# Fakeme

***Release 0.0.1***

**May 28, 2020**

# Contents

in TODO

Please check Examples section *More Examples*

You can use Fakeme with 2 ways. More flexible is to use from python script with import

```
from fakeme import Fakeme
```

Fakeme - main class that you need to call to run data generation.

Minimal that you need to provide to Generator is a **tables_list** argument.

**Fakeme(** tables=[()]

)

*tables* param You must provide a list of tuples: [()]. Each tuple define one table. You must provide table_id and schema, also (optional) you can provide dataset_id.

If you provide only table_id and schema order does not matter: library check type of elements in tuple. But if you provide tuple with 3 args: dataset_id, table_id and schema, table_id always must have index after dataset_id this is correct:

('dataset_id', 'table_1', [{'name': 'id'}, {'name': 'value'}])

**this is wrong ('dataset_id will be used as table name):** ('table_1', 'dataset_id', [{'name': 'id'}, {'name': 'value'}])

CHAPTER 1

# Tutorial

# CHAPTER 2

## More Examples

```
from fakeme import Fakeme
from fakeme.fields import FieldRules

# STEP 1: Define schemas
schema_one_parts_list = [
{
  "type": "STRING",
  "name": "part_identification",
  "mode": "NULLABLE"
 },
 {
  "type": "STRING",
  "name": "ship_type",
  "mode": "NULLABLE"
 },
 {
  "type": "STRING",
  "name": "price",
  "mode": "NULLABLE"
 }
]


# STEP 2: Add rules for field generation if you want, if not - will be used default␣
↪generation rules
FieldRules.user_rules.append(
    {"field": "count", "generator": "str(randint(100, 6000))", "len": ""})

# "generator" must contains code, that can be executed in Generator module with "eval
↪" command
# to see that methods are exist in Generator that you can use - you can simple just␣
↪check fakeme.generator module

# define more rule
```

(continues on next page)

(continued from previous page)

```
ship_type = {"field": "ship_type", "generator": "'Ship ' + text.word()",  "len": ""}
FieldRules.user_rules.append(ship_type)

# create list of tables, each tuple - one table, values in indexes:
# 1st - dataset/database name
# 2nd - table name
# 3rd - table's schema

list_of_tables = [
    ('robot_factory', 'parts', schema_one_parts_list),
    ('robot_factory', 'warehouse', 'warehouse_schema.json')  # second schema we will␣
↪read from the file
]

# STEP 3: define dependencies and generation rules
Fakeme(tables=list_of_tables,
             dump_schema=True,
             params={'row_numbers': 15},  # how much rows we want to generate
             # rls stands for  relationship - defining relationship between tables,
             # that field depend on that
             rls={'warehouse': {'part_id': {'alias': 'part_identification',
                                            'matches': 1,
                                            'table': 'parts'}}
                  }).run()

# now just run `python space_ship_warehouse_tables.py`

# as result you will see 2 json files, that contains same data
# in part_identification (in parts.json) and part_id (in warehouse.json) fields
```

CHAPTER 3

# Supported Schemas for Tables

Schema is needed for Fakeme to know that columns and of that data types must be generated.

As default, library use Schema style from BigQuery - https://cloud.google.com/bigquery/docs/schemas#creating_a_json_schema_file but without 'description' field.

Usually schema looks like a list of dicts - one dict per column and inside each dict: - 'name': column name, - 'type': data type of values in column, - 'mode': contains information nullabe / required. This field is not sensitive to capitalization.

Example:

```
[{
  "type": "STRING",
  "name": "part_identification",
  "mode": "Required"
 },
 {
  "type": "FLOAT",
  "name": "ship_type",
  "mode": "NULLABLE"
 },
 {
  "type": "INTEGER",
  "name": "price",
  "mode": "NULLABLE"
 }]
```

If you don't provide a type, like this:

And Fakeme does not have base rule for generating columns with such name - output will be random string.

But if we have generator for field. For example, with name "price":

```
{
  "name": "price",
```

(continues on next page)

(continued from previous page)

```
  "mode": "NULLABLE"
}
```

Result will be of float type based on existed Rule for Generating (check rules in fakeme/rules.py).

Schemas from DDL

Supported Databases

- MSSql
- MySql
- PostgreSql
- Oracle DB
- Hive (HQL)
- SQLite

How to fix errors in process of DDL parsing?

How to add new DDL-notation/custom implementation/database ?

CHAPTER 4

# Output Formats

CHAPTER 5

# User Customisation

CHAPTER 6

# How to add custom fields generators

Find example in:

fakeme/examples/space_ship_parts/space_ship_warehouse_tables.py

If you want to add your new field rule (how to generate it correct), you can do it from your python script runner:

at the bottom of your script (before you call RunGenerator) add:

from fakeme.fields import FieldRules

FieldRules.user_rules.append( {“field”: “count”, “generator”: “str(randint(100, 6000))”, “len”: “”})

CHAPTER 7

# Values Generators

CHAPTER 8

# Run generator from Command Line

Example in:

fakeme/examples/cli_usage

Define your generation config in any json file.

It can contain only settings and params that allowed and used in

In example for you already created fakeme_config.json

to run generator just run fakeme and provide correct relative or absolute path to config file

fakeme fakeme_config.json

CHAPTER 9

# Rules for value generating

## 9.1 What is the Rule?

Rule is a description, how **Fakeme** need to generate Value in the Field.

Rule mapped to field by field name and by default it try auto to decide that