribute	Description	Example
id	Unique identifier of the resource	aws_instance.example.id
public_ip	Public IP address (for instances)	aws_instance.example.public_ip
tags	Key-value pairs for labeling	tags = { Name = "MyInstance" }

🔹 Using Resource Attributes

```
output "instance_id" {
  value = aws_instance.example.id
}
```

Terraform Data Sources (data)

📌 What are Data Sources?

Data sources **fetch information** about existing resources **without modifying them**.

🔹 Example: Fetching an Existing AMI

```
data "aws_ami" "latest" {
  most_recent = true
  owners      = ["amazon"]
}
resource "aws_instance" "example" {
  ami           = data.aws_ami.latest.id
  instance_type = "t2.micro"
}
```

🔹 Common Data Source Attributes

Attribute	Description	Example
id	The unique identifier of the data source	data.aws_ami.latest.id
tags	Filters resources by tags	tags = { Name = "my-instance" }

Terraform Providers (provider)

📌 What are Providers?

Providers **allow Terraform to interact with cloud platforms** like AWS, Azure, or GCP.

🔹 Example: AWS Provider

```
provider "aws" {
  region = "us-east-1"
}
```

🔹 Common Provider Attributes

Attribute	Description	Example
region	Defines the cloud region	region = "us-east-1"
profile	Specifies the AWS profile	profile = "default"
version	Defines the provider version	version = "~> 3.0"

Terraform Modules (module)

📌 What are Modules?

Modules help **organize Terraform code into reusable components**.

🔹 Example: Using a Module

```
module "network" {
  source   = "./network_module"
  vpc_name = "my-vpc"
}
```

🔹 Common Module Attributes

Attribute	Description	Example
source	Specifies the module location	source = "./vpc"
variables	Passes input values	vpc_name = "my-vpc"

Terraform Locals (locals)

📌 What are Locals?

Locals **store reusable values** that don't need input variables.

🔹 Example: Using Locals

```
locals {
  env = "dev"
}
resource "aws_s3_bucket" "example" {
  bucket = "my-bucket-${local.env}"
}
```

🔹 Common Local Attributes

Attribute	Description	Example
local.name	Reference local variable	local.env

Terraform Functions

Terraform **provides built-in functions** for dynamic configurations.

🔹 Examples of Functions

Function	Description	Example
length	Returns list length	length(["a", "b"]) → 2
join	Joins strings	join("-", ["dev", "us"]) → "dev-us"
upper	Converts string to uppercase	upper("dev") → "DEV"

🚀 Summary

Attribute Type	Description	Example
Input Variables (var)	Define dynamic inputs	var.instance_type
Output Variables (output)	Return values after execution	output "instance_id" { value = aws_instance.example.id }
Resources (resource)	Define infrastructure components	resource "aws_instance" "example"
Data Sources (data)	Fetch existing resources	data "aws_ami" "latest"
Providers (provider)	Configure cloud platforms	provider "aws"
Modules (module)	Reusable Terraform code	module "network"
Locals (local)	Store computed values	local.env = "dev"

📌 Types of Loops in Terraform

Loop Type	Used In	Description
count	Resources	Simple iteration using a number (integer)
for_each	Resources, Modules, Variables	Iterates over maps & sets (key-value pairs)
for	Lists, Maps	Iterates inside variables & outputs
dynamic	Nested Blocks in Resources	Generates multiple blocks dynamically

---

count Loop

📌 What is count?

- The simplest way to create **multiple instances** of a resource.
- **Uses an integer value** to define how many times to create a resource.

🔹 **Example: Creating Multiple EC2 Instances**

```
resource "aws_instance" "example" {
 count       = 3
 ami        = "ami-12345678"
 instance_type = "t2.micro"
 tags = {
  Name = "Instance-${count.index}"
 }
}
```

✅ **How it Works:**

- Creates **3 instances** (count = 3).
- count.index is **0-based**, so instances will be:
  - Instance-0
  - Instance-1
  - Instance-2

🔹 **When to Use count**

✔ When you **know the exact number** of resources to create.
✔ Works **best for simple lists** but **not maps**.

---

for_each Loop

📌 **What is for_each?**

- Used to iterate over **lists, maps, or sets**.
- Works **better than count** for creating resources with unique identifiers.

🔹 **Example: Creating Multiple EC2 Instances from a Map**

```
resource "aws_instance" "example" {
 for_each = {
  "server1" = "t2.micro"
  "server2" = "t3.small"
 }
 ami        = "ami-12345678"
 instance_type = each.value
 tags = {
  Name = each.key
 }
}
```

✅ **How it Works:**

- Creates **two instances** with **different instance types**:
  - server1 → t2.micro
  - server2 → t3.small

🔹 **When to Use for_each**

✔ When working with **maps** or **sets**.
✔ When you need **unique resource names** instead of index numbers.
✔ Works well for **dynamic scaling**.

---

for Loop in Variables

📌 What is for in Variables?

- Allows transforming **lists and maps**.
- Can be used in **variables, locals, and outputs**.

🔹 Example: Transforming a List

```
variable "names" {
  default = ["Alice", "Bob", "Charlie"]
}
output "greetings" {
  value = [for name in var.names : "Hello, ${name}!"]
}
```

✅ **Output:**

```
greetings = ["Hello, Alice!", "Hello, Bob!", "Hello, Charlie!"]
```

🔹 Example: Filtering a List

```
variable "numbers" {
  default = [1, 2, 3, 4, 5, 6]
}
output "even_numbers" {
  value = [for num in var.numbers : num if num % 2 == 0]
}
```

✅ **Output:**

```
even_numbers = [2, 4, 6]
```

🔹 When to Use for

✔️ When you need to **transform lists or maps** dynamically.
✔️ When filtering or modifying lists inside **locals, outputs, or variables**.

---

dynamic Block

📌 What is dynamic?

- Used inside **resource blocks** to create multiple **nested blocks** dynamically.

🔹 Example: Creating Multiple Security Group Rules

```
resource "aws_security_group" "example" {
  name = "example-sg"
  dynamic "ingress" {
    for_each = [80, 443, 22] # Iterate over ports
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

✅ **How it Works:**

- Creates **three ingress rules** for **ports 80, 443, and 22**.

🔹 **When to Use dynamic**

✔ When a **nested block** needs to be **repeated multiple times**.
✔ Common for **security groups, IAM policies, and storage rules**.

🚀 Choosing the Right Loop

Loop Type	Best For	Works With	Example
count	Simple resource duplication	Number	count = 3
for_each	Unique resources with different values	Map/Set	for_each = { server1 = "t2.micro" }
for	Modifying lists or maps	Lists, Maps	value = [for i in list : i * 2]
dynamic	Repeating nested blocks inside resources	Lists, Maps	Security rules, IAM policies

---

🎯 Conclusion

Terraform loops **simplify infrastructure provisioning** by reducing repetition and making code more **dynamic and scalable**.

🔹 Use count for simple **number-based iterations**.
🔹 Use for_each for **maps and unique identifiers**.
🔹 Use for to **modify lists/maps** dynamically.
🔹 Use dynamic for **nested repeated blocks**.

Why Use Remote State?

✅ **Collaboration:** Multiple team members can work on the same infrastructure.
✅ **Locking Mechanism:** Prevents multiple users from modifying the same state simultaneously.
✅ **Backup & Recovery:** Avoids losing state files due to accidental deletion.
✅ **Version Control:** Some backends (like Terraform Cloud) provide state history.

📌 Terraform Core Concepts Explained

Terraform is a powerful **Infrastructure as Code (IaC)** tool that provides automation for infrastructure provisioning and management. Below are the key **Terraform concepts** you need to understand.

Terraform State (terraform.tfstate)

📌 What is Terraform State?

- Terraform keeps track of **infrastructure resources** using a **state file** (terraform.tfstate).

- It maps Terraform **configuration files** to real-world infrastructure.
- Stored **locally** by default but can be stored **remotely** (S3, Terraform Cloud, etc.).

🔹 Example of a State File (Snippet)

```
{
 "version": 4,
 "terraform_version": "1.3.0",
 "resources": [
  {
   "type": "aws_instance",
   "name": "example",
   "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
   "instances": [
    {
     "attributes": {
      "id": "i-0abcd1234efgh5678",
      "instance_type": "t2.micro",
      "ami": "ami-12345678"
     }
    }
   ]
  }
 ]
}
```

✅ Why is Terraform State Important?

- Tracks **existing resources**.
- Enables **dependency management**.
- Supports **remote state storage** for collaboration.

✅ Common State Commands

```
terraform state list     # List resources in state
terraform state show aws_instance.example   # Show details
terraform state rm aws_instance.example    # Remove resource from state
terraform state mv aws_instance.old aws_instance.new   # Rename a resource
```

Terraform State Locking

📌 What is State Locking?

State locking prevents **multiple Terraform executions** from modifying the state at the same time.

🔹 How to Enable State Locking?

Use AWS S3 with DynamoDB Locking

```
terraform {
 backend "s3" {
  bucket       = "my-terraform-state"
  key          = "terraform.tfstate"
  region       = "us-east-1"
  dynamodb_table = "terraform-lock"
 }
}
```

Create a Locking Table in DynamoDB

```
aws dynamodb create-table \
 --table-name terraform-lock \
 --attribute-definitions AttributeName=LockID,AttributeType=S \
 --key-schema AttributeName=LockID,KeyType=HASH \
 --billing-mode PAY_PER_REQUEST
```

🔹 Unlocking the State

If Terraform crashes while holding a lock, **force unlock it**:

```
terraform force-unlock LOCK_ID
```

✅ Why Use State Locking?

- Prevents **simultaneous state modifications**.
- Reduces risk of **state corruption** in **team environments**.

Terraform Backends

📌 What are Backends?

A backend defines **where Terraform stores state** and how it interacts with remote services.

🔹 Types of Terraform Backends

Backend	**Supports Locking?**	**Best For**
Local (Default)	❌ No	Local development
AWS S3 + DynamoDB	✅ Yes	AWS-based collaboration
Terraform Cloud	✅ Yes	Team collaboration
Azure Blob Storage	✅ Yes	Azure-based collaboration
Google Cloud Storage	❌ No	GCP-based collaboration

🔹 Example: Using an S3 Backend

```
terraform {
 backend "s3" {
  bucket = "my-terraform-state"
  key    = "terraform.tfstate"
  region = "us-east-1"
 }
}
```

✅ Benefits of Remote Backends:

- **Collaboration** (Teams can work on Terraform together).
- **State locking** (Prevents conflicts).
- **Security** (State is encrypted and backed up).

Terraform Workspaces

📌 What are Workspaces?

Workspaces allow Terraform to **manage multiple environments** (e.g., dev, staging, prod) in a **single configuration**.

🔹 **Managing Workspaces**

```
terraform workspace new dev     # Create a new workspace
terraform workspace select dev  # Switch to 'dev' workspace
terraform workspace list        # List all workspaces
terraform workspace delete dev  # Delete a workspace
```

✅ **Use Cases**

- Isolate environments **(dev, test, prod)**.
- Manage **different configurations** without duplicating code.

Terraform Provisioners

📌 **What are Provisioners?**

Provisioners **run scripts** inside resources **after creation**.

🔹 **Example: Running a Shell Script on an EC2 Instance**

```
resource "aws_instance" "example" {
 ami          = "ami-12345678"
 instance_type = "t2.micro"

 provisioner "remote-exec" {
  inline = [
   "sudo apt update",
   "sudo apt install nginx -y"
  ]
 }
}
```

✅ **Use Cases**

- Running **configuration scripts**.
- Installing **packages**.
- Bootstrapping **custom settings**.

Terraform Lifecycle Rules

📌 **What are Lifecycle Rules?**

Controls how Terraform **manages resources** over time.

🔹 **Example: Prevent Resource Deletion**

```
resource "aws_s3_bucket" "example" {
 bucket = "my-bucket"

 lifecycle {
  prevent_destroy = true
 }
}
```

✅ **Lifecycle Options**

Attribute	Description
create_before_destroy	Creates a new resource before destroying the old one
prevent_destroy	Prevents accidental resource deletion
ignore_changes	Ignores specific attribute changes

🔟 Summary of Key Terraform Concepts

Concept	Description
State File	Tracks Terraform-managed resources
State Locking	Prevents concurrent modifications
Backends	Stores Terraform state remotely
Workspaces	Isolates multiple environments (dev, prod)
Providers	Connects Terraform to cloud platforms

Concept	Description
Modules	Reusable Terraform configurations
Provisioners	Runs scripts on infrastructure
Variables	Makes configurations flexible
Outputs	Retrieves values from Terraform
Lifecycle Rules	Controls resource behavior

Command: terraform version

Terraform Version Example v1.5.2

MAJOR (X)	Breaking changes that require updates to configurations.
MINOR (Y)	Backward-compatible feature updates.
PATCH (Z)	Bug fixes and security updates.

Latest: 1.6 and older are 0.15, 0.14 and so on

```
terraform init -upgrade
terraform plan
terraform apply
```