TASKS ON TERRAFORM

- ❖ **NAME: S.MUZZAMMIL HUSSAIN**
- ❖ **DATE: 25/07/2025**
- ❖ **BATCH: 11**
- ❖ **NO.OF TASKS: 1**

Task 1. Explain Terraform functions

Terraform Functions:

Terraform offers a comprehensive set of built-in functions, allowing for efficient data manipulation, transformation, and dynamic infrastructure-as-code workflows. Using these functions enables smoother configuration management and streamlined automation, especially for complex scenarios involving variable inputs and dynamic resource creation.

1. Numeric (Math) Functions

These functions are built-in and let us perform common mathematical operations directly within Terraform configurations, enabling dynamic and flexible infrastructure definitions.

- **min**: Returns the smallest number among provided arguments.
- **max**: Returns the largest number among provided arguments.
- **pow**: Raises a base number to the power of an exponent number.
- **abs**: Returns the absolute value (non-negative) of the given number.
- **sum**: Adds together all numbers in a given list.
- **ceil**: Rounds a decimal number up to the nearest whole integer.
- **floor**: Rounds a decimal number down to the nearest whole integer.

Enable calculations and numeric processing directly in HCL expressions:

Function	Description	Example usage
min	Smallest value from a list	min(4, 7, 2) → 2
max	Largest value from a list	max(4, 7, 2) → 7
pow	Exponentiation	pow(2, 3) → 8
abs	Absolute value	abs(-5) → 5
sum	Total of all list elements	sum([2, 3,9
ceil	Ceiling to nearest integer	ceil(3.2) → 4
floor	Floor to nearest integer	floor(3.8) → 3

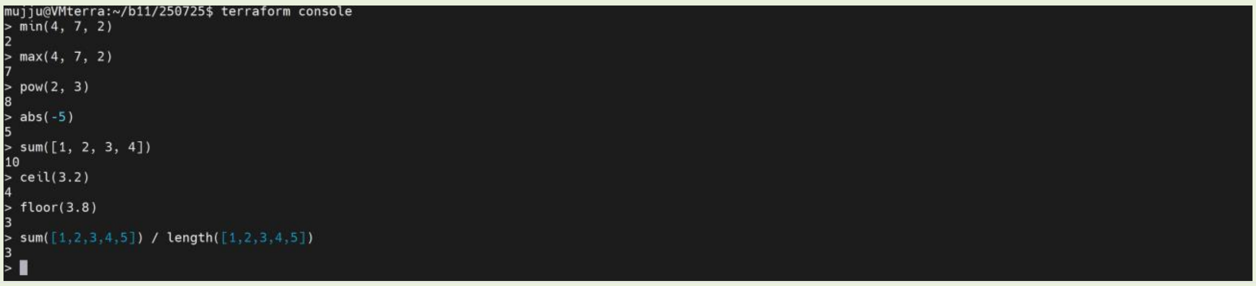
Initialize Terraform

```
terraform init
```

Start the Terraform Console

Launch the console with:

terraform console



```
mujju@VMterra:~/b11/250725$ terraform console
> min(4, 7, 2)
2
> max(4, 7, 2)
7
> pow(2, 3)
8
> abs(-5)
5
> sum([1, 2, 3, 4])
10
> ceil(3.2)
4
> floor(3.8)
3
> sum([1,2,3,4,5]) / length([1,2,3,4,5])
3
> 
```

2. String Functions

Here is an explanation of the Terraform string functions **upper, lower, join, replace, and substr**, describing what each does and how they are used in Terraform configurations:

1. upper

- **Purpose:** Converts all characters in a string to uppercase.
- **Use case:** Useful for normalizing strings to uppercase for resource names, tags, or comparison.
- **Example:** Converts "hello" to "HELLO".

2. lower

- **Purpose:** Converts all characters in a string to lowercase.
- **Use case:** Useful for ensuring consistent lowercase string values, such as hostnames or labels.
- **Example:** Converts "WORLD" to "world".

3. join

- **Purpose:** Concatenates a list of strings into a single string with a specified separator between elements.
- **Use case:** Helpful to build strings from list elements, such as constructing comma-separated values, paths, or tags.
- **Example:** Joins ["a", "b", "c"] with separator "," to form "a,b,c".

4. replace

- **Purpose:** Replaces all occurrences of a substring within a given string with another substring.
- **Use case:** Useful for modifying strings by substituting characters or substrings, such as replacing spaces with underscores.
- **Example:** Replaces "_" with "-" in "foo_bar" to get "foo-bar".

5. substr

- **Purpose:** Extracts a substring from a string, given a starting index and length.
- **Use case:** Helpful for parsing strings or extracting meaningful parts like dates or identifiers.
- **Indexing:** Index starts at 0 for the first character.
- **Example:** Extracts 3 characters starting at index 2 from "abcdef" to get "cde".

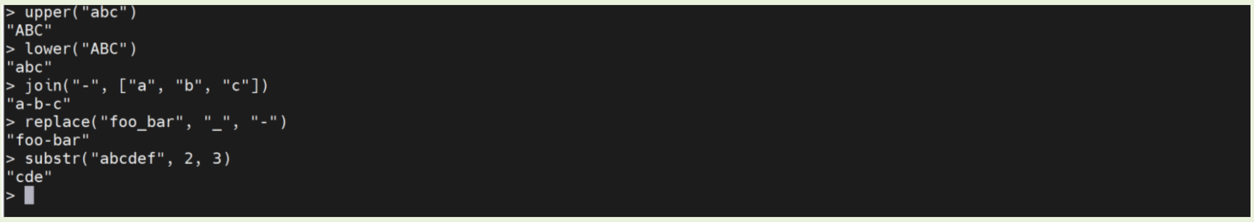
Manipulate and process strings for cleaner resource and variable handling.

Function	Description	Example usage
upper	Converts string to uppercase	upper("abc") → "ABC"
lower	Converts string to lowercase	lower("ABC") → "abc"
join	Joins list elements with separator	join("-", ["a", "b", "c"]) → "a-b-c"
replace	Replace substring(s) within a string	replace("foo_bar", "_", "-") → "foo-bar"
substr	Substring extraction	substr("abcdef", 2, 3) → "cde"

Start the Terraform Console

Launch the console with:

terraform console



```
> upper("abc")
"ABC"
> lower("ABC")
"abc"
> join("-", ["a", "b", "c"])
"a-b-c"
> replace("foo_bar", "_", "-")
"foo-bar"
> substr("abcdef", 2, 3)
"cde"
> 
```

3. Filesystem Functions

Here is an explanation of the Terraform **abspath** and **file** functions:

(i). abspath

- **Purpose:**
  Converts a given filesystem path (string) into an absolute path.
  If the input path is relative, abspath joins it with the current working directory to produce the full absolute path.
- **Use case:**
  Useful to normalize paths in Terraform configurations so that filesystem references remain consistent regardless of where the configuration is applied or run. It helps avoid subtle diffs caused by relative paths running from different directories or machines.
- **Example:**
  If you provide a relative path like "../data/config.yaml", abspath converts it to something like "/home/user/project/data/config.yaml" (depending on your current directory)
- **Notes:**
  Usually used with file to load file contents reliably or in provisioner/connection blocks that require absolute paths.

(ii). file

- **Purpose:**
  Reads the content of a file given its path and returns the file content as a string.
- **Use case:**
  Often used to read local configuration files, scripts, certificates, or templates into Terraform for use in resource arguments or provisioners.
- **Example:**
  Given a file config.txt containing "Hello World", the expression file("config.txt") returns the string "Hello World".
- **Notes:**
  The file path can be relative or absolute. Using abspath is recommended to avoid any ambiguity.

Summary

Function	Description
abspath	Converts a relative filesystem path to an absolute path based on the current working directory.
file	Reads the contents of the specified file and returns it as a string.

Work with files and file paths on the local machine where Terraform is executed.

4. Date/Time Functions

Here is a detailed explanation of the Terraform functions **timestamp** and **formatdate**:

(i). timestamp

- **Purpose:**
  Returns the current date and time as a UTC timestamp string in the RFC 3339 format (e.g., "2025-07-25T14:17:30Z").
- **Behavior:**
  The value changes every second because it captures the exact current time at the moment the function is evaluated (usually during terraform apply). Because of this continuous change, using timestamp() directly in resource attributes may cause Terraform to detect a change every run, which is often undesirable. It is typically better used for local values, outputs, or with lifecycle rules like ignore_changes if used in resources.
- **Use cases:**
  - Capturing resource creation time.
  - Adding timestamps to tags or metadata.
  - Generating time-based unique values on resource creation.
- **Example output:**

  2025-07-25T14:17:30Z

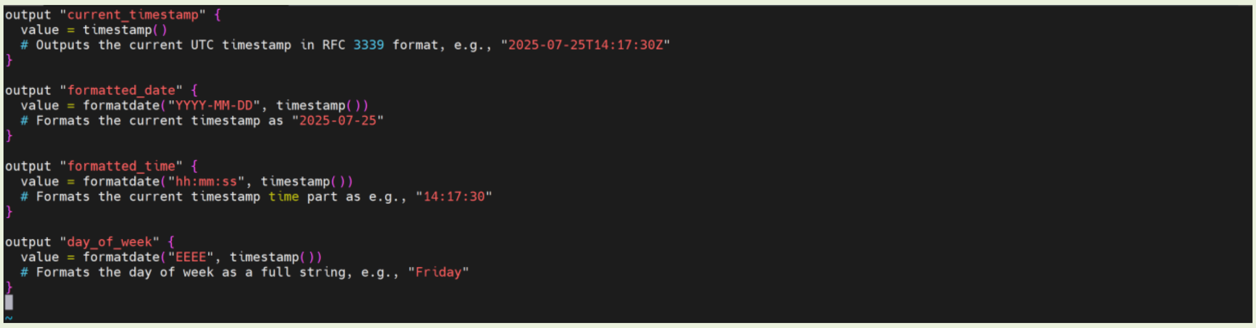
(ii). formatdate

- **Purpose:**
  Converts a given timestamp (usually in RFC 3339 format, such as the output of timestamp()) into a string formatted according to a specified date/time pattern. This allows you to display timestamps in a human-friendly or system-specific format.
- **Input:**
  - A **format string** describing the desired output format (e.g., "YYYY-MM-DD", "hh:mm:ss", "EEEE" for day name).
  - A **timestamp** string to format (often the string returned by timestamp() or other functions returning timestamp strings).
- **Functionality:**
  Uses date/time format specifiers (similar to many programming languages) to output parts of the date/time such as year (YYYY), month (MM), day (DD), hour (hh), minute (mm), second (ss), and day of the week (EEEE).
- **Use cases:**
  - Formatting timestamps for tags or human-readable logs.
  - Extracting specific date or time components.
  - Adapting timestamp formats to requirements of external systems or outputs.
- **Example outputs:**
  If timestamp() returns "2025-07-25T14:17:30Z":
  - formatdate("YYYY-MM-DD", timestamp()) → "2025-07-25"
  - formatdate("hh:mm:ss", timestamp()) → "14:17:30"
  - formatdate("EEEE", timestamp()) → "Friday"

Example:

Create a **main.tf** file using **vi main.tf** add the following configuration

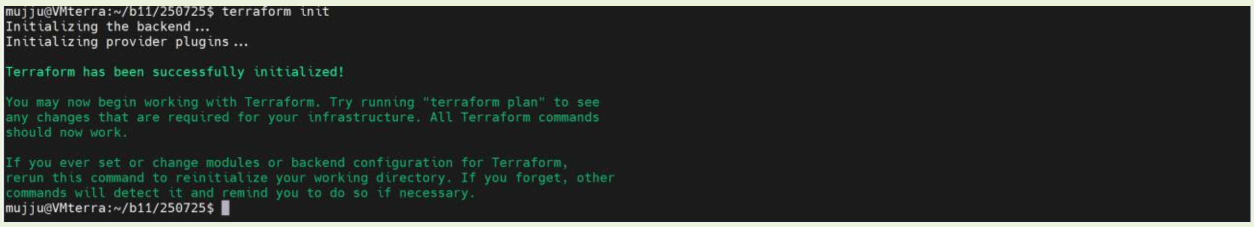
```
output "current_timestamp" {

  value = timestamp()

  # Outputs the current UTC timestamp in RFC 3339 format, e.g., "2025-07-25T14:17:30Z"

}

output "formatted_date" {

  value = formatdate("YYYY-MM-DD", timestamp())

  # Formats the current timestamp as "2025-07-25"

}

output "formatted_time" {

  value = formatdate("hh:mm:ss", timestamp())

  # Formats the current timestamp time part as e.g., "14:17:30"

}

output "day_of_week" {

  value = formatdate("EEEE", timestamp())

  # Formats the day of week as a full string, e.g., "Friday"

}
```



```
output "current_timestamp" {
  value = timestamp()
  # Outputs the current UTC timestamp in RFC 3339 format, e.g., "2025-07-25T14:17:30Z"
}

output "formatted_date" {
  value = formatdate("YYYY-MM-DD", timestamp())
  # Formats the current timestamp as "2025-07-25"
}

output "formatted_time" {
  value = formatdate("hh:mm:ss", timestamp())
  # Formats the current timestamp time part as e.g., "14:17:30"
}

output "day_of_week" {
  value = formatdate("EEEE", timestamp())
  # Formats the day of week as a full string, e.g., "Friday"
}

~
```

Initialize Terraform

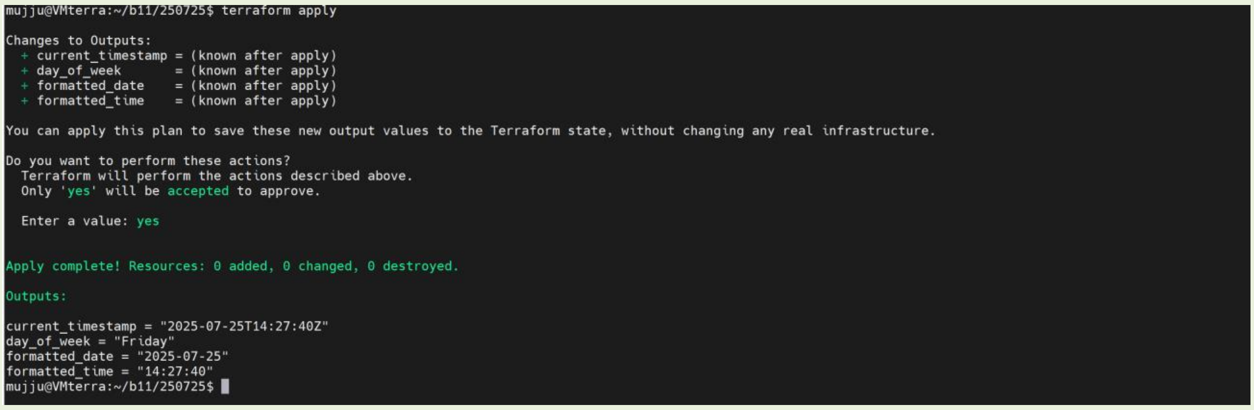
terraform init



```
mujju@VMterra:~/b11/250725$ terraform init
Initializing the backend...
Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
mujju@VMterra:~/b11/250725$
```

Run Terraform apply

terraform apply



```
mujju@VMterra:~/b11/250725$ terraform apply

Changes to Outputs:
  + current_timestamp = (known after apply)
  + day_of_week       = (known after apply)
  + formatted_date    = (known after apply)
  + formatted_time    = (known after apply)

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes


Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

current_timestamp = "2025-07-25T14:27:40Z"
day_of_week = "Friday"
formatted_date = "2025-07-25"
formatted_time = "14:27:40"
mujju@VMterra:~/b11/250725$ 
```

Key Terraform Outputs Displayed

- **current_timestamp**:
  Displays the UTC current date and time in RFC 3339 format (2025-07-25T14:27:40Z).
  Generated using the timestamp() function.
- **day_of_week**:
  Displays the full name of the current day, in this case, "Friday".
  Generated using formatdate("EEEE", timestamp()).
- **formatted_date**:
  Shows the date portion in the format "YYYY-MM-DD", resulting in "2025-07-25".
  Generated using formatdate("YYYY-MM-DD", timestamp()).
- **formatted_time**:
  Shows only the time portion in "hh:mm:ss" format, here "14:27:40".
  Generated using formatdate("hh:mm:ss", timestamp()).

Important Notes

- The output of timestamp() cannot be predicted during the plan phase because it is generated at apply time.
- timestamp() should be used cautiously in resource attributes to avoid unnecessary diffs.
- Combine timestamp() and formatdate() to generate timestamps in formats suitable for your use case.
- The format strings in formatdate() follow a specific pattern.

5. Collection Functions (Advanced Datatypes)

Empower efficient operations on lists, sets, and maps, crucial for complex infrastructure models.

(i) length

- **Purpose:** Returns the number of elements in a list, map, set, or the number of characters in a string.
- **Use Case:** Useful for resource counts, validations, and dynamic logic.

(ii) toset

- **Purpose:** Converts a list to a set, which automatically removes duplicates and has no defined order.
- **Use Case:** Ensures all values are unique for set-based operations or when removing duplicates is required.

(iii) sort

- **Purpose:** Returns a new list with all values from the input list, sorted in lexicographical (alphabetical or numeric) order.
- **Use Case:** Standardizes ordering for consistent outputs or comparisons.

(iv) lookup

- **Purpose:** Searches for a specified key within a map and returns its value, or a default value if the key is not present.
- **Use Case:** Safe retrieval of map values, avoiding errors if the key is missing.

(v) contains

- **Purpose:** Checks whether a certain value exists within a list, set, or map keys, returning true or false.
- **Use Case:** Conditional logic, filtering, or validation to confirm the presence of elements.

(vi) distinct

- **Purpose:** Returns a new list with duplicate elements removed, preserving the first appearance order.
- **Use Case:** Deduplicates values from a list for unique processing.

(vii) concat

- **Purpose:** Merges two or more lists into a single list.
- **Use Case:** Combines multiple sources of values for batch processing or dynamic resource arguments.

(viii) sum

- **Purpose:** Calculates the total by adding all the numbers in a list together.
- **Use Case:** Totals for capacity planning, resource sizing, or reporting.

These functions are core to managing and transforming complex variables and outputs in infrastructure as code, making your Terraform code flexible, concise, and reliable.

Example:

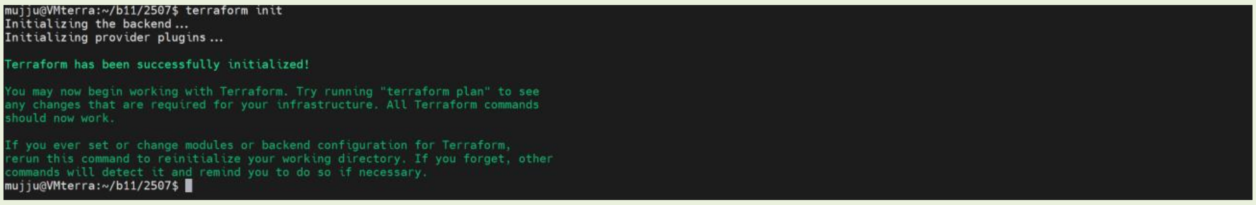
Create a **collections.tf** file using **vi collections.tf** add the following configuration:

```
output "length_of_list" {

  value = length(["x", "y", "z"])

}

output "length_of_map" {

  value = length({a = 1, b = 2})

}

output "length_of_string" {

  value = length("hello")

}
```

```
output "toset_example" {

  value = toset(["apple", "banana", "apple"])

}

output "sort_example" {

  value = sort(["pear", "apple", "banana"])

}

output "lookup_existing_key" {

  value = lookup({foo = "bar", baz = "qux"}, "foo", "default")

}

output "lookup_missing_key" {

  value = lookup({foo = "bar"}, "baz", "default")

}

output "contains_in_list" {

  value = contains(["a", "b", "c"], "b")

}

output "contains_in_map_keys" {

  value = contains(keys({a=1, b=2}), "b")

}

output "distinct_example" {

  value = distinct(["dog", "cat", "dog", "bird", "cat"])

}

output "concat_example" {

  value = concat(["a", "b"], ["c"], ["d", "e"])

}

output "sum_example" {

  value = sum([10, 20, 30])

}
```

Initialize Terraform

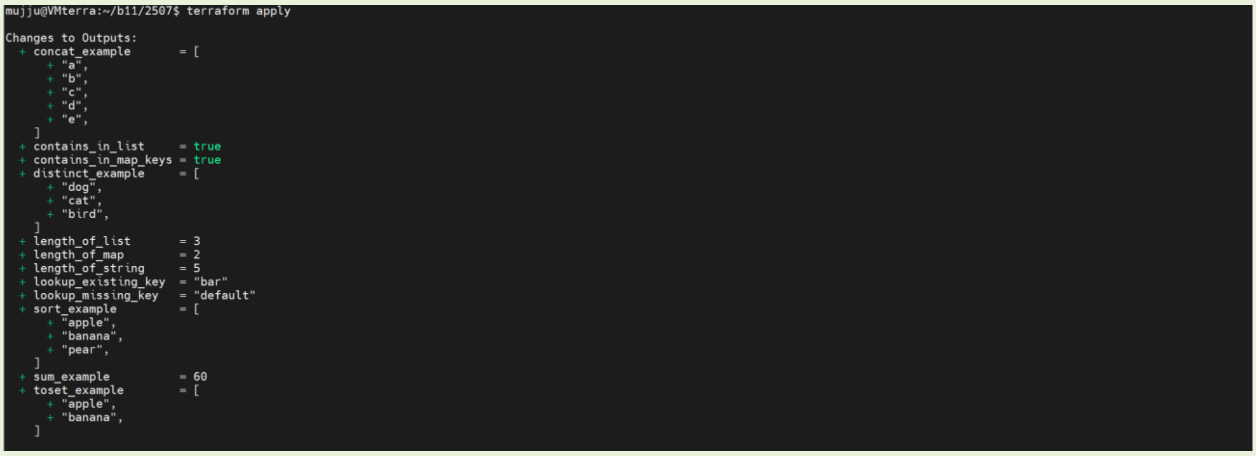
terraform init



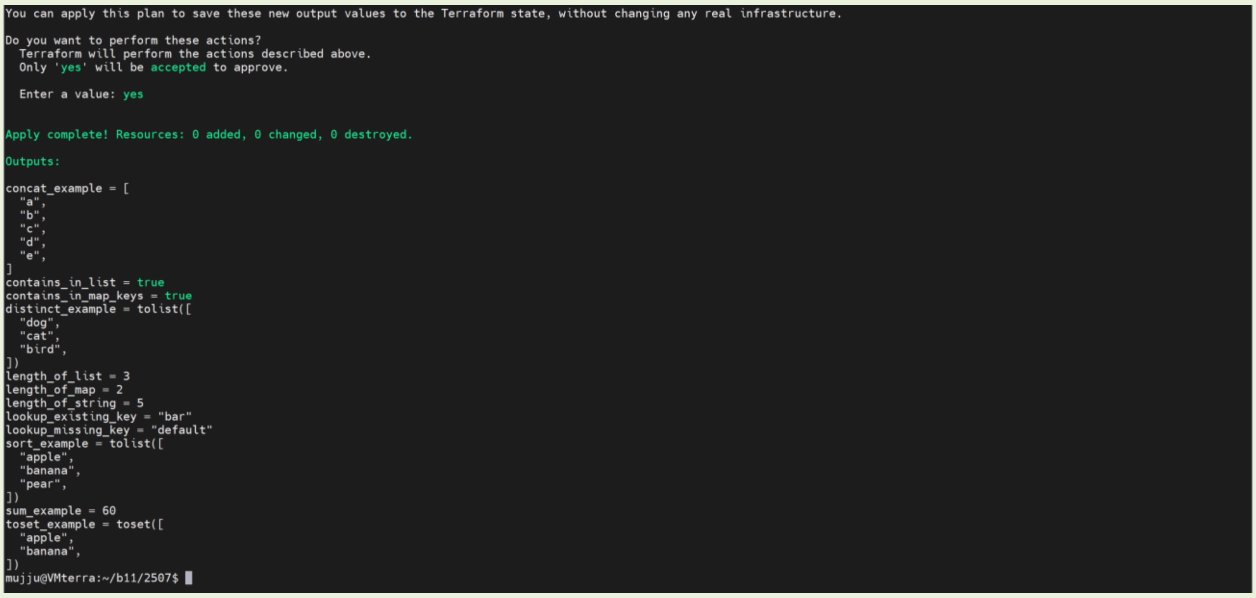
```
mujju@VMterra:~/b11/2507$ terraform init
Initializing the backend...
Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
mujju@VMterra:~/b11/2507$
```

Run Terraform apply

terraform apply



```
mujju@VMterra:~/b11/2507$ terraform apply

Changes to Outputs:
  + concat_example       = [
      + "a",
      + "b",
      + "c",
      + "d",
      + "e",
    ]
  + contains_in_list     = true
  + contains_in_map_keys = true
  + distinct_example     = [
      + "dog",
      + "cat",
      + "bird",
    ]
  + length_of_list       = 3
  + length_of_map        = 2
  + length_of_string     = 5
  + lookup_existing_key  = "bar"
  + lookup_missing_key   = "default"
  + sort_example         = [
      + "apple",
      + "banana",
      + "pear",
    ]
  + sum_example          = 60
  + toset_example        = [
      + "apple",
      + "banana",
    ]
```



```
You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes


Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

concat_example = [
  "a",
  "b",
  "c",
  "d",
  "e",
]
contains_in_list = true
contains_in_map_keys = true
distinct_example = tolist([
  "dog",
  "cat",
  "bird",
])
length_of_list = 3
length_of_map = 2
length_of_string = 5
lookup_existing_key = "bar"
lookup_missing_key = "default"
sort_example = tolist([
  "apple",
  "banana",
  "pear",
])
sum_example = 60
toset_example = toset([
  "apple",
  "banana",
])
mujju@VMterra:~/b11/2507$
```

Best Practices & Additional Execution Capabilities

- **Combining Functions:** Nest functions (e.g., upper(join("-", var.names))) for powerful data transformation.
- **Error Handling:** Use functions like can() and try() for safe, robust configuration logic.
- **Dynamic Resource Creation:** Pair with for expressions, count, or for_each for dynamic resource and data blocks.
- **Smoothing Execution:** Use collection and string functions for dynamic naming, resource configuration, and flexible automation based on input variables or external files.