Recipe

Release 0.31.5

Chris Gemignani

Jun 13, 2022

GETTING STARTED

Release v0.31.5. (*Installation*)

Recipe is an MIT licensed cross-database querying library, written in Python. It allows you to reuse SQL fragments to answer data questions consistently. Extension classes allow you to support data anonymization, automatic generation of where clauses, user permissioning to data, subselects, and response formatting.

```
>>> shelf = Shelf({ 'age': WtdAvgMetric(Census.age, Census.pop2000), 'state':␣
↪Dimension(Census.state)})
>>> recipe = Recipe().shelf(shelf).metrics('age').dimensions('state').order_by('-age')

>>> recipe.to_sql()
SELECT census.state AS state,
       CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.
↪pop2000) AS FLOAT), 0.0) + 1e-09) AS age
FROM census
GROUP BY census.state
ORDER BY CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.
↪pop2000) AS FLOAT), 0.0) + 1e-09) DESC

>>> recipe.dataset.csv
state,age,state_id
Florida,39.08283934000634,Florida
West Virginia,38.555058651148165,West Virginia
Maine,38.10118393261269,Maine
Pennsylvania,38.03856695544053,Pennsylvania
...
```

CHAPTER

ONE

INTRODUCTION

Recipe is a cross-database querying library, written in Python. It allows you to reuse SQL fragments that can be composed into queries. An extension classes allow you to support data anonymization, automatic generation of where clauses, subselects, and response formatting.

1.1 Recipe License

Recipe is released under terms of The MIT License.

```
Copyright 2017 Chris Gemignani

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

1.2 Pythons Supported

At this time, the following Python platforms are officially supported:

- cPython 3.6

Support for other Pythons will be rolled out soon.

Now, go *Installing Recipe*.

CHAPTER

TWO

INSTALLATION

2.1 Installing Recipe

2.1.1 Distribute & Pip

The recommended way to install Recipe is with pip:

```
$ pip install recipe
```

2.2 Download the Source

You can also install recipe from source. The latest release (0.31.5) is available from GitHub.

- tarball
- zipball

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily.

```
$ python setup.py install
```

2.2.1 Staying Updated

The latest version of Recipe will always be available here:

- PyPi: http://pypi.python.org/pypi/recipe/
- GitHub: http://github.com/juiceinc/recipe/

When a new version is available, upgrading is simple:

```
$ pip install recipe --upgrade
```

Now, go get a *Quick Start*.

CHAPTER

THREE

GETTING STARTED

This page gives a good introduction in how to get started with Recipe. This assumes you already have Recipe installed. If you do not, head over to *Installing Recipe*.

First, make sure that:

- Recipe is *installed*
- Recipe is *up-to-date*

Let's gets started with some simple use cases and examples.

3.1 Creating a Shelf

A *Shelf* is a place to store SQL fragments. In recipe these are called *Ingredients*.

Ingredients can contain columns that should be part of the `SELECT` portion of a query, filters that are part of a `WHERE` clause of a query, group_bys that contribute to a query's `GROUP BY` and havings which add `HAVING` limits to a query.

You won't have to construct an Ingredient with all these parts directly because Recipe contains convenience classes that help you build the most common SQL fragments. The two most common Ingredient subclasses are *Dimension* which provides both a column and a grouping on that column and *Metric* which provides a column aggregation.

Shelf acts like a dictionary. The keys are strings and the values are Ingredients. The keys are a shortcut name for the ingredient. Here's an example.

```
from recipe import *

# Define a database connection
oven = get_oven('sqlite://')
Base = declarative_base(bind=oven.engine)

# Define a SQLAlchemy mapping
class Census(Base):
    state = Column('state', String(), primary_key=True)
    sex = Column('sex', String())
    age = Column('age', Integer())
    pop2000 = Column('pop2000', Integer())
    pop2008 = Column('pop2008', Integer())

    __tablename__ = 'census'
    __table_args__ = {'extend_existing': True}
```

(continues on next page)

(continued from previous page)

```
# Use that mapping to define a shelf.
shelf = Shelf({
    'state': Dimension(Census.state),
    'age': WtdAvgMetric(Census.age, Census.pop2000),
    'population': Metric(func.sum(Census.pop2000))
})
```

This is a shelf with two metrics (a weighted average of age, and the sum of population) and a dimension which lets you group on US State names.

3.2 Using the Shelf to build a Recipe

Now that you have the shelf, you can build a *Recipe*.

```
r = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state')\
    .metrics('age')\
    .order_by('-age')

print(r.dataset.csv)
```

This results in

```
state,age,state_id
Florida,39.08283934000634,Florida
West Virginia,38.555058651148165,West Virginia
Maine,38.10118393261269,Maine
Pennsylvania,38.03856695544053,Pennsylvania
Rhode Island,37.20343773873182,Rhode Island
Connecticut,37.19867141455273,Connecticut
...
```

Note that a recipe contains data from a single table.`

3.3 Defining Shelves and Recipes Using Configuration

Recipes and shelves can be defined using plain ole' python objects. In the following example we'll use YAML. For instance, we can define the shelf using this yaml config.

```
state:
    kind: Dimension
    field: state
age:
    kind: WtdAvgMetric
    field: age
    weight: pop2000
population:
    kind: Metric
    field: pop2000
```

We can load this config by parsing it against any **selectable**, which can be a SQLAlchemy mapping, a SQLAlchemy select, or another Recipe.

```
shelf_yaml = yaml.load('shelf.yaml')
s = Shelf.from_config(shelf_yaml, Census)
```

We can also define a Recipe with Configuration

```
metrics:
- age
- population
dimensions:
- state
order_by:
- '-age'
```

If we load that we get a Recipe

```
recipe_yaml = yaml.load('shelf.yaml')
recipe = Recipe.from_config(s, recipe_yaml, session=oven.Session())
print(recipe.dataset.csv)
```

This results in a list of the oldest US states and their populations:

```
state,age,population,state_id
Florida,39.08283934000634,15976093,Florida
West Virginia,38.555058651148165,1805847,West Virginia
Maine,38.10118393261269,1271694,Maine
Pennsylvania,38.03856695544053,12276157,Pennsylvania
Rhode Island,37.20343773873182,1047200,Rhode Island
Connecticut,37.19867141455273,3403620,Connecticut
...
```

3.4 Adding Features with Extensions

Using extensions, you can add features to Recipe. Here are a few interesting thing you can do. This example mixes in two extensions.

AutomaticFilters defines filters (where clauses) using configuration. In this case were are filtering to states that start with the letter C.

CompareRecipe mixes in results from another recipe. In this case, we are using this comparison recipe to calculate an average age across all states.

```
recipe_yaml = yaml.load(r)
recipe = Recipe.from_config(s, recipe_yaml, session=oven.Session(),
    extension_classes=(AutomaticFilters, CompareRecipe))\
    .automatic_filters({'state__like': 'C%'})\
    .compare(Recipe(shelf=s, session=oven.Session()).metrics('age'))
print(recipe.to_sql())
print()
print(recipe.dataset.csv)
```

The output looks like this

```
SELECT census.state AS state,
    CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.pop2000)␣
↪AS FLOAT), 0.0) + 1e-09) AS age,
    sum(census.pop2000) AS population,
    avg(anon_1.age) AS age_compare
FROM census
LEFT OUTER JOIN
(SELECT CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.
↪pop2000) AS FLOAT), 0.0) + 1e-09) AS age
FROM census) AS anon_1 ON 1=1
WHERE census.state LIKE 'C%'
GROUP BY census.state
ORDER BY CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.
↪pop2000) AS FLOAT), 0.0) + 1e-09) DESC

state,age,population,age_compare,state_id
Connecticut,37.19867141455273,3403620,35.789568740450036,Connecticut
Colorado,34.5386073584527,4300877,35.789568740450036,Colorado
California,34.17872597484759,33829442,35.789568740450036,California
```

Now, go check out the *API Documentation* or look at an *Overview of Recipe Concepts*.

CHAPTER

FOUR

OVERVIEW OF RECIPE CONCEPTS

Ingredients are reusable fragments of SQL defined in SQLAlchemy. Ingredients can contribute to a SQL query's select, group by, where clause or having clause. For convenience, we define **Metric**, **Dimension**, **Filter**, and **Having** classes which support common query patterns.

A **Shelf** is a container for holding named ingredients. Shelves can be defined with python code or via configuration. Shelves defined with configuration can be bound to a SQLAlchemy selectable.

Note: By convention, all the ingredients on a Shelf should reference the same SQLAlchemy selectable.

A **Recipe** uses a **Shelf**. The Recipe picks dimensions, metrics, filters, and havings from the shelf. Dimensions and metrics can also be used to order results. While the Recipe can refer to items in the shelf by name, you can also supply raw Ingredient objects. Recipe uses a builder pattern to allow a recipe object to be modified.

A Recipe generates and runs a SQL query using SQLAlchemy. The query uses an **Oven** an abstraction on top of a SQLAlchemy connection. The query results are "enchanted" which adds additional properties to each result row. This allows ingredients to format or transform values with python code.

Recipe results can optionally be cached with the recipe_caching support library.

4.1 Extensions

Extensions add to Recipe to change how SQL queries get built.

Recipe includes the following built-in extensions.

- **AutomaticFilter**: Supports a configuration syntax for applying filters.
- **SummarizeOver**: Supports summarizing over a dimension
- **BlendRecipe**: Allows data from different tables to be combined
- **CompareRecipe**: Allows a secondary recipe against the same table to be combined.
- **Anonymize**: Allows result data to be anonymized.

CHAPTER

FIVE

INGREDIENTS

Ingredients are the building block of recipe.

Ingredients can contain columns that are part of the `SELECT` portion of a query, filters that are part of a `WHERE` clause of a query, group_bys that contribute to a query's `GROUP BY` and havings which add `HAVING` limits ot a query.

5.1 Creating ingredients in python

Ingredients can be created either in python or via configuration. To created Ingredients in python, use one of the four convenience classes.

- **Metric**: Create an aggregated calculation using a column. This value appears only in the SELECT part of the SQL statement.
- **Dimension**: Create a non-aggregated value using a column. This value appears in the SELECT and GROUP BY parts of the SQL statement.
- **Filter**: Create a boolean expression. This value appears in the WHERE part of the SQL statement. Filters can be created automatically using the AutomaticFilters extension or by using a Dimension or Metric'sales `build_filter` method.
- **Having**: Create a boolean expression with an aggregated ColumnElement. This value appears in the HAVING part of the SQL statement.

Metrics and Dimensions are commonly reused in working Recipe code, while filters are often created temporarily based on data.

5.1.1 Features of ingredients

Let's explore some capabilities.

Formatters

Formatters are a list of python callables that take a single value. This let you manipulate the results of an ingredient with python code. If you use formatters, the original, unmodified value is available as `{ingredient}_raw`.

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'age': WtdAvgMetric(Census.age, Census.pop2000),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.pop2000), formatters=[
```

(continues on next page)

(continued from previous page)

```
        lambda value: int(round(value, -6) / 1000000)
    ])
})

recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('gender').metrics('population')

for row in recipe.all():
    print('{} has {} people'.format(row.gender, row.population))
    print('\tThe original value is: {}'.format(row.population_raw))
```

The results look like

```
F has 144 million people
    The original value is: 143534804
M has 137 million people
    The original value is: 137392517
```

Building filters

Ingredient.build_filter

Storing extra attributes in meta

Extra keyword arguments that get passed to ingredient initialization get stored in the `meta` object. This can be used to extend the capabilities of ingredients and add extra features.

```
d = Dimension(Census.age, icon='cog')
print(d.meta.icon)
>>> 'cog'
```

5.2 Types of Ingredients

List of ingredients

5.2.1 Dimension

Dimensions are groupings that exist in your data. Dimension objects add the column to the select statement and the group by of the SQL query.

```
# A simple dimension
self.shelf['state'] = Dimension(Census.state)
```

Adding an id

Dimensions can use separate columns for ids and values. Consider a table of employees with an `employee_id` and a `full_name`. If you had two employees with the same name you need to be able to distinguish between them.

```
# Support an id and a label
self.shelf['employee']: Dimension(Employee.full_name,
                                  id_expression=Employee.id)
```

The id is accessible as `employee_id` in each row and their full name is available as `employee`.

If you build a filter using this dimension, it will filter against the id.

Adding an ordering

If you want to order a dimension in a custom way, pass a keyword argument `order_by_expression`. This code adds an order_by_expression that causes the values to sort case insensitively.

```
from sqlalchemy import func

# Support an id and a label
self.shelf['employee']: Dimension(Employee.full_name,
                                  order_by_expression=func.lower(
                                    Employee.full_name
                                  ))
```

The order_by expression is accessible as `employee_order_by` in each row and the full name is available as `employee`. If the *employee* dimension is used in a recipe, the recipe will **always** be ordered by `func.lower(Employee.full_name)`.

Adding additional groupings

Both `id_expression` and `order_by_expression` are special cases of Dimension's ability to be passed additional columns can be used for grouping. Any keyword argument suffixed with `_expression` adds additional roles to this Dimension. The first *required* expression supplies the dimension's value role. For instance, you could create a dimension with an `id`, a `latitude` and a `longitude`.

For instance, the following

```
Dimension(Hospitals.name,
          latitude_expression=Hospitals.lat
          longitude_expression=Hospitals.lng,
          id='hospital')
```

would add columns named "hospital", "hospital_latitude", and "hospital_longitude" to the recipes results. All three of these expressions would be used as group bys.

Using lookups

You can use a lookup table to map values in your data to descriptive names. The `_id` property of your dimension contains the original value.

```
# Convert M/F into Male/Female
self.shelf['gender']: Dimension(Census.sex, lookup={'M': 'Male',
    'F': 'Female'}, lookup_default='Unknown')
```

If you use the gender dimension, there will be a `gender_id` in each row that will be "M" or "F" and a `gender` in each row that will be "Male" or "Female".

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender_desc': Dimension(Census.gender, lookup={'M': 'Male',
        'F': 'Female'}, lookup_default='Unknown'),
    'age': WtdAvgMetric(Census.age, Census.pop2000),
    'population': Metric(func.sum(Census.pop2000))
})

recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('gender_desc').metrics('population')
print(recipe.to_sql())
print(recipe.dataset.csv)
```

Lookups inject a formatter in the first position. Because a formatter is used, recipe creates a `gender_desc_raw` on the response that contains the unformatted value then uses the lookup to create the `gender_desc` property. All dimensions also generate an `{ingredient}_id` property.

Here is the query and the results.

```
SELECT census.gender AS gender_desc_raw,
    sum(census.pop2000) AS population
FROM census
GROUP BY census.gender

gender_desc_raw,population,gender_desc,gender_desc_id
F,143534804,Female,F
M,137392517,Male,M
```

5.2.2 Metric

Metrics are aggregations performed on your data. Here's an example of a few Metrics.

```
shelf = Shelf({
    'total_population': Metric(func.sum(Census.pop2000)),
    'min_population': Metric(func.min(Census.pop2000)),
    'max_population': Metric(func.max(Census.pop2000))
})
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .metrics('total_population', 'min_population', 'max_population')
print(recipe.to_sql())
print(recipe.dataset.csv)
```

The results of this recipe are:

```
SELECT max(census.pop2000) AS max_population,
    min(census.pop2000) AS min_population,
    sum(census.pop2000) AS total_population
FROM census

max_population,min_population,total_population
294583,217,280927321
```

5.2.3 DivideMetric

Division in SQL introduces the possibility of division by zero. DivideMetric guards against division by zero while giving you a quick way to divide one calculation by another.

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'popgrowth': DivideMetric(func.sum(Census.pop2008-Census.pop2000), func.sum(Census.
↪pop2000)),
})
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state').metrics('popgrowth')
```

This creates results like:

```
SELECT census.state AS state,
    CAST(sum(census.pop2008 - census.pop2000) AS FLOAT) /
      (coalesce(CAST(sum(census.pop2000) AS FLOAT), 0.0) + 1e-09) AS popgrowth
FROM census
GROUP BY census.state

state,popgrowth,state_id
Alabama,0.04749469366071285,Alabama
Alaska,0.09194726152996757,Alaska
Arizona,0.2598860676785905,Arizona
Arkansas,0.06585681816651036,Arkansas
California,0.0821639328251409,California
Colorado,0.14231283526592364,Colorado
...
```

The denominator has a tiny value added to it to prevent division by zero.

5.2.4 WtdAvgMetric

`WtdAvgMetric` generates a weighted average of a number using a weighting.

> **Warning:** `WtdAvgMetric` takes two ColumnElements as arguments. The first is the value and the second is the weighting. Unlike other Metrics, these are **not aggregated**.

Here's an example.

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'avgage': WtdAvgMetric(Census.age, Census.pop2000),
})
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state').metrics('avgage')

print(recipe.to_sql())
print(recipe.dataset.csv)
```

This generates results that look like this:

```
SELECT census.state AS state,
    CAST(sum(census.age * census.pop2000) AS FLOAT) / (coalesce(CAST(sum(census.pop2000)␣
↪AS FLOAT), 0.0) + 1e-09) AS avgage
FROM census
GROUP BY census.state

state,avgage,state_id
Alabama,36.27787892421841,Alabama
Alaska,31.947384766048568,Alaska
Arizona,35.37065466080318,Arizona
Arkansas,36.63745110262778,Arkansas
California,34.17872597484759,California
...
```

Note: WtdAvgMetric uses safe division from `DivideMetric`.

5.2.5 Filter

Filter objects add a condition to the where clause of your SQL query. Filter objects can be added to a Shelf.

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'population': Metric(func.sum(Census.pop2000)),
    'teens': Filter(Census.age.between(13,19)),
})
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state')\
    .metrics('population')\
    .filters('teens')
print(recipe.to_sql())
print(recipe.dataset.csv)
```

This results in output like:

```
SELECT census.state AS state,
    sum(census.pop2000) AS population
FROM census
WHERE census.age BETWEEN 13 AND 19
GROUP BY census.state
```

(continues on next page)

(continued from previous page)

```
state,population,state_id
Alabama,451765,Alabama
Alaska,71655,Alaska
Arizona,516270,Arizona
```

Different ways of generating Filters

Recipe has several ways of filtering recipes.

- **Filter objects can be added to the shelf**. They can be added to the recipe by name from a shelf. This is best when you have a filter that you want to use in many place.

```
shelf = Shelf({
    'age': Dimension(Census.age),
    'state': Dimension(Census.state),
    'population': Metric(func.sum(Census.pop2000)),
    'teens': Filter(Census.age.between(13,19)),
})
...
recipe = recipe.filters('teens')
```

- **Filter objects can be created dynamically** and added to the recipe. This is best if the filtering needs to change dynamically.

```
recipe = recipe.filters(Filter(Census.age.between(13,19))
```

- **Ingredient.build_filter** can be used to build filters that refer to the ingredient's column.

```
age_filter = shelf['age'].build_filter([13,19], 'between')
recipe = recipe.filters(age_filter)
```

  This is best when you want to reuse a column definition defined in an ingredient.

- **AutomaticFilters**: The AutomaticFilters extension adds filtering syntax directly to recipe.

```
recipe = recipe.automatic_filters({
  'age__between': [13,19]
})
```

  This is best when you want to add many filters consistently. AutomaticFilters uses `Ingredient.build_filter` behind the scenes.

5.2.6 Having

Having objects are binary expressions with an aggregated column value. One easy way to generate `Having` objects is to `build_filter` using a `Metric`.

```
shelf = Shelf({
    'age': Dimension(Census.age),
    'avgage': WtdAvgMetric(Census.age, Census.pop2000),
    'state': Dimension(Census.state),
    'population': Metric(func.sum(Census.pop2000)),
```

(continues on next page)

(continued from previous page)

```
})
# Find states with a population greater than 15 million
big_states = shelf['population'].build_filter(15000000, operator='gt')
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state')\
    .metrics('population')\
    .order_by('-population')\
    .filters(big_states)

print(recipe.to_sql())
print(recipe.dataset.csv)
```

This generates the following results.

```
SELECT census.state AS state,
    sum(census.pop2000) AS population
FROM census
GROUP BY census.state
HAVING sum(census.pop2000) > 15000000
ORDER BY sum(census.pop2000) DESC

state,population,state_id
California,33829442,California
Texas,20830810,Texas
New York,18978668,New York
Florida,15976093,Florida
```

CHAPTER

SIX

SHELVES

A shelf is a container for holding Ingredients.

Shelves act like dictionaries with keys that are strings and values that are Ingredients.

Adding ingredients to a Shelf sets the ingredient `id` to the key used in the Shelf.

CHAPTER

SEVEN

DEFINING SHELVES FROM CONFIGURATION

Shelves are defined as dictionaries containing keys and ingredient. All the examples below use YAML.

7.1 Defining Shelves

Shelves are defined in configuration as dictionaries with keys and values that are Ingredient configuration definitions. A simple example looks like this.

```
total_population:
  kind: Metric
  field: pop2000
state:
  kind: Dimension
  field: state
```

See *examples* for more Shelf examples.

7.2 Defining Ingredients

Ingredients are defined using *fields* (which may contain *conditions*). Those *conditions* may reference more *fields* in turn and so forth.

7.2.1 Metric

Metrics will always apply a default aggregation of 'sum' to any fields used.

```
kind: Metric
field: {field}
divide_by: {field} (optional)
```

`divide_by` is an optional denominator that `field` will be divided by safely.

7.2.2 Dimension

Metrics will always apply a default aggregation of 'sum' to their field.

```
kind: Dimension
field: {field}
{role}_field: {field} (optional)
buckets: A list of labeled conditions (optional)
buckets_default_label: string (optional)
quickselects: A list of labeled conditions (optional)
```

Adding *id* and other roles to Dimension

Dimensions can be defined with extra fields. The prefix before `_field` is the field's role. The role will be suffixed to each value in the recipe rows. Let's look at an example.

```
hospital:
  field: hospital_name
  id_field: hospital_id
  latitude_field: hospital_lat
  longitude_field: hospital_lng
```

Each result row will include

- `hospital`
- `hospital_id` The field defined as `id_field`
- `hospital_latitude` The field defined as `latitude_field`
- `hospital_longitude` The field defined as `longitude_field`

Defining buckets

Buckets let you group continuous values (like salaries or ages). Here's an example:

```
groups:
    kind: Dimension
    field: age
    buckets:
    - label: 'northeasterners'
      field: state
      in: ['Vermont', 'New Hampshire']
    - label: 'babies'
      lt: 2
    - label: 'children'
      lt: 13
    - label: 'teens'
      lt: 20
    buckets_default_label: 'oldsters'
```

The conditions are evaluated **in order**. **buckets_default_label** is used for any values that didn't match any condition.

For convenience, conditions defined in buckets will use the field from the Dimension unless a different field is defined in the condition. In the example above, the first bucket uses `field: state` explicitly while all the other conditions use `field: age` from the Dimension.

If you use order_by a bucket dimension, the order will be the order in which the buckets were defined.

Adding quickselects to a Dimension

quickselects are a way of associating conditions with a dimension.

```
region:
    kind: Dimension
    field: sales_region
total_sales:
    kind: Metric
    field: sales_dollars
date:
    kind: Dimension
    field: sales_date
    quickselects:
    - label: 'Last 90 days'
      between:
      - 90 days ago
      - tomorrow
    - label: 'Last 180 days'
      between:
      - 180 days ago
      - tomorrow
```

These conditions can then be accessed through `Ingredient.build_filter`. The `AutomaticFilters` extension is an easy way to use this.

```
recipe = Recipe(session=oven.Session(), extension_classes=[AutomaticFilters]). \
            .dimensions('region') \
            .metrics('total_sales') \
            .automatic_filters({
              'date__quickselect': 'Last 90 days'
            })
```

7.3 Defining Fields

Fields can be defined with a short string syntax or a dictionary syntax. The string syntax always is normalized into the dictionary syntax.

```
field:
    value: '{column reference}'
    aggregation: '{aggregation (optional)}'
    operators: {list of operators}
    as: {optional type to coerce into}
    default: {default value, optional}
```

(continues on next page)

(continued from previous page)

```
or

field: '{string field definition}'
This may include field references that look like
@{ingredient name from the shelf}.
```

7.3.1 Defining Fields with Dicts

Dictionaries provide access to all options when defining a field.

Table 1: dictionary field options

Key	Required	Description
value	required	string What column to use.
aggregation	optional	string (default is 'sum' for Metric and 'none' for Dimension) What aggregation to use, if any. Possible aggregations are: • 'sum' • 'min' • 'max' • 'avg' • 'count' • 'count_distinct' • 'month' (round to the nearest month for dates) • 'week' (round to the nearest week for dates) • 'year' (round to the nearest year for dates) • 'quarter' (round to the nearest quarter for dates) • 'age' (calculate age based on a date and the current date) • 'none' (perform no aggregation) • 'median' (calculate the median value, note: this aggregation is not available on all databases). • 'percentile[1,5,10,25,50,75,90,95,99]' (calculate the nth percentile value where higher values correspond to higher percentiles, note: this aggregation is not available on all databases).
condition	optional	A `condition` Condition will limit what rows of data are aggregated for a field.
operators	optional	A list of `operator` Operators are fields combined with a math operator to the base field.
default	optional	An integer, string, float, or boolean value (optional) A value to use if the column is NULL.

Warning: The following two fields are for internal use.

Table 2: internal dictionary field options

Key	Required	Description
ref	optional	string Replace this field with the field defined in the specified key in the shelf.
_use_raw_value	optional	boolean Don't evaluate value as a column, treat it as a constant in the SQL expression.

7.3.2 Defining Fields with Strings

Fields can be defined using strings. When using strings, words are treated as column references. If the words are prefixed with an '@' (like @sales), the field of the ingredient named sales in the shelf will be injected.

Aggregations can be called like functions to apply that aggregation to a column.

Table 3: string field examples

Field string	Description

revenue - expenses	The sum of column revenue minus the sum of column expenses.
	`field: revenue - expenses`
	
	`# is the same as`
	
	`field:`
	`  value: revenue`
	`  aggregation: sum  # this may be omitted because 'sum'`
	`  # is the default aggregation for Metrics`
	`  operators:`
	`  - operator: '-'`
	`  field:`
	`  value: expenses`
	`  aggregation: sum`

@sales / @student_count	Find the field definition of the field named 'sales' in the shelf. Divide it by the field definition of the field named 'student_count'.

count_distinct(student_id)	Count the distinct values of column student_id.
	`field: count_distinct(student_id)`
	
	`# is the same as`
	
	`field:`
	`  value: student_id`
	`  aggregation: count_distinct`

7.4 Defining Field Operators

Operators lets you perform math with fields.

Table 4: operator options

Key	Required	Description
operator	required	string One of '+', '-', '*', '/'
field	required	A field definition (either a string or a dictionary)

For instance, operators can be used like this:

```
# profit - taxes - interest
field:
  value: profit
  operators:
  - operator: '-'
    field: taxes
  - operator: '-'
    field: interest
```

7.5 Defining Conditions

Conditions can include a field and operator or a list of conditions and-ed or or-ed together.

```
field: {field definition}
label: string (an optional string label)
{operator}: {value} or {list of values}

or

or:     # a list of conditions
- {condition1}
- {condition2}
...
- {conditionN}

or

and:    # a list of conditions
- {condition1}
- {condition2}
...
- {conditionN}

or

a condition reference @{ingredient name from the shelf}.
```

Conditions consist of a field and **exactly one** operator.

Table 5: condition options

Condition	Value is...	Description

gt	A string, int, or float.	Find values that are greater than the value
		For example:
		`# Sales dollars are greater than 100.`
		`condition:`
		`  field: sales_dollars`
		`  gt: 100`

gte (or ge)	A string, int, or float.	Find values that are greater than or equal to the value
lt	A string, int, or float.	Find values that are less than the value
lte (or le)	A string, int, or float.	Find values that are less than or equal to the value
eq	A string, int, or float.	Find values that are equal to the value
ne	A string, int, or float.	Find values that are not equal to the value

like	A string	Find values that match the SQL LIKE expression
		For example:
		`# States that start with the capital letter C`
		`condition:`
		`  field: state`
		`  like: 'C%'`

ilike	A string	Find values that match the SQL ILIKE (case insensitive like) expression.
between	A list of **two** values	Find values that are between the two values.
in	A list of values	Find values that are in the list of values
notin	A list of values	Find values that are not in the list of values

7.5.1 ands and ors in conditions

Conditions can `and` and `or` a list of conditions together.

Here's an example:

```
# Find states that start with 'C' and end with 'a'
# Note the conditions in the list don't have to
# use the same field.
condition:
```

(continues on next page)

(continued from previous page)

```
  and:
  - field: state
    like: 'C%'
  - field: state
    like: '%a'
```

7.5.2 Date conditions

If the `field` is a date or datetime, absolute and relative dates can be defined in values using string syntax. Recipe uses the Dateparser library.

Here's an example.

```
# Find sales that occured within the last 90 days.
condition:
  field: sales_date
  between:
  - '90 days ago'
  - 'tomorrow'
```

7.5.3 Labeled conditions

Conditions may optionally be labeled by adding a label property.

quickselects are a feature of Dimension that are defined with a list of labeled conditions.

7.6 Examples

7.6.1 A simple shelf with conditions

This shelf is basic.

```
teens:
    kind: Metric
    field:
        value: pop2000
        condition:
            field: age
            between: [13,19]
state:
    kind: Dimension
    field: state
```

Using this shelf in a recipe.

```
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state')\
    .metrics('teens')
```

(continues on next page)

(continued from previous page)

```
print(recipe.to_sql())
print(recipe.dataset.csv)
```

The results look like:

```
SELECT census.state AS state,
       sum(CASE
               WHEN (census.age BETWEEN 13 AND 19) THEN census.pop2000
           END) AS teens
FROM census
GROUP BY census.state

state,teens,state_id
Alabama,451765,Alabama
Alaska,71655,Alaska
Arizona,516270,Arizona
Arkansas,276069,Arkansas
...
```

7.6.2 Metrics referencing other metric definitions

The following shelf has a Metric `pct_teens` that divides one previously defined Metric `teens` by another `total_pop`.

```
teens:
    kind: Metric
    field:
        value: pop2000
        condition:
            field: age
            between: [13,19]
total_pop:
    kind: Metric
    field: pop2000
pct_teens:
    field: '@teens'
    divide_by: '@total_pop'
state:
    kind: Dimension
    field: state
```

Using this shelf in a recipe.

```
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('state')\
    .metrics('pct_teens')
print(recipe.to_sql())
print(recipe.dataset.csv)
```

Here's the results. Note that recipe performs safe division.

```
SELECT census.state AS state,
      CAST(sum(CASE
                   WHEN (census.age BETWEEN 13 AND 19) THEN census.pop2000
               END) AS FLOAT) / (coalesce(CAST(sum(census.pop2000) AS FLOAT), 0.0) + 1e-
↪09) AS pct_teens
FROM census
GROUP BY census.state

state,pct_teens,state_id
Alabama,0.10178190714599038,Alabama
Alaska,0.11773975168751254,Alaska
Arizona,0.10036487658951877,Arizona
Arkansas,0.10330245760980436,Arkansas
...
```

7.6.3 Dimensions containing buckets

Dimensions may be created by bucketing a field.

```
total_pop:
    kind: Metric
    field: pop2000
age_buckets:
    kind: Dimension
    field: age
    buckets:
    - label: 'babies'
      lt: 2
    - label: 'children'
      lt: 13
    - label: 'teens'
      lt: 20
    buckets_default_label: 'oldsters'
mixed_buckets:
    kind: Dimension
    field: age
    buckets:
    - label: 'northeasterners'
      in: ['Vermont', 'New Hampshire']
      field: state
    - label: 'babies'
      lt: 2
    - label: 'children'
      lt: 13
    - label: 'teens'
      lt: 20
    buckets_default_label: 'oldsters'
```

Using this shelf in a recipe.

```
recipe = Recipe(shelf=shelf, session=oven.Session())\
    .dimensions('mixed_buckets')\
```

(continues on next page)

(continued from previous page)

```
    .metrics('total_pop')\
    .order_by('mixed_buckets')
print(recipe.to_sql())
print(recipe.dataset.csv)
```

Here's the results. Note this recipe orders by `mixed_buckets`. The buckets are ordered in the **order they are defined**.

```
SELECT CASE
          WHEN (census.state IN ('Vermont',
                                 'New Hampshire')) THEN 'northeasterners'
          WHEN (census.age < 2) THEN 'babies'
          WHEN (census.age < 13) THEN 'children'
          WHEN (census.age < 20) THEN 'teens'
          ELSE 'oldsters'
      END AS mixed_buckets,
      sum(census.pop2000) AS total_pop
FROM census
GROUP BY CASE
            WHEN (census.state IN ('Vermont',
                                   'New Hampshire')) THEN 'northeasterners'
            WHEN (census.age < 2) THEN 'babies'
            WHEN (census.age < 13) THEN 'children'
            WHEN (census.age < 20) THEN 'teens'
            ELSE 'oldsters'
        END
ORDER BY CASE
            WHEN (census.state IN ('Vermont',
                                   'New Hampshire')) THEN 0
            WHEN (census.age < 2) THEN 1
            WHEN (census.age < 13) THEN 2
            WHEN (census.age < 20) THEN 3
            ELSE 9999
        END

mixed_buckets,total_pop,mixed_buckets_id
northeasterners,1848787,northeasterners
babies,7613225,babies
children,44267889,children
teens,28041679,teens
oldsters,199155741,oldsters
```

CHAPTER

EIGHT

USING SHELVES FROM CONFIGURATION

8.1 When are Shelves from configuration bound to columns?

Shelf configuration can be **bound** at any time to a selectable. This can be any one of:

- A SQLAlchemy Mapping
- A SQLAlchemy subselect
- A Recipe

8.2 Binding a shelf to a Mapping

Binding shelves to Mappings is the most common usage of shelves. It connects the shelf config to database table columns.

Let's look at an example of binding a shelf to a Mapping.

```
Create simple census shelf
Average age by state
Get min/max average ages
```

The results look like this:

```
dfs
```

8.3 Binding a shelf to a SQLAlchemy subselect

Binding shelves to Mappings is the most common usage of shelves. It connects the shelf config to database table columns.

Let's look at an example of binding a shelf to a Mapping.

```
Create a subselect that joins the table to additional data
```

The results look like this:

```
dfs
```

8.4 Binding a shelf to a Recipe

CHAPTER

NINE

USING EXTENSIONS

Extensions build on the core behavior or recipe to let you perform

9.1 AutomaticFilters: Simple filtering

The AutomaticFilters extension provides a simpler approach to building filters using Ingredient.build_filter.

class recipe.**AutomaticFilters**(**args*, ***kwargs*)

Automatic generation and addition of Filters to a recipe.

Automatic filters take a dictionary of keys and values. For each key in the dictionary, if the key is the id of a `Dimension` on the shelf, a filter will be added to the recipe containing the values.

apply_automatic_filters(*value*)

Toggles whether automatic filters are applied to a recipe. The following will disable automatic filters for this recipe:

```
recipe.apply_automatic_filters(False)
```

automatic_filters(*value*)

Sets a dictionary of automatic filters to apply to this recipe. If your recipe uses a shelf that has dimensions 'state' and 'gender' you could filter the data to Men in California and New Hampshire with:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state': ['California', 'New Hampshire'],
    'gender': 'M'
})
```

Automatic filter keys can optionally include an `operator`.

List operators

If the value provided in the automatic_filter dictionary is a list, the following operators are available. The default operator is `in`:

```
in (default)
notin
quickselect (applies multiple conditions matching the
  named quickselect, quickselects are ORed together)
between (requires a list of two items)
```

Scalar operators

If the value provided in the automatic_filter dictionary is a scalar (a string, integer, or number), the following operators are available. The default operator is `eq`:

```
eq (equal) (the default)
ne (not equal)
lt (less than)
lte (less than or equal)
gt (greater than)
gte (greater than or equal)
like (SQL LIKE)
ilike (Case insensitive LIKE)
quickselect (applies the condition matching the named quickselect)
```

An example using operators

Here's an example that filters to states that start with the letters A-C:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state__lt': 'D'
})
```

Compound filters

If the key provided in the automatic_filter dictionary contains a comma, the filters will be treated as compound. Compound operators will be matched to the values by splitting the key on the commas then zipping the keys to values.

For instance, you could find newborns in California and 20 year olds in New Hampshire with:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'age': Dimension(Census.age),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state,age': [['California',0], ['New Hampshire',20]]
})
```

This would generate a SQL where clause that looked like:

```
WHERE
  (Census.state = 'California' and Census.age = 0) OR
  (Census.state = 'New Hampshire' and Census.age = 20)
```

Not all keys need to match in compound filters and may be provided. For instance, the following example uses operators and "unbalanced" keys:

```
recipe.dimensions('state').metrics('population').automatic_filters({
    'state,age__notin': [['California'], ['New Hampshire',[20,21,22,23]]]
})
```

This would generate a SQL where clause that looked like:

```
WHERE
  (Census.state = 'California') OR
  (Census.state = 'New Hampshire' and Census.age NOT IN (20,21,22,23))
```

Note: Using large numbers of compound filters is not efficient and may generate extremely large SQL.

exclude_automatic_filter_keys(**keys*)

A "blacklist" of automatic filter keys to exclude. The following will cause `'state'` to be ignored if it is present in the `automatic_filters` dictionary:

```
recipe.exclude_automatic_filter_keys('state')
```

include_automatic_filter_keys(**keys*)

A "whitelist" of automatic filter keys to use. The following will **only use** `'state'` for automatic filters regardless of what is provided in the automatic_filters dictionary:

```
recipe.include_automatic_filter_keys('state')
```

The AutomaticFilters extension.

9.2 SummarizeOver: Grouping over a dimenension

The SummarizeOver extension provides generates calculations over an aggregation of a Dimension.

class recipe.**SummarizeOver**(**args*, ***kwargs*)

9.3 CompareRecipe: Generating comparison values

The CompareRecipe extension lets you generate different Recipes againt the same table to generate comparison values.

class recipe.**CompareRecipe**(**args*, ***kwargs*)

Add compare recipes, used for presenting comparative context vis-a-vis a base recipe.

Supply a second recipe with the same `from`. Metrics from the second recipe will be hoisted to the base recipe and suffixed with a string (the default is "_compare" Dimensions will be used to match the base recipe to the compare recipe. Ordering from the base recipe is maintained.

compare(*compare_recipe*, *suffix='_compare'*)

Adds a comparison recipe to a base recipe.

9.4 BlendRecipe: Combining recipes from different tables

The BlendRecipe extension lets you combine data from multiple recipes.

class recipe.**BlendRecipe**(**args*, ***kwargs*)

Add blend recipes, used for joining data from another table to a base table

Supply a second recipe with a different `from` Optionally supply join criteria, if no join criteria is provided join will be attempted using constraints. All ingredients from the blended recipe will be hoisted to the base recipe except for ingredients that are used for joins (they must be the same anyway).

Supports blend (inner) and full_blend (outer) joins.

blend(*blend_recipe*, *join_base*, *join_blend*)

Blend a recipe into the base recipe. This performs an inner join of the blend_recipe to the base recipe's SQL.

full_blend(*blend_recipe*, *join_base*, *join_blend*)

Blend a recipe into the base recipe preserving values from both recipes.

This performs an outer join of the blend_recipe to the base recipe.

9.5 Anonymize: Realistic random data

The Anonymize extension lets generate anonymous data that resembles real data.

class recipe.**Anonymize**(**args*, ***kwargs*)

Allows recipes to be anonymized by adding an anonymize property. This flips the anonymize flag on all Ingredients used in the recipe.

Injects an ingredient.meta._anonymize boolean property on each used ingredient.

AnonymizeRecipe should occur last

add_ingredients()

Put the anonymizers in the last position of formatters

anonymize(*value*)

Should this recipe be anonymized

9.6 Paginate and PaginateInline: Returning data in pages

The Paginate and PaginateInline extensions lets recipes be paginated, searched and sorted.

class recipe.**Paginate**(**args*, ***kwargs*)

Allows recipes to paginate results. Pagination also supports searching and sorting within paginated data.

Using and controlling pagination

Pagination returns pages of data using limit and offset.

Pagination is enabled by setting a nonzero page size, like this:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf, extension_classes=[Paginate])            .dimensions(
↪'state')            .metrics('population')            .pagination_page_size(10)
```

Pagination may be disabled by setting *.apply_pagination(False)*.

Searching

pagination_q allows a recipe to be searched for a string. The default search fields are all dimensions used in the recipe. Search keys can be customized with *pagination_search_keys*. Search may be disabled by setting *.apply_pagination_filters(False)* The value role will be targetted when searching dimensions.

Sorting

Pagination can override ordering applied to a recipe by setting *.pagination_order_by(...)* to a list of ordering keys. If keys are preceded by a "-", ordering is descending, otherwise ordering is ascending.

An example using all features

Here's an example that searches for keys that start with "t", showing the fifth page of results:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'age': Dimension(Census.age),
    'population': Metric(func.sum(Census.population)),
})
recipe = self.recipe()            .metrics("pop2000")            .dimensions("state
↪", "sex", "age")            .pagination_page_size(10)            .pagination_
↪page(5)            .pagination_q('t%')            .pagination_search_keys("state",
↪ "sex")
```

This will generate SQL like:

```
SELECT census.age AS age,
       census.sex AS sex,
       census.state AS state,
       sum(census.population) AS population
FROM census
WHERE lower(census.state) LIKE lower('t%')
  OR lower(census.sex) LIKE lower('t%')
GROUP BY census.age,
         census.sex,
         census.state
LIMIT 10
OFFSET 40
```

apply_pagination(*value: bool*)

Should this recipe be paginated.

Parameters

value (*bool*) – Enable or disable pagination for this recipe, default True

apply_pagination_filters(*value: bool*)

Should this recipe apply the paginations query filtering.

Should paginate_q be used to apply a search on paginate_search_keys or all dimensions used in the recipe.

Parameters
value (*bool*) – Enable or disable pagination filtering for this recipe, default True

pagination_order_by(**value: Union[list, tuple]*)

Sort this pagination by these keys. Pagination ordering is applied before any other order_bys defined in the recipe.

Parameters
value (*list(str)*) – A list of keys to order the paginated recipe by

pagination_page(*value: int*)

Fetch this page.

Parameters
value (*integer*) – A positive integer page number to fetch

pagination_q(*value: str*)

Search this recipe for this string. The search is an case insensitive like that ORs all dimensions in the recipe by default.

To search for a substring, use a percentage sign for wildcard, like '%searchval%'.

pagination_search_keys can be used to customize what keys are used for search.

Parameters
value (*str*) – A query string to search for this in this recipe. The query string is evaluated as a *ilike* on all dimensions in the recipe or pagination_search_keys if provided

pagination_search_keys(**value: Union[list, tuple]*)

When querying this recipe with a *pagination_q*, search these keys

pagination_search_keys do not have to be used in the recipe.

Parameters
value (*list(str)*) – A list of keys to search in the paginated recipe

CHAPTER

TEN

SETTINGS

CHAPTER

ELEVEN

OVENS

Ovens are used to bake (execute) the queries generated by recipes. A standard oven is included in a recipe library, which provides connectivity to any database supported by SQLAlchemy. Remember, you'll need to have the required database driver installed. You can learn more in the SQLAlchemy documentation.

11.1 Initializing an Oven

To initialize an oven, you pass it the connection string for your database to the get_oven() function. You'll get back an oven that has a ready to use engine and session. For example, to connect to an in-memory sqlite database and use with a recipe.

```
from recipe import get_oven

oven = get_oven('sqlite://')
recipe = Recipe(session=oven.Session())
```

If you need to access the SQLAlchemy engine for any reason, it is available via the *engine* attribute.

11.2 Oven Drivers

Developers can also build custom oven drivers that provide advanced features. One example of that is the recipe_caching oven. You can pip install the recipe_caching python package, and you'll have access to an oven that caches the results of every query. If you want to use a custom oven driver, you pass the drivers name to the name keyword argument as shown here:

```
from recipe import get_oven

oven = get_oven('sqlite://', name='caching')
```

Note: Other ovens may require additional configuration or settings. So make sure to review their documentation.

You can learn more about creating your own drivers in the *Custom Oven Drivers* section.

CHAPTER

TWELVE

HOOKS

Recipes can call optional hooks to modify the recipe as it progresses towards execution. This is done by add the desired hooks names to the `dynamic_extensions` property of the recipe. Currently, no hooks are implemented in the base recipe library. However, much like ovens, they can be loaded via third party libraries.

For example, if we installed the `recipe_caching` library, we could add it's extension as shown here:

```
Recipe(shelf=shelf, session=oven.Session(), dynamic_extensions=['caching'])
```

You can learn more about creating your own in the *Dynamic Extensions* section.

CHAPTER

THIRTEEN

API

This part of the documentation covers all the interfaces of Recipe.

13.1 Recipe

class recipe.**Recipe**(*shelf=None*, *metrics=None*, *dimensions=None*, *filters=None*, *order_by=None*, *session=None*, *extension_classes=()*, *dynamic_extensions=None*)

A tool for getting data.

Parameters

- **shelf** (*Shelf*) – A shelf to use for shared metrics
- **metrics** (`list` of `str`) – A list of metrics to use from the shelf. These can also be *Metric* objects.
- **dimensions** (`list` of `str`) – A list of dimensions to use from the shelf. These can also be *Dimension* objects.
- **filters** (`list` of `str`) – A list of filters to use from the shelf. These can also be *Filter* objects.
- **order_by** (`list` of `str`) – A list of dimension or metric keys from the shelf to use for ordering. If prefixed by '-' the ordering will be descending.
- **session** (`Session`)
- **extension_classes** (`list` of *RecipeExtension*) – Extensions to apply to this recipe.
- **dynamic_extensions** (`list` of `str`) – Dynamic extensions to apply to this recipe.

Returns

A Recipe object.

all()

Return a (potentially cached) list of result objects.

as_table(*name=None*)

Return an alias to a table

cache_prefix(*value*) → *Recipe*

Set a cache prefix for recipe-caching to use

cache_region(*value*) → *Recipe*

Set a cache region for recipe-caching to use

dimensions(**dimensions*) → *Recipe*

Add a list of Dimension ingredients to the query. These can either be Dimension objects or strings representing dimensions on the shelf.

The Dimension expression will be added to the query's select statement and to the group_by.

Parameters
dimensions (*list*) – Dimensions to add to the recipe. Dimensions can either be keys on the `shelf` or Dimension objects

filters(**filters*) → *Recipe*

Add a list of Filter ingredients to the query. These can either be Filter objects or strings representing filters on the service's shelf. `.filters()` are additive, calling .filters() more than once will add to the list of filters being used by the recipe.

The Filter expression will be added to the query's where clause

Parameters
filters (*list*) – Filters to add to the recipe. Filters can either be keys on the `shelf` or Filter objects or binary expressions

first()

Return the first element on the result

classmethod from_config(*shelf*, *obj*, ***kwargs*)

Construct a Recipe from a plain Python dictionary.

Most of the directives only support named ingredients, specified as strings, and looked up on the shelf. But filters can be specified as objects.

Additionally, each RecipeExtension can extract and handle data from the configuration.

limit(*limit*) → *Recipe*

Limit the number of rows returned from the database.

Parameters
limit (*int*) – The number of rows to return in the recipe. 0 will return all rows.

metrics(**metrics*) → *Recipe*

Add a list of Metric ingredients to the query. These can either be Metric objects or strings representing metrics on the shelf.

The Metric expression will be added to the query's select statement. The metric value is a property of each row of the result.

Parameters
metrics (*list*) – Metrics to add to the recipe. Metrics can either be keys on the `shelf` or Metric objects

offset(*offset*) → *Recipe*

Offset a number of rows before returning rows from the database.

Parameters
offset (*int*) – The number of rows to offset in the recipe. 0 will return from the first available row

one()

Return the first element on the result

order_by(**order_bys*) → *Recipe*

Apply an ordering to the recipe results.

Parameters
order_bys (*list(str)*) – Order_bys to add to the recipe. Order_bys must be keys of ingredients already added to the recipe. If the key is prefixed by "-" the ordering will be descending.

query()

Generates a query using the ingredients supplied by the recipe.

Returns
A SQLAlchemy query

shelf(*shelf=None*) → *Recipe*

Defines a shelf to use for this recipe

subquery(*name=None*)

The recipe's query as a subquery suitable for use in joins or other queries.

to_sql()

A string representation of the SQL this recipe will generate.

total_count(*query=None*)

Return the number of rows that would be returned by this Recipe, ignoring any *limit* that has been applied.

Parameters
query – An optional SQLAlchemy query to calculate total_count for. If None, the recipe query will be used. If a query is passed, no caching will be done.

Returns
A count of the number of rows that are returned by this query.

use_cache(*value*) → *Recipe*

If False, invalidate the cache before fetching data.

13.2 Shelf

class recipe.**Shelf**(**args*, ***kwargs*)

Holds ingredients used by a recipe.

Can be initialized with no arguments, but also accepts: - a dictionary of ingredients as a positional argument - ingredients as keyword arguments

These keyword arguments have special meaning:

Parameters

- **select_from** – The SQLALchemy-compatible object which will be queried (usually a Table or ORM object).
- **table** – Unused, but stored on the *Meta* attribute.
- **metadata** – Unused, but stored on the *Meta* attribute.

brew_query_parts(*order_by_keys=[]*)

Make columns, group_bys, filters, havings

property dimension_ids

Return the Dimensions on this shelf in the order in which they were used.

enchant(*data*, *cache_context=None*)

Add any calculated values to each row of a resultset generating a new namedtuple

Parameters

- **data** – a list of row results
- **cache_context** – optional extra context for caching

Returns
a list with ingredient.cauldron_extras added for all ingredients

property filter_ids

Return the Filters on this shelf in the order in which they were used.

find(*obj*, *filter_to_class=<class 'recipe.ingredients.Ingredient'>*, *constructor=None*)

Find an Ingredient, optionally using the shelf.

Parameters

- **obj** – A string or Ingredient
- **filter_to_class** – The Ingredient subclass that obj must be an instance of
- **constructor** – An optional callable for building Ingredients from obj

Returns
An Ingredient of subclass *filter_to_class*

classmethod from_config(*obj*, *selectable*, *ingredient_constructor=<function ingredient_from_validated_dict>*, *metadata=None*)

Create a shelf using a dict shelf definition.

Parameters

- **obj** – A Python dictionary describing a Shelf.
- **selectable** – A SQLAlchemy Table, a Recipe, a table name, or a SQLAlchemy join to select from.
- **metadata** – If *selectable* is passed as a table name, then in order to introspect its schema, we must have the SQLAlchemy MetaData object to associate it with.

Returns
A shelf that contains the ingredients defined in obj.

classmethod from_validated_yaml(*yaml_str*, *selectable*, ***kwargs*)

Create a shelf using a yaml shelf definition.

Parameters

- **yaml_str** – A string containing yaml ingredient definitions.
- **selectable** – A SQLAlchemy Table, a Recipe, or a SQLAlchemy join to select from.

Returns
A shelf that contains the ingredients defined in yaml_str.

classmethod from_yaml(*yaml_str*, *selectable*, ***kwargs*)

Shim that calls from_validated_yaml.

This used to call a different implementation of yaml parsing

ingredients()

Return the ingredients in this shelf in a deterministic order

items()

Return an iterator over the ingredient names and values.

keys()

Return an iterator over the ingredient keys.

property metric_ids

Return the Metrics on this shelf in the order in which they were used.

pop(*k*, *d=<object object>*)

Pop an ingredient off of this shelf.

values()

Return an iterator over the ingredients.

13.3 Ingredients

class recipe.**Ingredient**(***kwargs*)

Ingredients combine to make a SQLAlchemy query.

Any unknown keyword arguments provided to an Ingredient during initialization are stored in a meta object.

```
# icon is an unknown keyword argument
m = Metric(func.sum(MyTable.sales), icon='cog')
print(m.meta.icon)
>>> 'cog'
```

This meta storage can be used to add new capabilities to ingredients.

Parameters

- **id** (`str`) – An id to identify this Ingredient. If ingredients are added to a Shelf, the id is automatically set as the key in the shelf.
- **columns** (`list` of `ColumnElement`) – A list of SQLAlchemy columns to use in a query select.
- **filters** (`list` of `BinaryExpression`) – A list of SQLAlchemy BinaryExpressions to use in the .filter() clause of a query.
- **havings** (`list` of `BinaryExpression`) – A list of SQLAlchemy BinaryExpressions to use in the .having() clause of a query.
- **columns** (`list` of `ColumnElement`) – A list of SQLAlchemy columns to use in the *group_by* clause of a query.
- **formatters** – (`list` of `callable`): A list of callables to apply to the result values. If formatters exist, property *{ingredient.id}_raw* will exist on each result row containing the unformatted value.
- **cache_context** (`str`) – Extra context when caching this ingredient. DEPRECATED
- **ordering** (*string*, 'asc' or 'desc') – One of 'asc' or 'desc'. 'asc' is the default value. The default ordering of this ingredient if it is used in a `recipe.order_by`. This is added to the ingredient when the ingredient is used in a `recipe.order_by`.

- **group_by_strategy** (`str`) – A strategy to use when preparing group_bys for the query "labels" is the default strategy which will use the labels assigned to each column. "direct" will use the column expression directly. This alternative is useful when there might be more than one column with the same label being used in the query.
- **quickselects** (`list` of named filters) – A list of named filters that can be accessed through `build_filter`. Named filters are dictionaries with a `name` (:obj:str) property and a `condition` property (`BinaryExpression`)
- **datatype** (`str`) – The identified datatype (num, str, date, bool, datetime) of the parsed expression
- **datatype_by_role** (`dict`) – The identified datatype (num, str, date, bool, datetime) for each role.

Returns
An Ingredient object.

`build_filter`(*value*, *operator=None*, *target_role=None*)

Builds a filter based on a supplied value and optional operator. If no operator is supplied an `in` filter will be used for a list and a `eq` filter if we get a scalar value.

`build_filter` is used by the AutomaticFilter extension.

Parameters

- **value** – A value or list of values to operate against
- **operator** (`str`) – An operator that determines the type of comparison to do against value.

  The default operator is 'in' if value is a list and 'eq' if value is a string, number, boolean or None.
- **target_role** (*str*) – An optional role to build the filter against

Returns
A SQLAlchemy boolean expression

`property cauldron_extras`

Yield extra tuples containing a field name and a callable that takes a row.

`describe`()

A string representation of the ingredient.

`property expression`

An accessor for the SQLAlchemy expression representing this Ingredient.

`make_column_suffixes`()

Make sure we have the right column suffixes. These will be appended to *id* when generating the query.

Developers note: These are generated when the query runs because the recipe may be run with anonymization on or off, which will inject a formatter.

`property order_by_columns`

Yield columns to be used in an order by using this ingredient. Column ordering is in reverse order of columns. When grouping, recipe supports two strategies. group_by_strategy == "labels" uses the labels added to columns. This is preferable and is supported by some databases. SQL Server requires grouping by the original column expressions

`property query_columns`

Yield labeled columns to be used as a select in a query.

class recipe.**Dimension**(*expression*, ***kwargs*)

A Dimension is an Ingredient that adds columns and groups by those columns. Columns should be non-aggregate SQLAlchemy expressions.

The required expression supplies the dimension's "value" role. Additional expressions can be provided in keyword arguments with keys that look like "{role}_expression". The role is suffixed to the end of the SQL column name.

For instance, the following

```
Dimension(Hospitals.name,
          latitude_expression=Hospitals.lat
          longitude_expression=Hospitals.lng,
          id='hospital')
```

would add columns named "hospital", "hospital_latitude", and "hospital_longitude" to the recipes results. All three of these expressions would be used as group bys.

Two special roles that can be added are "id" and "order_by". If a keyword argument "id_expression" is passed, this expression will appear first in the list of columns and group_bys. This "id" will be used if you call *build_filter* on the dimension.

If the keyword argument "order_by_expression" is passed, this expression will appear last in the list of columns and group_bys.

The following additional keyword parameters are also supported:

Parameters

- **lookup** (`dict`) – A dictionary that is used to map values to new values.

  Note: Lookup adds a `formatter` callable as the first item in the list of formatters.

- **lookup_default** (`object`) – A default to show if the value can't be found in the lookup dictionary.

Returns
A Filter object

Parameters

- **lookup** – dict A dictionary to translate values into
- **lookup_default** – A default to show if the value can't be found in the lookup dictionary.

property cauldron_extras

Yield extra tuples containing a field name and a callable that takes a row

property id_prop

The label of this dimensions id in the query columns

make_column_suffixes()

Make sure we have the right column suffixes. These will be appended to *id* when generating the query.

class recipe.**IdValueDimension**(*id_expression*, *value_expression*, ***kwargs*)

DEPRECATED: A convenience class for creating a Dimension with a separate `id_expression`. The following are identical.

```
d = Dimension(Student.student_name, id_expression=Student.student_id)

d = IdValueDimension(Student.student_id, Student.student_name)
```

The former approach is recommended.

Parameters

- **id_expression** (`ColumnElement`) – A column expression that is used to identify the id for a Dimension
- **value_expression** (`ColumnElement`) – A column expression that is used to identify the value for a Dimension

`class recipe.Metric(expression, **kwargs)`

A simple metric created from a single expression

`build_filter(value, operator=None)`

Building filters with Metric returns Having objects.

`class recipe.WtdAvgMetric(expression, weight_expression, **kwargs)`

A metric that generates the weighted average of a metric by a weight.

`class recipe.DivideMetric(numerator, denominator, **kwargs)`

A metric that divides a numerator by a denominator handling several possible error conditions

The default strategy is to add an small value to the denominator Passing ifzero allows you to give a different value if the denominator is zero.

`class recipe.Filter(expression, **kwargs)`

A simple filter created from a single expression.

`property expression`

An accessor for the SQLAlchemy expression representing this Ingredient.

`class recipe.Having(expression, **kwargs)`

A Having that limits results based on an aggregate boolean clause

`property expression`

An accessor for the SQLAlchemy expression representing this Ingredient.

13.4 Extensions

`class recipe.RecipeExtension(recipe)`

Recipe extensions plug into the recipe builder pattern and can modify the generated query.

recipe generates a query in the following way

(RECIPE) recipe checks if a query has been generated

(EXTENSIONS) all extension `add_ingredients` run to inject ingredients directly on the recipe

(RECIPE) recipe runs gather_all_ingredients_into_cauldron to build a global lookup for ingredients

(RECIPE) recipe runs cauldron.brew_query_parts to gather sqlalchemy columns, group_bys and filters

(EXTENSIONS) all extension `modify_recipe_parts(recipeparts)` run to directly modify the collected sqlalchemy columns, group_bys or filters

(RECIPE) recipe builds a preliminary query with columns

(EXTENSIONS) all extension `modify_prequery_parts(prequery_parts)` run to modify the query

(RECIPE) recipe builds a full query with group_bys, order_bys, and filters.

(RECIPE) recipe tests that this query only uses a single from

(EXTENSIONS) all extension `modify_postquery_parts( postquery_parts)` run to modify the query

(RECIPE) recipe applies limits and offsets on the query

(RECIPE) recipe caches completed query

When the recipe fetches data the results will be `enchanted` to add fields to the result. `RecipeExtensions` can modify result rows with

enchant_add_fields: Return a tuple of field names to add to a result row

enchant_row(row): Return a tuple of field values for each row in results.

add_ingredients()

Add ingredients to the recipe

This method should be overridden by subclasses

enchant_add_fields() → tuple

This method allows extensions to add fields to a result row. Return a tuple of the field names that are being added with this method

enchant_row(*row*)

This method adds the fields named in `enchant_add_fields` to each result row.

modify_postquery_parts(*postquery_parts: dict*) → dict

This method allows extensions to directly modify query, group_bys, filters, and order_bys generated from collected ingredients after a final query using columns has been created.

modify_prequery_parts(*prequery_parts: dict*) → dict

This method allows extensions to directly modify query, group_bys, filters, and order_bys generated from collected ingredients after a preliminary query using columns has been created.

modify_recipe_parts(*recipe_parts: dict*) → dict

Modify sqlalchemy components of the query

This method allows extensions to directly modify columns, group_bys, filters, and order_bys generated from collected ingredients.

class recipe.**AutomaticFilters**(**args*, ***kwargs*)

Automatic generation and addition of Filters to a recipe.

Automatic filters take a dictionary of keys and values. For each key in the dictionary, if the key is the id of a `Dimension` on the shelf, a filter will be added to the recipe containing the values.

apply_automatic_filters(*value*)

Toggles whether automatic filters are applied to a recipe. The following will disable automatic filters for this recipe:

```
recipe.apply_automatic_filters(False)
```

automatic_filters(*value*)

Sets a dictionary of automatic filters to apply to this recipe. If your recipe uses a shelf that has dimensions 'state' and 'gender' you could filter the data to Men in California and New Hampshire with:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state': ['California', 'New Hampshire'],
    'gender': 'M'
})
```

Automatic filter keys can optionally include an `operator`.

List operators

If the value provided in the automatic_filter dictionary is a list, the following operators are available. The default operator is `in`:

```
in (default)
notin
quickselect (applies multiple conditions matching the
  named quickselect, quickselects are ORed together)
between (requires a list of two items)
```

Scalar operators

If the value provided in the automatic_filter dictionary is a scalar (a string, integer, or number), the following operators are available. The default operator is `eq`:

```
eq (equal) (the default)
ne (not equal)
lt (less than)
lte (less than or equal)
gt (greater than)
gte (greater than or equal)
like (SQL LIKE)
ilike (Case insensitive LIKE)
quickselect (applies the condition matching the named quickselect)
```

An example using operators

Here's an example that filters to states that start with the letters A-C:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state__lt': 'D'
})
```

Compound filters

If the key provided in the automatic_filter dictionary contains a comma, the filters will be treated as compound. Compound operators will be matched to the values by splitting the key on the commas then zipping

the keys to values.

For instance, you could find newborns in California and 20 year olds in New Hampshire with:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'age': Dimension(Census.age),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf)
recipe.dimensions('state').metrics('population').automatic_filters({
    'state,age': [['California',0], ['New Hampshire',20]]
})
```

This would generate a SQL where clause that looked like:

```
WHERE
  (Census.state = 'California' and Census.age = 0) OR
  (Census.state = 'New Hampshire' and Census.age = 20)
```

Not all keys need to match in compound filters and may be provided. For instance, the following example uses operators and "unbalanced" keys:

```
recipe.dimensions('state').metrics('population').automatic_filters({
    'state,age__notin': [['California'], ['New Hampshire',[20,21,22,23]]]
})
```

This would generate a SQL where clause that looked like:

```
WHERE
  (Census.state = 'California') OR
  (Census.state = 'New Hampshire' and Census.age NOT IN (20,21,22,23))
```

Note: Using large numbers of compound filters is not efficient and may generate extremely large SQL.

exclude_automatic_filter_keys(**keys*)

A "blacklist" of automatic filter keys to exclude. The following will cause `'state'` to be ignored if it is present in the `automatic_filters` dictionary:

```
recipe.exclude_automatic_filter_keys('state')
```

include_automatic_filter_keys(**keys*)

A "whitelist" of automatic filter keys to use. The following will **only use** `'state'` for automatic filters regardless of what is provided in the automatic_filters dictionary:

```
recipe.include_automatic_filter_keys('state')
```

class recipe.**BlendRecipe**(**args*, ***kwargs*)

Add blend recipes, used for joining data from another table to a base table

Supply a second recipe with a different `from` Optionally supply join criteria, if no join criteria is provided join will be attempted using constraints. All ingredients from the blended recipe will be hoisted to the base recipe except for ingredients that are used for joins (they must be the same anyway).

Supports blend (inner) and full_blend (outer) joins.

blend(*blend_recipe*, *join_base*, *join_blend*)

Blend a recipe into the base recipe. This performs an inner join of the blend_recipe to the base recipe's SQL.

full_blend(*blend_recipe*, *join_base*, *join_blend*)

Blend a recipe into the base recipe preserving values from both recipes.

This performs an outer join of the blend_recipe to the base recipe.

class recipe.**CompareRecipe**(**args*, ***kwargs*)

Add compare recipes, used for presenting comparative context vis-a-vis a base recipe.

Supply a second recipe with the same `from`. Metrics from the second recipe will be hoisted to the base recipe and suffixed with a string (the default is "_compare" Dimensions will be used to match the base recipe to the compare recipe. Ordering from the base recipe is maintained.

compare(*compare_recipe*, *suffix='_compare'*)

Adds a comparison recipe to a base recipe.

class recipe.**SummarizeOver**(**args*, ***kwargs*)

class recipe.**Anonymize**(**args*, ***kwargs*)

Allows recipes to be anonymized by adding an anonymize property. This flips the anonymize flag on all Ingredients used in the recipe.

Injects an ingredient.meta._anonymize boolean property on each used ingredient.

AnonymizeRecipe should occur last

add_ingredients()

Put the anonymizers in the last position of formatters

anonymize(*value*)

Should this recipe be anonymized

class recipe.**Paginate**(**args*, ***kwargs*)

Allows recipes to paginate results. Pagination also supports searching and sorting within paginated data.

Using and controlling pagination

Pagination returns pages of data using limit and offset.

Pagination is enabled by setting a nonzero page size, like this:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'population': Metric(func.sum(Census.population)),
})
recipe = Recipe(shelf=shelf, extension_classes=[Paginate])            .dimensions(
↪'state')            .metrics('population')            .pagination_page_size(10)
```

Pagination may be disabled by setting *.apply_pagination(False)*.

Searching

pagination_q allows a recipe to be searched for a string. The default search fields are all dimensions used in the recipe. Search keys can be customized with *pagination_search_keys*. Search may be disabled by setting *.apply_pagination_filters(False)* The value role will be targetted when searching dimensions.

Sorting

Pagination can override ordering applied to a recipe by setting *.pagination_order_by(...)* to a list of ordering keys. If keys are preceded by a "-", ordering is descending, otherwise ordering is ascending.

An example using all features

Here's an example that searches for keys that start with "t", showing the fifth page of results:

```
shelf = Shelf({
    'state': Dimension(Census.state),
    'gender': Dimension(Census.gender),
    'age': Dimension(Census.age),
    'population': Metric(func.sum(Census.population)),
})
recipe = self.recipe()            .metrics("pop2000")            .dimensions("state
↪", "sex", "age")            .pagination_page_size(10)            .pagination_
↪page(5)            .pagination_q('t%')            .pagination_search_keys("state",
↪ "sex")
```

This will generate SQL like:

```
SELECT census.age AS age,
       census.sex AS sex,
       census.state AS state,
       sum(census.population) AS population
FROM census
WHERE lower(census.state) LIKE lower('t%')
  OR lower(census.sex) LIKE lower('t%')
GROUP BY census.age,
         census.sex,
         census.state
LIMIT 10
OFFSET 40
```

add_ingredients()

Apply pagination ordering and search to this query if necessary.

apply_pagination(*value: bool*)

Should this recipe be paginated.

Parameters
value (*bool*) – Enable or disable pagination for this recipe, default True

apply_pagination_filters(*value: bool*)

Should this recipe apply the paginations query filtering.

Should paginate_q be used to apply a search on paginate_search_keys or all dimensions used in the recipe.

Parameters
value (*bool*) – Enable or disable pagination filtering for this recipe, default True

do_pagination()

Should pagination be added to this recipe.

modify_postquery_parts(*postquery_parts*)

Apply validated pagination limits and offset to a completed query.

pagination_default_order_by(**value: Union[list, tuple]*)

Paginated queries must be ordered. This ordering is applied if the recipe has no order_by and no pagination_order_by has been set.

Parameters
value (*list(str)*) – A list of keys to order the paginated recipe by if not other ordering is applied.

pagination_order_by(**value: Union[list, tuple]*)

Sort this pagination by these keys. Pagination ordering is applied before any other order_bys defined in the recipe.

Parameters
value (*list(str)*) – A list of keys to order the paginated recipe by

pagination_page(*value: int*)

Fetch this page.

Parameters
value (*integer*) – A positive integer page number to fetch

pagination_page_size(*value: int*)

Paginate recipe responses into pages of this size.

A page size of zero disables pagination.

Parameters
value (*integer*) – A page size (zero or a positive integer)

pagination_q(*value: str*)

Search this recipe for this string. The search is an case insensitive like that ORs all dimensions in the recipe by default.

To search for a substring, use a percentage sign for wildcard, like '%searchval%'.

pagination_search_keys can be used to customize what keys are used for search.

Parameters
value (*str*) – A query string to search for this in this recipe. The query string is evaluated as a *ilike* on all dimensions in the recipe or pagination_search_keys if provided

pagination_search_keys(**value: Union[list, tuple]*)

When querying this recipe with a *pagination_q*, search these keys

pagination_search_keys do not have to be used in the recipe.

Parameters
value (*list(str)*) – A list of keys to search in the paginated recipe

validated_pagination()

Return pagination validated against the actual number of items in the response. Returns None if the recipe has not run.

13.5 Exceptions

exception recipe.**BadIngredient**

Something is wrong with an ingredient

exception recipe.**BadRecipe**

Something is wrong with a recipe

Now, go start some *Recipe Development*.

CHAPTER

FOURTEEN

DEVELOPMENT

Recipe is under active development, and contributors are welcome.

If you have a feature request, suggestion, or bug report, please open a new issue on GitHub. To submit patches, please send a pull request on GitHub.

14.1 Conventions

Recipe code wraps at 79 characters and passes flake8. Strings should single quoted unless double quoting leads to less escaping. Add tests to achieve 100% code coverage.

14.2 Source Control

The project is hosted on at https://github.com/juiceinc/recipe

The repository is publicly accessible. To check it out, run:

```
git clone git://github.com/juiceinc/recipe.git
```

14.2.1 Git Branch Structure

develop
: The "next release" branch. Likely unstable.

master
: Current production release (0.31.5) on PyPi.

Each release is tagged.

When submitting patches, please place your feature/change in its own branch prior to opening a pull request on GitHub.

14.3 Adding New Extensions

Recipe welcomes new extensions.

Extensions subclass RecipeExtension and plug into the base recipe's `.query()` method which builds a SQLAlchemy query. Extensions can either modify the base recipe like these do.

- AutomaticFilters
- Anonymize
- SummarizeOver

Or extensions can merge one or more recipes into the base recipe. Extensions that require another recipe should have a classname that ends with **Recipe**.

- CompareRecipe
- BlendRecipe

When adding an extension, do the following.

1) Add extension to src/extensions.py
2) Add tests to tests/test_extensions.py, cover 100% of extension function and test that the extension doesn't interfere with other extensions
3) Make sure your extension code passes flake8
4) Add extension description to docs/extensions/
5) Submit a PR!

14.4 Adding New Ingredients

Recipe welcomes new ingredients, particularly metrics and dimensions that cover common patterns of data aggregation.

Subclass the appropriate ingredient and don't duplicate something that a superclass does. For instance `WtdAvgMetric` is a subclass of `DivideMetric` that generates it's expressions differently.

Extra functionality can be added by using Ingredient.meta in structured ways.

A checklist of adding an extension.

1) Add extension to src/ingredients.py
2) Add tests to tests/test_ingredients.py, cover 100% of ingredient parameters.
3) Make sure your ingredient passes flake8
4) Submit a PR!

14.5 Testing Recipe

Testing is crucial to confident development and stability. This stable project is used in production by many companies and developers, so it is important to be certain that every version released is fully operational. When developing a new feature for Recipe, be sure to write proper tests for it as well.

When developing a feature for Recipe, the easiest way to test your changes for potential issues is to simply run the test suite directly.

```
$ make tests
```

This will run tests under pytest and show code coverage data.

14.6 Continuous Integration

Every commit made to the **develop** branch is automatically tested and inspected upon receipt with **`Travis CI`_**. If you have access to the main repository and broke the build, you will receive an email accordingly.

Anyone may view the build status and history at any time.

> https://travis-ci.org/juiceinc/tablib

Additional reports will also be included here in the future, including PEP 8
checks and stress reports for extremely large datasets.

14.7 Building the Docs

Documentation in reStructured Text and powered by Sphinx.

The Docs live in `recipe/docs`. In order to build them, you will first need
to install Sphinx.

```
$ pip install sphinx
```

To build an HTML version of the docs, simply run the following from the **docs** directory:

```
$ make html
```

Your `docs/_build/html` directory will then contain the fully build documentation, ready for publishing. You can also generate the documentation in tons of other formats.

If you want to learn more, check out the *API Documentation*.

CHAPTER

FIFTEEN

CUSTOM OVEN DRIVERS

It's possible to implement your own custom oven drivers to get a desired behavior for the engine or the session. An abstract base class is provided for you to inherit from called `OvenBase`. You need to implement an `init_engine` that returns a SQLAlchemy engine, and an `init_session` that returns a SQLAlchemy sessionmaker. The default `__init__` method sets the output of both of these to to the oven's `engine` and `Session` properties respectively.

Note: Remember to use recipe's built in settings to handle any configuration options/settings you made need for your driver.

15.1 OvenBase

class recipe.oven.base.**OvenBase**(*connection_string=None*)

Base class for ovens

abstract init_engine(*connection_string=None*, ***kwargs*)

Initializes a SQLAlchemy Engine for a given connection string with all other keyword arguments passed to the create_engine function. The connection uses pre-ping to verify connections.

Parameters

- **self** (*Oven*) – a reference to ourselves
- **connection_string** (*str*) – a reference to ourselves
- **kwargs** (*dict*) – a collection of arguments passed to the engine

Returns
A SQLAlchemy Engine with connection checking

Return type
SQLAlchemy.Engine

abstract init_session()

Initializes a SQLAlchemy Session with the Oven's engine

Parameters
self (*Oven*) – a reference to ourselves

Returns
A SQLAlchemy Session using self.engine

Return type
SQLAlchemy.Session

CHAPTER

SIXTEEN

DYNAMIC EXTENSIONS

Recipes can to load dynamic plugins and extensions as hooks. The hooks are expected to accept a recipe_parts dict or object and have an execute method that returns a new recipe_parts dict or object. The plugins must be in the appropiate namespace depending on where they get called. The *recipe.hooks.modify_query* namespace is one of the namespaces that is available. You can see the `recipe_caching` library for a concrete implementation.

Note: Remember to use recipe's built in settings to handle any configuration options/settings you made need for your extension.

16.1 DynamicExtensionBase

class recipe.dynamic_extensions.**DynamicExtensionBase**(*recipe_parts*, *hook_type='modify_query'*)

Base class for dynamic extensions

abstract execute()

Perform transformations on recipe_parts here

CHAPTER

SEVENTEEN

CHANGELOG

17.1 v0.31.5 (2022-06-13)

- Fix timestamp conversion functions in bigquery

17.2 v0.31.4 (2022-04-04)

- Support *and* operator in complex filters

17.3 v0.31.3 (2022-04-04)

- no changes

17.4 v0.31.2 (2022-03-25)

- Disallow literal-only expressions
- Allow count for boolean expressions

17.5 v0.31.1 (2022-03-24)

- Add caching for total_count

17.6 v0.31.0 (2022-03-23)

- Allow nested operators and values within an in operator
- **"notin" filter operator is refactored to not use separate code from in. Instead we generate the in code and then**
  wrap it in _not. This will change the sql generated when automatic filtering but the results will be the same.
- Code cleanups and refactorings

17.7 v0.30.1 (2022-03-22)

- Fix an error in ordering with mixed case columns/labels when using snowflake
- Update requirements to use lark
- Update requirements for dateparser past a broken version (See issue https://github.com/scrapinghub/dateparser/issues/1045)
- Don't create expression grammar for columns with invalid names

17.8 v0.30.0 (2022-02-15)

- Breaking chagne: removed support for v1 ingredient configuration.
- Refactor tests to use unittests
- Add type annotations
- Add substr function

17.9 v0.29.3 (2021-12-07)

- Add support for like and ilike in parsed expressions

17.10 v0.29.1 (2021-12-03)

- Fix automatic filters when dimension ids contain double underscores

17.11 v0.29.0 (2021-11-17)

- Improve mssql support

17.12 v0.28.1 (2021-10-28)

- Fix for splitting operators in automatic filters

17.13 v0.28.0 (2021-10-15)

- Add directives that will convert dates and datetimes to the nearest year/month/day

17.14 v0.27.1 (2021-09-14))

- Allow compound selection to take a list of json encoded strings

17.15 v0.27.0 (2021-08-26)

- Update requirements
- Drop support for python3.6
- Save metric and dimension keys without deduping

17.16 v0.26.1 (2021-07-29)

- Fix aggregation for PaginateInline extension

17.17 v0.26.0 (2021-07-15)

- Add PaginateInline extension

17.18 v0.25.1 (2021-06-15)

- Fix datatype tracking in some cases

17.19 v0.25.0 (2021-06-07)

- Add to date syntax
- Avoid installing a top-level tests package in setup.py

17.20 v0.24.1 (2021-06-10)

- Fix datatype tracking in some cases

17.21 v0.24.0 (2021-05-14)

- Track the datatype used by ingredient columns
- Require parsed metrics to generate a number

17.22 v0.23.4 (2021-05-03)

- Improve automatic filtering with uncompilable ingredients

17.23 v0.23.3 (2021-04-29)

- Fix column_type for timestamps

17.24 v0.23.2 (2021-02-09)

- Apply a default ordering when paginating

17.25 v0.23.1 (2021-02-08)

- Fix sql generation of timestamp truncated columns in bigquery

17.26 v0.23.0 (2021-02-01)

- Improve the lark parser to validate explicitly using the database columns and column types available in the data.
- Run a validation phase on a parsed tree to make sure that arguments are correct types.
- Return descriptive errors
- Improve cross database support

17.27 v0.22.1 (2020-12-23)

- Like and ilike filter generation is more lenient

17.28 v0.22.0 (2020-12-10)

- Drop python2 support

17.29 v0.21.0 (2020-10-20)

- Add [syntax] to disambiguate database columns in parsed fields
- Save original config to ingredient when generating parsed fields.

17.30 v0.20.1 (2020-10-07)

- Fix issue with parsing >= and <=

17.31 v0.20.0 (2020-10-02)

- Update total_count to use caching
- Fix datatime auto conversions

17.32 0.19.1 (2020-09-10)

- Drop python2.7 testing support (Python2.7 support will be dropped in 0.20)
- Improve type identification in Ingredient.build_filter

17.33 0.19.0 (2020-09-04)

- Support and documentation for compound selection in automatic filters
- Support for different sqlalchemy generation when using parsed fields
- Add support for date conversions and percentiles in bigquery.
- Ingredient.build_filters now returns SQLAlchemy BinaryExpression rather than Filter objects.

17.34 0.18.1 (2020-08-07)

- Fix a bug in filter binning
- Happy birthday, Zoe!

17.35 0.18.0 (2020-07-31)

- Add automatic filter binning for redshift to reduce required query compilations
- Add parsed field converters to perform casting and date truncation.

17.36 0.17.2 (2020-07-21)

- Fix Paginate search to use value roles

17.37 0.17.1 (2020-07-09)

- Fix parsed syntax for *field IS NULL*

17.38 0.17.0 (2020-06-26)

- Set bucket default label to "Not found"
- Use sureberus to validate lookup is a dictionary if present in Dimension config
- Fix to ensure pagination page is 1 even if there is no data
- On shelf construction, create InvalidIngredient for ingredients that fail construction

17.39 0.16.0 (2020-06-19)

- Ignore order_by on a recipe if the ingredient has not been added to the dimensions or metrics.
- Allows case insensitivity in "kind:" and support "kind: Measure" as an alternative to "kind: Metric"
- Fix like/ilike and pagination_q filtering against dimensions that have a non-string ID.
- Fix parsed sql generation for AND and OR
- Fix parsed sql generation for division when one of the terms is a constant (like sum(people) / 100.0)
- Adds IS NULL as a boolean expression
- Adds "Intelligent date" calculations to allow more useful date calculations relative to current date

17.40 0.15.0 (2020-05-08)

- Ignore order_by if ingredients have not been added
- Support measure as a synonym for metric and be lenient about capitalization in shelf config

17.41 0.14.0 (2020-03-06)

- Support graceful ingredient failures when ingredients can not be constructed from config.

17.42 0.13.1 (2020-02-11)

- Fix a pg8000 issue

17.43 0.13.0 (2020-01-28)

- Extend grouping strategies so recipes can also order by column labels
- Create a new shelf configuration that uses lark to parse text into SQLAlchemy.

17.44 0.12.0 (2019-11-25)

- remove flapjack_stack and pyhash dependencies
- Add percentile aggregations to metrics from config.
- Use more accurate fetched_from_cache caching query attribute
- Add grouping strategies so recipes can group by column labels

17.45 0.11.0 (2019-11-07)

- Add Paginate extension
- Fix deterministic Anonymization in python3
- CI improvements

17.46 0.10.0 (2019-08-07)

- Support multiple quickselects which are ORed together

17.47 0.9.0 (2019-08-07)

- Replace quickfilter with quickselect
- Improve and publish docs on at recipe.readthedocs.io
- Happy birthday, Zoe!

17.48 0.8.0 (2019-07-08)

- Add cache control options.

17.49 0.7.0 (2019-06-24)

- Support date ranges in configuration defined ingredients
- Add like, ilike, between in ingredients defined from config
- Better handling in automatic filters when Nones appear in lists
- Remove dirty flag
- Ingredients defined from config support safe division by default
- [ISSUE-37] Allow Dimension defined from config to be defined using buckets

17.50 0.6.2 (2019-06-11)

17.51 0.1.0 (2017-02-05)

- First release on PyPI.

PYTHON MODULE INDEX

INDEX