dammy

Release 1.2.0

Dec 21, 2020

Contents

Welcome to the official dammy documentation. Here you will find tutorials, documentation of the code and ways to contribute to the project.

CHAPTER 1

Tutorial

Welcome to the tutorial section! Here you will learn to use dammy to its fullest following an example that will grow each step, starting at the simplest and ending using all the functionalities available.

In this example wie will build a dataset containing persons and cars, each person being the owner of at least one car.

1.1 What is a generator?

In dammy, a generator is a class which generates random data. Plain and simple. All generators must inherit *dammy.BaseGenerator*. Generators can be simple or

composite. A simple generator generates a value, while a composite generator is composed of multiple simple generators and generates a value for each of them.

For example, a integer generator is a simple generator, because it only generates integers. A person generator is composite because generates a name, an age, a height...

In our example, we will need 2 composite generators, one for the person and one for the car:

```
class Person():
    # The description of a person goes here
    pass

class Car():
    # The description of a person goes here
    pass
```

Composite generators must inherit from *dammy.EntityGenerator*. So the final code looks like this:

```
from dammy import EntityGenerator

class Person(EntityGenerator):
    # The description of a person goes here
    pass
```

(continues on next page)

(continued from previous page)

```
class Car(EntityGenerator):
    # The description of a car goes here
    pass
```

Now you could instantiate these classes and call their generate() method, although it will return an empty dictionary because they contain no attributes. We will add some attributes right now.

1.2 Built-in generators

A person must have some attributes such as name, age, country of origin... and a car should have a brand, a model and a plate number.

But how to generate all of this? Don't worry about it, we have taken care of it and done the hard work for you. To generate all of this, just import stdlib (*Built-in generators*) and add the generators as follows:

```
from dammy import EntityGenerator
from dammy.stdlib import RandomName, CountryName, RandomInteger, CarBrand, CarModel

class Person(EntityGenerator):
    name = RandomName()          # Generate a random name (any gender)
    age = RandomInteger(18, 99)  # Generate a randim integer between 18 and 99
    country = CountryName()      # Generate a country name

class Car(EntityGenerator):
    brand = CarBrand()                    # Generate a car brand
    model = CarModel(car_brand=brand)     # Generate a car model matching that brand
    plate = RandomInteger(1000, 9999)     # Generate a plate number
```

You can check all the available generators at *Built-in generators*

Note: Note that the car brand can be passed as a parameter on CarModel to generate car models corresponding to the generated manufacturer.

With this, you can already generate individual cars and people! Just run:

```
print(Car())
print(Person())
```

And you will get the following output:

```
>>> print(Car())
{'brand': 'Kia', 'model': 'Cadenza', 'plate': 9138}
>>> print(Person())
{'name': 'Meir', 'age': 35, 'country': 'Switzerland'}
```

Note: Keep in mind that you don't have to create an instance every time you want to generate a new entity. The example above prints a new entity because the __str__() method calls the .generate() method.

In other words, you can also generate a car and a person like this:

```
c = Car()
c.generate()
```

1.3 Playing with generators

Now, lets suppose we want a person to have a field called birthdate, which obviously contains the persons birth date. We also want the car model name to be uppercase. How do we make the birthdate date match the age? And how can we alter the generated values if we cant access them until they are generated?

It is quite easy if you already know how to do all of this in python. You want to convert a string to uppercase, just call the .upper() method on the string. Want to get someones age? Get the current date and the birthdate and substract them.

With dammy it's just the same. If you are generating a string, you can call any methods, access any attributes and use all the operators of the string class. This principle extends to every dammy entity, no matter the type of the generated value.

The updated example looks like this:

```
from datetime import datetime

from dammy import EntityGenerator
from dammy.functions import cast
from dammy.stdlib import RandomName, CountryName, RandomInteger, CarBrand, CarModel,␣
↪RandomDateTime

class Person(EntityGenerator):
    name = RandomName()          # Generate a random name (any gender)
    birthdate = RandomDateTime(start=datetime(1980, 1, 1), end=datetime(2000, 12, 31),
↪ date_format='%d/%m/%Y')  # Generate a random datetime
    age = cast((datetime.now() - birthdate).days / 365.25, int) # Get the difference␣
↪in days, divide it by 365.25 to get it in years and cast it to an integer
    country = CountryName()      # Generate a country name

class Car(EntityGenerator):
    brand = CarBrand()                          # Generate a car brand
    model = CarModel(car_brand=brand).upper()   # Generate a car model matching that␣
↪brand and convert it to uppercase
    plate = RandomInteger(1000, 9999)           # Generate a plate number
```

Note that some new imports are required

Now if you generate a car and a person as we did before you will get the following:

```
>>> print(Car())
{'brand': 'Opel', 'model': 'MERIVA', 'plate': 8130}
>>> print(Person())
{'name': 'Brianny', 'birthdate': '16/04/1991', 'age': 28, 'country': 'Guyana'}
```

1.4 Generating a a dataset

To generate a dataset, persons and cars must be linked in some way. You could just do this:

```
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
from dammy import EntityGenerator
from dammy.functions import cast
from dammy.stdlib import RandomName, CountryName, RandomInteger, CarBrand, CarModel,␣
↪RandomDateTime

class Person(EntityGenerator):
    name = RandomName()          # Generate a random name (any gender)
    birthdate = RandomDateTime(start=datetime(1980, 1, 1), end=datetime(2000, 12, 31),
↪ date_format='%d/%m/%Y')  # Generate a random datetime
    age = cast((datetime.now() - birthdate).days / 365.25, int) # Get the difference␣
↪in days, divide it by 365.25 to get it in years and cast it to an integer
    country = CountryName()      # Generate a country name

class Car(EntityGenerator):
    brand = CarBrand()                              # Generate a car brand
    model = CarModel(car_brand=brand).upper()    # Generate a car model matching that␣
↪brand and convert it to uppercase
    plate = RandomInteger(1000, 9999)               # Generate a plate number
    owner = Person()                                # Generate a person
```

And just generating a new car would generate a person associated to that car:

```
>>> print(Car())
{'brand': 'Ford', 'model': 'KA', 'plate': 7970, 'owner': {'name': 'Ayat', 'birthdate
↪': '27/12/1981', 'age': 38, 'country': 'Bermuda'}}
```

But this way one to one relationships can only be established, and does not work very well when working with relational databases.

Primary and foreign keys can be used to achive this, as you would do with a regular database:

```
from datetime import datetime

from dammy import EntityGenerator
from dammy.db import PrimaryKey, ForeignKey, AutoIncrement
from dammy.functions import cast
from dammy.stdlib import RandomName, CountryName, RandomInteger, CarBrand, CarModel,␣
↪RandomDateTime

class Person(EntityGenerator):
    identifier = PrimaryKey(AutoIncrement())    # Add an autoincrement and make it␣
↪primary key
    name = RandomName()          # Generate a random name (any gender)
    birthdate = RandomDateTime(start=datetime(1980, 1, 1), end=datetime(2000, 12, 31),
↪ date_format='%d/%m/%Y')  # Generate a random datetime
    age = cast((datetime.now() - birthdate).days / 365.25, int) # Get the difference␣
↪in days, divide it by 365.25 to get it in years and cast it to an integer
    country = CountryName()      # Generate a country name

class Car(EntityGenerator):
    brand = CarBrand()                              # Generate a car brand
    model = CarModel(car_brand=brand).upper()    # Generate a car model matching that␣
↪brand and convert it to uppercase
    plate = RandomInteger(1000, 9999)               # Generate a plate number
    owner = ForeignKey(Person, 'identifier')    # Reference to an existing person
```

Notice once again that new imports have been added

Warning: Generating a Car now requires a dataset containing persons to be passed when calling the generate() method. If a dataset is not present a `dammy.exception.DatasetRequiredException` will be raised.

In fact, it is not recommended to generate entities this way when they contain references. The safest way is using a *dammy.db.DatasetGenerator*.

To generate a car, now we need a dataset containing persons. The dataset can be a dictionary or a *dammy.db.DatasetGenerator* But now cars contain references to people, so the best way to generate them is generating a dataset containing cars and people. This can be done using *dammy.db.DatasetGenerator*:

```
from datetime import datetime

from dammy import EntityGenerator
from dammy.db import PrimaryKey, ForeignKey, AutoIncrement, DatasetGenerator
from dammy.functions import cast
from dammy.stdlib import RandomName, CountryName, RandomInteger, CarBrand, CarModel,␣
↪RandomDateTime

class Person(EntityGenerator):
    identifier = PrimaryKey(AutoIncrement())     # Add an autoincrement and make it␣
↪primary key
    name = RandomName()          # Generate a random name (any gender)
    birthdate = RandomDateTime(start=datetime(1980, 1, 1), end=datetime(2000, 12, 31),
↪ date_format='%d/%m/%Y')  # Generate a random datetime
    age = cast((datetime.now() - birthdate).days / 365.25, int) # Get the difference␣
↪in days, divide it by 365.25 to get it in years and cast it to an integer
    country = CountryName()      # Generate a country name

class Car(EntityGenerator):
    brand = CarBrand()                           # Generate a car brand
    model = CarModel(car_brand=brand).upper()    # Generate a car model matching that␣
↪brand and convert it to uppercase
    plate = RandomInteger(1000, 9999)            # Generate a plate number
    owner = ForeignKey(Person, 'identifier')     # Reference to an existing person

generator = DatasetGenerator((Car, 15), (Person, 10))
```

This way you will generate a dataset containing 15 cars and 10 people, with each car associated to a person. You can visualize it by printing it:

```
>> print(generator)
{'Car': [{'brand': 'Peugeot', 'model': '3008', 'plate': 8321, 'owner': 7}, {'brand':
↪'Volvo', 'model': 'V60', 'plate': 2509, 'owner': 6}, {'brand': 'Lexus', 'model': 'LX
↪', 'plate': 9135, 'owner': 4}, {'brand': 'Ferrari', 'model': 'DINO', 'plate': 8054,
↪'owner': 7}, {'brand': 'Renault', 'model': 'LAGUNA', 'plate': 8199, 'owner': 1}, {
↪'brand': 'Audi', 'model': 'A8', 'plate': 8439, 'owner': 9}, {'brand': 'Lexus',
↪'model': 'ES', 'plate': 1363, 'owner': 10}, {'brand': 'Ferrari', 'model': 'DINO',
↪'plate': 1670, 'owner': 3}, {'brand': 'Ferrari', 'model': '208', 'plate': 1157,
↪'owner': 1}, {'brand': 'Ford', 'model': 'FIESTA', 'plate': 9069, 'owner': 6}, {
↪'brand': 'Dacia', 'model': 'LOGAN', 'plate': 6268, 'owner': 9}, {'brand': 'Chevrolet
↪', 'model': 'SONIC', 'plate': 8634, 'owner': 10}, {'brand': 'Mazda', 'model': 'MX-5␣
↪MIATA', 'plate': 2442, 'owner': 4}, {'brand': 'Volvo', 'model': 'S90', 'plate':␣
↪4562, 'owner': 7}, {'brand': 'Kia', 'model': 'SOUL', 'plate': 5322, 'owner': 6}],
↪'Person': [{'identifier': 1, 'name': 'Julianna', 'birthdate': '26/05/2000', 'age':␣
↪19, 'country': 'Saint Barthélemy'}, {'identifier': 2, 'name': 'Lizbeth', 'birthdate
↪': '20/09/1992', 'age': 27, 'country': 'Ethiopia'}, {'identifier': 3, 'name':
↪'Kaylie', 'birthdate': '06/05/1990', 'age': 29, 'country': 'Korea, Republic of'}, {
↪'identifier': 4, 'name': 'Simon', 'birthdate': '12/03/2000', 'age': 19, 'country':
↪'Finland'}, {'identifier': 5, 'name': 'Elisheva', 'birthdate': '09/05/1982', 'age':␣
↪37, 'country': 'Chad'}, {'identifier': 6, 'name': 'Bethany', 'birthdate': '17/07/
↪1988', 'age': 31, 'country': 'Chad'}, {'identifier': 7, 'name': 'Eddy', 'birthdate
↪': '24/03/1982', 'age': 37, 'country': 'Nauru'}, {'identifier': 8, 'name': 'Selena',
↪ 'birthdate': '21/08/1982', 'age': 37, 'country': 'Réunion'}, {'identifier': 9,
↪'name': 'Joziah', 'birthdate': '11/01/1988', 'age': 32, 'country': 'Turkey'}, {
```

(continues on next page)

(continued from previous page)

And it can be exported to SQL:

```
>> print(generator.to_sql())
CREATE TABLE IF NOT EXISTS Person (
    identifier INTEGER,
    name VARCHAR(15),
    birthdate DATETIME,
    age DATETIME,
    country VARCHAR(50),
    CONSTRAINT pk_Person PRIMARY KEY (identifier)
);
CREATE TABLE IF NOT EXISTS Car (
        brand VARCHAR(15),
        model VARCHAR(25),
        plate INTEGER,
        owner_identifier INTEGER,
        CONSTRAINT fk_owner (owner_identifier) REFERENCES Person(identifier)
);
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (1, "Catherine",
↪ "09/10/1981", 38, "Antigua and Barbuda");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (2, "Juliette",
↪"07/01/1995", 25, "Malaysia");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (3, "Ahron",
↪"25/09/1985", 34, "Syrian Arab Republic");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (4, "Emanuel",
↪"28/10/1981", 38, "Uganda");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (5, "Leandro",
↪"04/10/1993", 26, "Burkina Faso");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (6, "Amanda",
↪"28/05/1999", 20, "Uzbekistan");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (7, "Ishmael",
↪"19/01/1995", 24, "Samoa");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (8, "Cormac",
↪"07/02/1986", 33, "Guatemala");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (9, "Stephen",
↪"15/04/1988", 31, "Senegal");
INSERT INTO Person (identifier, name, birthdate, age, country) VALUES (10, "Lara",
↪"25/07/1984", 35, "Puerto Rico");
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Volvo", "S90", 9950,␣
↪2);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Ferrari", "208",␣
↪1225, 7);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("BMW", "F15 X5", 3505,
↪ 1);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Fiat", "500L", 8031,␣
↪10);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Fiat", "500L", 2153,␣
↪10);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Audi", "Q2", 4191,␣
↪7);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("BMW", "F10 5 SERIES",
↪ 4197, 9);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Volvo", "S60", 9587,␣
↪8);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Mercedes-Benz", "A-
↪CLASS", 5285, 4);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Toyota", "CAMRY",␣
↪7922, 3);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Kia", "FORTE", 4746,␣
↪3);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Suzuki", "APV", 7193,
↪ 9);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("BMW", "G06 X6", 6532,
↪ 10);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("Tesla", "MODEL X",␣
↪6701, 3);
INSERT INTO Car (brand, model, plate, owner_identifier) VALUES ("SEAT", "TARRACO",␣
↪5301, 6);
```

Note: To be properly defined and fully compliant with the relational model, Car should have a primary key, which could be the plate number

Please see the full documentation for *dammy.db.DatasetGenerator*.

1.5 Extending built-in generators

If the built-in generators are not enugh for you and the one you need is not available, you can roll your own. This is a more advanced topic so you should read the *Documentation* and then head to the *API reference*.

CHAPTER 2

Documentation

You can access the basic classes by just importing dammy:

```
import dammy
```

But if you want to add more functionality, there are more modules available. Check the documentation page for each module to see the available classes and functions.

db	this module contains everything database related, allowing you to create primary keys, foreign keys and autoincrement fields
exceptions	This module contains all the custom exceptions used by dammy.
functions	This module includes some standard functions to be used on dammy objects
stdlib	This module contains some basic and common utilities, such as random generation for person names, countries, integers, strings, dates...

2.1 The main module

The classes in the main module are listed below.

class dammy.**BaseGenerator**(*sql_equivalent*)

DAMMY_LOCALIZATION = 'default'

The base class from which all generators must inherit.

generate(*dataset=None*, *localization=None*)

Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw()

is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generate without posterior treatment. All generators must implement this method. If a generator does not implement this method a NotImplementedError will be raised

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

Raises NotImplementedError

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.**EntityGenerator**
The class from which all composite generators must inherit.

Note: All clases inheriting from this class should not have any other methods than the ones listed below, for the sake of consistency.

---

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Gets all the attributes of the class and generates a new value.

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A dict where every key value pair is an attribute and its value

Raises *dammy.exceptions.DatasetRequiredException*

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

to_csv(*number*, *save_to*)
Save the specified amount of instances in a csv file

Parameters

- **number** (*int*) – The number of instances
- **save_to** (*str*) – The path to the file where the instances will be saved

to_json(*number*, *save_to=None*, *indent=4*)
Get the specified amount of instances as a list of json dicts

Parameters

- **number** (*int*) – The number of instances
- **save_to** (*str*) – The path where the generated json will be saved. If none, it will be returned as a string

Returns str containing the generated json if save_to=None, None in other cases

class dammy.**FunctionResult**(*function*, *obj*, **args*, ***kwargs*)
Allows the manipulation of generators by functions

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generate a value and call the function using the generated value as a parameter

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The result of running the generated value through the function

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.**AttributeGetter**(*obj*, *attr*)
Allows getting attribute values from values generated by generators

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generate a value and get the specified attribute

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value of the attribute on the generated object

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.**MethodCaller**(*obj*, *method*, **args*, ***kwargs*)
Allows calling methods of values generated by generators

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generate a value and call the specified method on the generated value

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value returned by the called method

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.**OperationResult**(*a*, *b*, *op*, *sql*)
Allows binary operations with regular and Dammy objects and it is returned when any of such operations is performed

class Operator
Enumerated type containing all the available operators

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generates a value and performs the operation. It will raise a TypeError if the operator is invalid.

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value returned after performing the operation

Raises TypeError

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

2.2 More modules

In addition to the main module, dammy contains other modules extending the functionalities

2.2.1 Databases and datasets

Database constraints and dataset generation.

this module contains everything database related, allowing you to create primary keys, foreign keys and autoincrement fields

The classes in the module are listed below.

class dammy.db.**AutoIncrement**(*start=1*, *increment=1*)
Represents an automatically incrementing field. By default starts by 1 and increments by 1

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generates and updates the next value

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved. It will be ignored.

Returns The next value of the sequence

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.db.**PrimaryKey**(*max_retries=100*, ***kwargs*)

Represents a primary key. Every field encapsulated by this class becomes a member of the primary key. A table cannot contain more than one primary key. This class is an alias of the Unique class, with the exception that no more than a primary key can exist on each table, but multiple unique values are supported.

Parameters **k** (dammy.db.BaseGenerator) – The fields which will be part of the primary key

In this example the primary key of A will be the field called 'primary' which is an autoincrement field:

```
from dammy import EntityGenerator
from dammy.db import PrimaryKey, AutoIncrement

class A(EntityGenerator):
    primary = PrimaryKey(AutoIncrement())
    # More attributes...
```

In this other example the primary key of B will be formed by the fields called 'field1' and 'field2':

```
from dammy import EntityGenerator
from dammy.db import PrimaryKey, AutoIncrement

class B(EntityGenerator):
    field1 = PrimaryKey(AutoIncrement())
    field2 = PrimaryKey(AutoIncrement())
    # More attributes...
```

generate(*dataset=None*, *localization=None*)

Generates a unique value

Implementation of the generate() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved.

Returns A unique value generated by the associated generator

Raises MaximumRetriesExceededException

generate_raw(*dataset=None*, *localization=None*)

Generates a unique value

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved.

Returns A unique value generated by the associated generator

Raises MaximumRetriesExceededException

iterator(*dataset=None*)

Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

reset()
Reset the uniqueness of the generator.

class dammy.db.**ForeignKey**(*ref_table*, *ref_field*)
Represents a foreign key. The first parameter is the class where the referenced field is and the second a list of strings, each of them containing the name of a field forming the primary key. If the referenced attribut is not unique or primay key, a InvalidReferenceException is raised

Parameters

- **ref_table** (dammy.db.EntityGenerator) – The table where the referenced field is
- ***args** (*str*) – List of the names of the fields forming the referenced key

Raises *dammy.exceptions.InvalidReferenceException*

generate(*dataset=None*, *localization=None*)
Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Gets the values corresponding to the key from the given dataset. If the dataset is not specified, a DatasetRequiredException will be raised.

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved.

Returns A unique value generated by the associated generator

Raises DatasetRequiredException

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

class dammy.db.**Unique**(*max_retries=100*, ***kwargs*)

Represents a unique field. The generator encapsulated here, will be guaranteed to generate unique values

Parameters

- **u** (BaseGenerator) – The generator which will generate unique values
- **max_retries** (*int*) – The number of times it will retry to generate the value when it has already been generated

generate(*dataset=None*, *localization=None*)

Generates a unique value

Implementation of the generate() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved.

Returns A unique value generated by the associated generator

Raises MaximumRetriesExceededException

generate_raw(*dataset=None*, *localization=None*)

Generates a unique value

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved.

Returns A unique value generated by the associated generator

Raises MaximumRetriesExceededException

iterator(*dataset=None*)

Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)

Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

reset()

Reset the uniqueness of the generator.

class dammy.db.**DatasetGenerator**(**args*)

generate(*dataset=None*, *localization=None*)

Generate a value and perform a posterior treatment. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns The value generated by the generator

generate_raw(*dataset=None*, *localization=None*)
Generate a new dataset with the previously given specifications

Implementation of the generate_raw() method from BaseGenerator.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A dict where every key value pair is an attribute and its value

Raises *dammy.exceptions.DatasetRequiredException*

iterator(*dataset=None*)
Get a iterator which generates values and performs a posterior treatment on them. By default, no treatment is done and generate_raw() is called.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns A Python iterator

iterator_raw(*dataset=None*)
Get a generator which generates values without posterior treatment.

Parameters **dataset** (*dammy.db.DatasetGenerator* or dict) – The dataset from which all referenced fields will be retrieved

Returns Python generator

Raises NotImplementedError

to_json(*save_to=None*, *indent=4*)
Get the JSON representation of the dataset. If a path is specified, a file is created and the resulting JSON is written to the file. If no path is given, the generated JSON will be returned.

Parameters

- **save_to** (*str*) – The path where the JSON will be saved
- **indent** (*int*) – The indentation level of the resulting JSON

Returns String containing the JSON encoded dataset or none if it has been written to a file

to_sql(*save_to=None*, *create_tables=True*)
Gets the dataset as SQL INSERT statements. The generated SQL is always returned and if save_to is specified, it is saved to that location. Additional CREATE TABLE statements are added if create_tables is set to True

Parameters

- **save_to** (*str*) – The path where the resulting SQL will be saved.
- **create_tables** (*bool*) – If set to true, it will generate the instructions to create the tables.

Returns A string with the SQL sentences required to insert all the tuples

2.2.2 Functions

Functions to be used on generators

This module includes some standard functions to be used on dammy objects

dammy.functions.**average**(*lst*)
Get the average value of a list

Parameters **lst** (*list*) – A list containing numeric values

Returns *dammy.FunctionResult*

dammy.functions.**call_function**(*obj*, *fun*, **args*, ***kwargs*)
Call a user given function. This function is useful when none of the available functions fulfills the task the user wants, so the user can feed the desired function here.

Parameters

- **obj** (*BaseDammy*) – The object on which the function will be called
- **fun** (*callable*) – The function to call
- **args** – Comma separated arguments for the function
- **kwargs** – Comma separated keyword identified arguments for the function

Returns *dammy.FunctionResult*

dammy.functions.**cast**(*obj*, *t*)
Casts an object to the specified type

Parameters

- **obj** – The object to cast
- **t** – The type

Returns *dammy.FunctionResult*

dammy.functions.**maximum**(*lst*)
Get the maximum value of a list

Parameters **lst** (*list*) – A list containing numeric values

Returns *dammy.FunctionResult*

dammy.functions.**minimum**(*lst*)
Get the minimum value of a list

Parameters **lst** (*list*) – A list containing numeric values

Returns *dammy.FunctionResult*

2.2.3 Built-in generators

All the builtins.

This module contains some basic and common utilities, such as random generation for person names, countries, integers, strings, dates…

2.2.4 Exceptions

All the exceptions raised by dammy are included in this module.

This module contains all the custom exceptions used by dammy.

exception dammy.exceptions.**DammyException**
The base exception from which all dammy exceptions inherit

Note: This exception should not be raised by any means. Its only purpose is to check in a try-catch block wether an exception has been raised by dammy or not.

---

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception dammy.exceptions.**DatasetRequiredException**
This exception is raised when a generator requires a dataset to get references from it but no dataset is given

Example:

```
from dammy import EntityGenerator
from dammy.stdlib import RandomInteger

class B(EntityGenerator):
    attribute1 = RandomInteger(15, 546)
    reference_to_A = ForeignKey(A, 'attribute1')

B().generate()  # generate() requires a dataset because B contains a reference to␣
↪another generator
                # thus, this call will result in a DatasetRequiredException
```

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception dammy.exceptions.**EmptyKeyException**
Raised when a primary key or a unique filed is empty

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception dammy.exceptions.**IntegrityException**
Raised when database integrity rules are violated

Example:

```
from dammy import EntityGenerator
from dammy.db import ForeignKey
from dammy.stdlib import RandomInteger

class A(EntityGenerator):
    attribute1 = RandomInteger(1, 10)
    attribute2 = RandomInteger(2, 5)

class B(EntityGenerator):
    attribute1 = RandomInteger(15, 546)
    reference_to_A = ForeignKey(A, 'attribute1')
```

This example will raise a IntegrityException because reference_to_A in class B is a foreign key referencing the field attribute1 from class A, and A.attribute1 is not a primary key

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception dammy.exceptions.**InvalidReferenceException**
Raised when a foreign key references a field which is not a primary key or unique

This is commonly raised when the amount of generated data is bigger than the available data on a generator obtaining its data from a file.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception dammy.exceptions.**MaximumRetriesExceededException**
Raised when a unique field exceeds the maximum number of retries to generate a unique value

This is commonly raised when the amount of generated data is bigger than the available data on a generator obtaining its data from a file.

Example:

```
from dammy.db import Unique
from dammy.stdlib import RandomInteger

x = Unique(RandomInteger(1, 10))

for i in range(0, 50):
    print(x)            # Exception after generating 10 values
```

The code will result in an exception because it will try to generate 50 unique integers in the [1, 10] interval, which is obviously impossible because there are 10 unique integers available

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER 3

API reference

This page is not currently available

CHAPTER 4

Contributing

The ways to contribute to the project are - You can fork the GitHub repository make your changes and perform a pull request.

- The same applies to documentation translations or improvements if you have knowledge of ReStructuredText and Sphinx.
- You can open a new issue requesting new features, submitting bugs…
- Spread the word
- Donate (Not available yet)

CHAPTER 5

Tutorial

Tutorials are a great way to start, and much more intuitive than the *Documentation*. If you want a quickstart or are having a hard time with the documentation, check the *Tutorial*. page.

CHAPTER 6

Documentation

The documentation explains every detail of how dammy works. Every function, class, and parameter is well documented. Check the *Documentation*. page.

CHAPTER 7

API Reference

If you are thinking on creating your own generator, you might want to check the *API reference*. page.

CHAPTER 8

Contribute

If you want to contribute to the project, please read the *Contributing*. page.

Python Module Index

d

Index

M

O

P

R

T

U

W