Terrascript

Release 0.9.0

Feb 05, 2022

Contents:

CHAPTER 1

Introduction

1.1 Looking for more contributors

IF you feel that this project is useful to you, please consider contributing some of your time towards improving it! For more details on contributions, please have a look at CONTRIBUTORS.md and DEVELOPMENT.md.

1.2 About

Python-Terrascript is a Python package for generating Terraform configurations in JSON format.

Creating Terraform through a Python script offers a degree of flexibility superior to writing Terraform configurations by hand.

- Control structures like `if`/`else`, `for`/`continue`/`break` or `try`/`except`/`finally`.
- More string methods.
- Python functions may be used as an alternative to Terraform Modules.
- Access to the Python Standard Library and third-party packages.

1.3 Compatibility

1.3.1 Terraform releases

Terraform 0.12 introduced some changes to how it deals with configuration files in JSON format. This is reflected in Terrascript by currently having separate releases for Terraform 0.12 and Terraform 0.11. Earlier releases of Terraform are not supported.

Terraform	Terrascript	Notes
0.13.x	0.9.x	Cleanup efforts and bug fixes, dropping support for Python <3.6, supporting Terraform 0.13.x
0.12.x	0.8.x	Terrascript 0.8 are a (almost) complete rewrite
0.12.x	0.7.x	Never released
0.11.x	0.6.x	Last releases to support Terraform 0.11 and earlier

Terrascript supports Python 3.6 and later.

1.3.2 Module layout

Python-Terrascript release 0.8.0 changed the location of modules. Providers, resources and data sources are now all available through just three modules.

```
import terrascript
import terrascript.provider     # aws, google, ...
import terrascript.resource     # aws_instance, google_compute_instance, ...
import terrascript.data         # aws_ami, google_compute_image, ...
```

The legacy layout is still available but should not be used for new projects.

```
import terrascript
import terrascript.aws          # aws
import terrascript.aws.r        # aws_instance, ...
import terrascript.aws.d        # aws_ami, ...
```

1.4 A first example

The following example has been taken from the official Terraform documentation for the AWS Provider and then converted into a Python script that generates the equivalent configuration in JSON syntax.

The original Terraform HCL format.

```
provider "aws" {
  version = "~> 2.0"
  region  = "us-east-1"
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

The Terrascript code would look like this.

```
import terrascript
import terrascript.provider as provider
import terrascript.resource as resource

config = terrascript.Terrascript()

config += provider.aws(version='~> 2.0', region='us-east-1')
config += resource.aws_vpc('example', cidr_block='10.0.0.0/16')

with open('config.tf.json', 'wt') as fp:
    fp.write(str(config))
```

The content of `config.tf.json` is shown below. It is equivalent to the original HCL format.

```
{
  "provider": {
    "aws": [
      {
        "version": "~> 2.0",
        "region": "us-east-1"
      }
    ]
  },
  "resource": {
    "aws_vpc": {
      "example": {
        "cidr_block": "10.0.0.0/16"
      }
    }
  }
}
```

Terrascript does not verify that the generated JSON code is a valid Terraform configuration. This is a deliberate design decision and is explained in the Frequently Asked Questions (FAQ)

1.5 Links

- Documentation for Python-Terrascript.
- Github page of Python-Terrascript.
- Community Chat on Zulip.
- Terraform JSON syntax.

CHAPTER 2

Installing Terrascript

Terarscript is available from the Python Package Repository PyPi or alternatively from its Github repository.

2.1 Installing Terrascript from PyPi

It is easiest to install Terrascript directly from the Python Package Index.

```
$ pip install terrascript
```

2.2 Installing Terrascript from Github

Terrascript can also be installed from its Github repository.

```
$ git clone https://github.com/mjuenema/python-terrascript.git
$ cd python-terrascript/
$ git fetch
$ git fetch --tags
```

The `master` branch should be identical to the version on PyPi.

```
$ git checkout master
$ python3 setup.py install
```

The `develop` branch includes the latest changes but may not always in a stable state. Do not use the `develop` branch unless you want to submit a merge request on github.

```
$ git checkout develop
$ python3 setup.py install
```

CHAPTER 3

Quickstart

The following samples shows Terraform's native HCL format and the Terrascript equivalent.

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

resource "aws_instance" "example" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
}
```

In Python, the first step is to import the required modules. In this example this covers the main Terrascript module a the Amazon Web Services (AWS) specific modules for the provider and resources like `aws_instance`.

```
import terrascript
import terrascript.provider
import terrascript.resource
```

The `terrascript.Terrascript` class is the top-level 'container' for configurations. Provider, resource, data sources and other blocks are then added (`` `+=` `` or `` `terrascript.Terrascript.add(...)` ``) later.

```
config = terrascript.Terrascript()
config += terrascript.provider.aws(profile='default', region="us-east-1")
config += terrascript.resource.aws_instance('example', ami='ami-2757f631',
                                            instance_type='t2.micro')
```

The content of `config` is actually just a Python dictionary with some additional smarts.

```
assert isinstance(config, dict) is True
```

Finally `config` has to be converted into JSON which can be achieved in three different ways.

Option 1: Simply use the `str()` representation to print or convert to JSON.

```
print(config)
# or
cfg = str(config)
```

Option 2: Use the `json` module from the Python Standard Library but ensure that `sort_keys` is set to `False`. `indent` can be set as preferred.

```
import json

cfg = json.dumps(config, indent=2, sort_keys=False).
```

Option 3: Use the `.dump()` method. This option has been retained for backward compatibility and may be removed in the future.

```
cfg = config.dump()
```

Whichever method you chose, the output will be the following JSON code.

```
{
  "provider": {
    "aws": [
      {
        "profile": "default",
        "region": "us-east-1"
      }
    ]
  },
  "resource": {
    "aws_instance": {
      "example": {
         "ami": "ami-2757f631",
         "instance_type": "t2.micro"
      }
    }
  }
}
```

The generated JSON file is valid input for Terraform.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (terraform-providers/aws) 2.25.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.25"
```

(continues on next page)

(continued from previous page)

```
Terraform has been successfully initialized!

$ terraform validate
Success! The configuration is valid.
```

CHAPTER 4

Tutorial

This section explains the different Terrascript Python classes that can be used to generate a Terraform configuration.

Since release 0.8.0 all Terrascript classes are available by importing just four modules.

```
import terrascript
import terrascript.provider
import terrascript.resource
import terrascript.data
```

Note: The old layout, e.g. `import terarscript.aws.r` is still available for backward compatibility but its use is discouraged.

4.1 Provider

Providers can be found in the `terrascript.provider` module, with one class for each provider. Terrascript supports most Terraform providers. The full list can be found in List of Providers.

HCL

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

provider "aws" {
  alias   = "west"
  region  = "us-west-1"
}
```

Python

```
import terrascript
import terrascript.provider

config = terrascript.Terrascript()

# Amazon Web Service with aliases
config += terrascript.provider.aws(alias="east", region="us-east-1")
config += terrascript.provider.aws(alias="west", region="us-west-1")
```

JSON

```
{
  "provider": {
    "aws": [
      {
        "alias": "east",
        "region": "us-east-1"
      },
      {
        "alias": "west",
        "region": "us-west-1"
      }
    ]
  }
}
```

4.2 Resource

Resources can be found in the `terrascript.resource` module. The example below shows the original HCL syntax for creating an AWS S3 bucket and the equivalent Python code.

HCL

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

resource "aws_s3_bucket" "mybucket" {
    bucket = "mybucket"
    acl    = "private"
    tags = {
        Name        = "My bucket"
        Environment = "Dev"
    }
}
```

Python

And here is the same as a Python script. The first argument to `terrascript.resource.aws_s3_bucket()` is the Terraform label under which it can be referenced later. Note how the `tags` is a dictionary as in the HCL syntax.

```
import terrascript
import terrascript.provider
import terrascript.resource

```

(continues on next page)

(continued from previous page)

```
config = terrascript.Terrascript()

# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# Add an AWS S3 bucket resource
config += terrascript.resource.aws_s3_bucket(
    "mybucket",
    bucket="mybucket",
    acl="private",
    tags={"Name": "My bucket", "Environment": "Dev"},
)
```

JSON

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "resource": {
    "aws_s3_bucket": {
      "mybucket": {
        "bucket": "mybucket",
        "acl": "private",
        "tags": {
          "Name": "My bucket",
          "Environment": "Dev"
        }
      }
    }
  }
}
```

4.3 Data Source

Data Sources are located in the `terrascript.data` module. The example creates a Google Compute Instance based on the Debian-9 image. First the Terrascript HCL syntax.

```
provider "google" {
    credentials = "${file("account.json")}"
    project     = "myproject"
    region      = "us-central1"
}

data "google_compute_image" "debian9" {
    family  = "debian-9"
    project = "debian-cloud"
}

resource "google_compute_instance" "myinstance" {
```

(continues on next page)

(continued from previous page)

```
    name         = "test"
    machine_type = "n1-standard-1"
    zone         = "us-central1-a"
    boot_disk {
        initialize_params {
            image = data.google_compute_image.debian9.self_link
        }
    }
    network_interface {
        network = "default"
        access_config {
            // Ephemeral IP
        }
    }
}
```

And the same as Python code.

```
import terrascript
import terrascript.provider
import terrascript.resource
import terrascript.data

config = terrascript.Terrascript()

# Google Cloud Compute provider
config += terrascript.provider.google(
    credentials='${file("account.json")}', project="myproject", region="us-central1"
)

# Google Compute Image (Debian 9) data source
config += terrascript.data.google_compute_image("image", family="debian-9")

# Add Google Compute Instance resource
config += terrascript.resource.google_compute_instance(
    "myinstance",
    name="myinstance",
    machine_type="n1-standard-1",
    zone="us-central1-a",
    boot_disk={
        "initialize_params": {"image": "data.google_compute_image.image.self_link"}
    },
    network_interface={"network": "default", "access_config": {}},
)
```

The example above is mostly a one-to-one adaptation of the HCL syntax. Let's make some changes to show how generating Terraform configurations through Python-Terrascript may help.

- Define the Google Compute Image family and Google Compute Instance machine type at the beginning of the script so they are easier to change.
- Reference an instance of the Python-Terrascript class `terrascript.data.google_compute_image()` as the boot disk image.

```
IMAGE_FAMILY = "debian-9"
MACHINE_TYPE = "n1-standard-1"
```

(continues on next page)

(continued from previous page)

```
import terrascript
import terrascript.provider
import terrascript.resource
import terrascript.data

config = terrascript.Terrascript()

# Google Cloud Compute provider
config += terrascript.provider.google(
    credentials='${file("account.json")}', project="myproject", region="us-central1"
)

# Google Compute Image (Debian 9) data source
image = terrascript.data.google_compute_image("image", family=IMAGE_FAMILY)
config += image

# Add Google Compute Instance resource
config += terrascript.resource.google_compute_instance(
    "myinstance",
    name="myinstance",
    machine_type=MACHINE_TYPE,
    zone="us-central1-a",
    boot_disk={"initialize_params": {"image": image.self_link}},
    network_interface={"network": "default", "access_config": {}},
)
```

4.4 Variable

The `terrascript.Variable` class can be used to define variables that can be referenced later. Python-Terrascript automatically takes care of converting a reference to a Python variable into the correct Terraform JSON syntax.

HCL

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

variable "image_id" {
  type = string
}

resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = var.image_id
}
```

Python

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()
```

(continues on next page)

(continued from previous page)

```
# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# Define Variable and add to config
v = terrascript.Variable("image_id", type="string")
config += v

# AWS EC2 instance referencing the variable.
config += terrascript.resource.aws_instance(
    "example",
    instance_type="t2.micro",
    ami=v,
)
```

JSON

In the output the reference to the `image_id` has been converted from a reference to a Python variable `ami=v` to the correct Terraform JSON syntax of `${var.image_id}`.

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "variable": {
    "image_id": {
      "type": "string"
    }
  },
  "resource": {
    "aws_instance": {
      "example": {
        "instance_type": "t2.micro",
        "ami": "${var.image_id}"
      }
    }
  }
}
```

4.5 Output

Output is implemented as the `terrascript.Output` class.

HCL

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

variable "image_id" {
  type = string
}
```

(continues on next page)

(continued from previous page)

```
resource "aws_instance" "example" {
  instance_type = "t2.micro"
  ami           = var.image_id
}

output "instance_ip_addr" {
  value         = aws_instance.server.private_ip
  description   = "The private IP address of the instance."
}
```

Python

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()

# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# Define Variable and add to config
v = terrascript.Variable("image_id", type="string")
config += v

# Define AWS EC2 instance and add to config
i = terrascript.resource.aws_instance("example", instance_type="t2.micro", ami=v)
config += i

# Output the instance's private IP
config += terrascript.Output(
    "instance_ip_addr",
    value=i.private_ip,
    description="The private IP address of the instance.",
)
```

JSON

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "variable": {
    "image_id": {
      "type": "string"
    }
  },
  "resource": {
    "aws_instance": {
      "example": {
        "instance_type": "t2.micro",
        "ami": "${var.image_id}"
```

(continues on next page)

(continued from previous page)

```
      }
    }
  },
  "output": {
    "instance_ip_addr": {
      "value": "aws_instance.example.private_ip",
      "description": "The private IP address of the instance."
    }
  }
}
```

4.6 Module

Calls to other Terraform modules are implemented through the `terrascript.Module` class.

HCL

```
module "ec2_cluster" {
  source                 = "terraform-aws-modules/ec2-instance/aws"
  version                = "~> 2.0"

  name                   = "my-cluster"
  instance_count         = 5

  ami                    = "ami-ebd02392"
  instance_type          = "t2.micro"
  key_name               = "user1"
  monitoring             = true
  vpc_security_group_ids = ["sg-12345678"]
  subnet_id              = "subnet-eddcdzz4"
}
```

Python

```
"""Terrascript module example based on https://registry.terraform.io/modules/
↪terraform-aws-modules/ec2-instance/aws/"""

import terrascript
import terrascript.provider

config = terrascript.Terrascript()

# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# AWS EC2 module
config += terrascript.Module(
    "ec2_cluster",
    source="terraform-aws-modules/ec2-instance/aws",
    version="~> 2.0",
    name="my-cluster",
    instance_count=5,
    ami="ami-ebd02392",
    instance_type="t2.micro",
```

(continues on next page)

(continued from previous page)

```
    key_name="user1",
    monitoring=True,
    vpc_security_group_ids=["sg-12345678"],
    subnet_id="subnet-eddcdzz4",
)
```

JSON

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "module": {
    "ec2_cluster": {
      "source": "terraform-aws-modules/ec2-instance/aws",
      "version": "~> 2.0",
      "name": "my-cluster",
      "instance_count": 5,
      "ami": "ami-ebd02392",
      "instance_type": "t2.micro",
      "key_name": "user1",
      "monitoring": true,
      "vpc_security_group_ids": [
        "sg-12345678"
      ],
      "subnet_id": "subnet-eddcdzz4"
    }
  }
}
```

4.7 Backend

HCL

```
terraform {
  backend "consul" {
    address = "demo.consul.io"
    scheme  = "https"
    path    = "example_app/terraform_state"
  }
}
```

Python

```
import terrascript

config = terrascript.Terrascript()

backend = terrascript.Backend(
    "consul",
```

(continues on next page)

(continued from previous page)

```
    address="demo.consul.io",
    scheme="https",
    path="example_app/terraform_state",
)

config += terrascript.Terraform(backend=backend)
```

JSON

```
{
  "terraform": {
    "backend": {
      "consul": {
        "address": "demo.consul.io",
        "scheme": "https",
        "path": "example_app/terraform_state"
      }
    }
  }
}
```

4.8 Connection

4.9 Locals

Terraform local values are not supported. Use Python variables instead.

Python

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()

# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# Local values as Python variables
tags = {"service_name": "forum", "owner": "Community Team"}

# Resource with two provisioners
config += terrascript.resource.aws_instance(
    "instance1",
    instance_type="t2.micro",
    ami="ami-4bf3d731",
    tags=tags,
)
```

JSON

```
{
  "provider": {
```

(continues on next page)

(continued from previous page)

```
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "resource": {
    "aws_instance": {
      "instance1": {
        "instance_type": "t2.micro",
        "ami": "ami-4bf3d731",
        "tags": {
          "service_name": "forum",
          "owner": "Community Team"
        }
      }
    }
  }
}
```

4.10 Provisioner

One or multiple provisioners can be added to a Terraform resource. Multiple provisioners must be added as a Python list, a single provisioner can be either on its own or inside a list.

The example adds one “create” and one “destroy” provisioner.

Python

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()

# AWS provider
config += terrascript.provider.aws(region="us-east-1")

# Provisioners
create = terrascript.Provisioner("local-exec", command="echo 'Create'")
destroy = terrascript.Provisioner(
    "local-exec",
    when="destroy",
    command="echo 'Destroy'",
)

# Resource with two provisioners
config += terrascript.resource.aws_instance(
    "instance1",
    instance_type="t2.micro",
    ami="ami-4bf3d731",
    provisioner=[
        create,
        destroy,
```

(continues on next page)

(continued from previous page)

```
    ],
)
```

JSON

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1"
      }
    ]
  },
  "resource": {
    "aws_instance": {
      "instance1": {
        "instance_type": "t2.micro",
        "ami": "ami-4bf3d731",
        "provisioner": [
          {
            "local-exec": {
              "command": "echo 'Create'"
            }
          },
          {
            "local-exec": {
              "when": "destroy",
              "command": "echo 'Destroy'"
            }
          }
        ]
      }
    }
  }
}
```

CHAPTER 5

Examples

More examples can be found in in Terrascripts examples/ directory.

5.1 AWS

5.1.1 VPCs

This example has been copied from the Terraform documentation for the AWS Provider.

Terraform HCL code:

Python code:

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()
config += terrascript.provider.aws(region="us-east-1", version="~> 2.0")
config += terrascript.resource.aws_vpc("example", cidr_block="10.0.0.0/16")

print(config)
```

JSON output:

```
{
  "provider": {
    "aws": [
      {
        "version": "~> 2.0",
        "region": "us-east-1"
      }
    ]
```

(continues on next page)

(continued from previous page)

```
  },
  "resource": {
    "aws_vpc": {
      "example": {
        "cidr_block": "10.0.0.0/16"
      }
    }
  }
}
```

5.1.2 Provider Endpoints

Terraform HCL code:

Python code:

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()

config += terrascript.provider.aws(
    region="us-east-1",
    version="~> 2.0",
    endpoints=terrascript.Block(
        dynamodb="http://localhost:4569",
        s3="http://localhost:4572",
    ),
)

config += terrascript.resource.aws_vpc("example", cidr_block="10.0.0.0/16")

print(config)
```

JSON output:

```
{
  "provider": {
    "aws": [
      {
        "version": "~> 2.0",
        "region": "us-east-1",
        "endpoints": {
          "dynamodb": "http://localhost:4569",
          "s3": "http://localhost:4572"
        }
      }
    ]
  },
  "resource": {
    "aws_vpc": {
      "example": {
        "cidr_block": "10.0.0.0/16"
      }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

CHAPTER 6

List of Providers

- alicloud
- archive
- arukas
- aws
- azure
- azurerm
- bitbucket
- chef
- circonus
- clc
- cloudflare
- cloudscale
- cloudstack
- cobbler
- datadog
- digitalocean
- dme
- dns
- dnsimple
- docker
- dyn
- external

- fastly
- github
- gitlab
- google
- grafana
- heroku
- http
- icinga2
- ignition
- influxdb
- kubernetes
- librato
- local
- logentries
- logicmonitor
- mailgun
- matchbox
- mysql
- newrelic
- nomad
- ns1
- oci
- oneandone
- opc
- openstack
- opsgenie
- ovh
- packet
- pagerduty
- postgresql
- powerdns
- profitbricks
- rabbitmq
- rancher
- random
- rundeck

- scaleway
- shell
- softlayer
- spotinst
- statuscake
- template
- tls
- triton
- ultradns
- vault
- vcd
- vsphere

CHAPTER 7

Terraform JSON format

The following sections show the JSON output Python-Terrascript generates for the different Terraform elements.

7.1 Resource

Resources are coded as nested dictionaries. The top-level key is the type of the resource. The second-level key is the name of a resource instance.

Example:

- Two `aws_instance` resources named `instance1` and `instance2`
- One `aws_subnet` resource named `subnet1`.

```
{
  "resource": {
    "aws_instance": {
      "instance1": {
        "instance_type": "t2.micro"
      },
      "instance2": {
        "instance_type": "t2.micro"
      }
    },
    "aws_subnet": {
      "subnet1" {
        "availability_zone": "usa-west-2a"
      }
    }
  }
}
```

7.2 Provider

A Terraform configuration contains one or more provider sections. Unlike resources and data sources providers don't have a name and are distinguised by their `alias` attribute instead. Therefore multiple instances of the same provider type are encoded as a list.

Example:

- Two `aws` providers with aliases `aws1` and `aws2`.
- One `google` provider with alias `google1`.

```
{
  "provider": {
    "aws": [
      {
        "region": "us-east-1",
        "alias": "aws1"
      },
      {
        "region": "us-east-2",
        "alias": "aws2"
      }
    ],
    "google": [
      {
        "region": "us-central-1",
        "alias": "google1"
      }
    ]
  }
}
```

7.3 Variable

Variables are encoded as a dictionary whose keys are the names of the variables.

Example:

- Two variables names `image_id` and `availability_zone_names`.

```
{
  "variable": {
    "image_id": {
      "type": "string"
    },
    "availability_zone_names": {
      "type": "list(string)",
      "default": [
        "us-west-1a",
        "us-west-1b"
      ]
    }
  }
}
```

7.4 Variable references

When using a variable as a attribute value for a object, a reference are used.

Example: * Variable `image_id` are used as name for instance.

```
{
  "variable": {
    "image_id": {
      "type": "string"
    }
  },
  "resource": {
    "aws_instance": {
      "instance1": {
        "instance_type": "${var.image_id}"
      }
    }
  }
}
```

7.5 Output Values

Output values are encoded as a dictionary whose keys are the names of the value.

Example:

- Output values for two resources.

```
{
  "output": {
    "instance1_ip_addr": {
      "value": "instance1.server.private_ip"
    },
    "instance2_ip_addr": {
      "value": "instance2.server.private_ip"
    },
  }
}
```

7.6 Local Values

Local values are encoded as a dictionary whose keys are the names of the value.

Example:

```
{
  "locals": {
    "service_name": "forum",
    "owner": "Community Team",
    "Service": "local.service_name",
    "Owner": "local.owner"
  }
}
```

7.7 Modules

Module calls are dictionaries keyed by the name of the module and module arguments as values.

Note: In contrast to calling existing modules, creating modules is not supported by Python-Terrascript as Python functions could be used as an alternative.

Example:

- Calling module `vpc`.

```
{
  "module": {
    "vpc": {
      "source": "terraform-aws-modules/vpc/aws",
      "version": "2.9.0"
    }
  }
}
```

7.8 Data Sources

Data sources are coded as nested dictionaries. The top-level key is the type of the resource. The second-level key is the name of the data source.

Example:

- Two `aws_ami` data sources named `ami1` and `ami2`.

```
{
  "data": {
    "aws_ami": {
      "ami1": {
        "most_recent": true
      },
      "ami2": {
        "most_recent": true
      },
    }
  }
}
```

7.9 Expressions

7.10 Functions

Functions are encoded as text. Example: `"content": "file('hello_world.txt')"`.

7.11 Terraform Settings

Terraform settings are a simple dictionary although the values of settings may contain nested data structures.

Example:

- Terraform backend configuration.

```
{
  "terraform": {
    "backend": {
      "s3": {
        "bucket": "mybucket"
      }
    }
  }
}
```

CHAPTER 8

Frequently Asked Questions

Questions I frequently asked myself ;-)

8.1 Why no error checking?

Python-Terrascript does not perform any error checking whatsoever! This was a deliberate design decision to keep the code simple. Therefore it is perfectly possible to generate JSON output that Terraform will later reject.

```
import terrascript
import terrascript.provider
import terrascript.resource

config = terrascript.Terrascript()

config += terrascript.provider.aws(region='us-east-1')

i = terrascript.resource.aws_instance('myinstance', foo='bar')
config += i

# AWS Instance resource does not have a `foo` argument but accepts it anyway.
assert i.foo == 'bar'
```

```
 {
 "provider": {
     "aws": [
     {
         "region": "us-east-1"
     }
     ]
 },
 "resource": {
     "aws_instance": {
       "myinstance": {
```

(continues on next page)

(continued from previous page)

```
            "foo": "bar"
        }
    }
}
```

Terraform will reject the generated JSON as the *aws_instance* resource does not accept the *foo* argument.

At an early stage I contemplated parsing the Terraform Go source code and auto-create Python code that does indeed verify whether the generated JSON configuration is valid Terraform input. This attempt proved much too difficult so I abandonded that approach.

8.2 Why is provider XYZ not supported?

All provider specific code is auto-generated through the `tools/makecode.py` script based on the list of providers in `tools/providers.yml`. So if a provider is missing it simply has not yet been added to `tools/providers.yml`.

If you believe a provider is missing simply open a new issue or submit a pull request with the missing provider added to `tools/providers.yml`.

8.3 Why is there sometimes so little progress in this project?

Python-Terrascript is a hobby project which I can only work on in my very limited spare time.

Professionally I am a Systems and Network Engineer working on an Intelligent Transport Systems project. None of its infrastructure is in the Cloud, it's all very physcially located on the freeways around Melbourne, Australia. Until someone writes Terraform modules for traffic cameras or vehicle sensors my manager simply wouldn't appreciate if I spent time on **Python-Terrascript** during work hours.

8.4 Is Python-Terrascript better than writing configurations by hand?

I regard **Python-Terrascript** as an *alternative* to writing Terraform configurations by hand. Whether it is better or not is for everyone to decide for themselves.

8.5 How can I contribute to the project?

There a various ways you can contribute to the development of **Python-Terrascript**.

- Issues: Do not hestiate to raise an issue if you believe you found a bug or simply have a question. As the maintainer of a small Github project it is amazingly satisfying to see that there are other people out there who find **Python-Terrascript** useful. Do not get discouraged if I don't respond immediately. As mentioned earlier I can only spend limited time on **Python-Terrascript**. You will hear from me eventually.
- Pull Requests: Yes, *Pull Requests* are welcome but please bear in my that this is a *personal* project and not a community project. I promise to provide an explanation if should I reject a Pull Request.
- Documentation: Documentation can always be improved so any support (Issues, Pull Requests) is very welcome. The biggest problem with documentation is always what may be obvious to me may not be obvious to the reader at all. I also tend to be rather terse when writing documentation which is not a good thing.

- Examples: I would like to include a collection of real-world examples of how **Python-Terrascript** is used. So if you are willing to share your code please come forward.
- Drinks: Anyone willing to catch-up for a chat over coffee (hot chocolate in my case) or beer when they are in Melbourne?

8.6 Are there any alternatives to Python-Terrascript?

I know that there are comparable projects to **Python-Terrascript**. I just haven't managed to compile a list yet. Please stand-by for updates...

CHAPTER 9

Indices and tables

- genindex
- modindex
- search