agate Documentation

Release 0.11.0 (alpha)

Christopher Groskopf

October 06, 2015

Contents

CHAPTER 1

About

agate is a Python data analysis library designed for humans working in the real world. It is an alternative to numpy and pandas that is optimized for making humans faster at working with normal-sized datasets.

agate was previously known as journalism.

Important links:

- Documentation: http://agate.rtfd.org
- Repository: https://github.com/onyxfish/agate
- Issues: https://github.com/onyxfish/agate/issues

CHAPTER 2

Why agate?

Why use agate?

- A clean, readable API.
- Self-documenting code patterns.
- A full set of SQL-like operations.
- Unicode support everywhere.
- Decimal precision everywhere.
- Pure Python. It runs everywhere.
- 100% test coverage.
- Exhaustive user documentation.
- Optional extensions that add support for SQL integration and more.

CHAPTER 3

Table of contents

3.1 Installation

3.1.1 Users

If you only want to use agate, install it this way:

```
pip install agate
```

Note: Need more speed? If you're running Python 2.6, 2.7 or 3.2, you can `pip install cdecimal` for a significant speed boost. This isn't installed automatically because it can create additional complications.

3.1.2 Developers

If you are a developer that also wants to hack on agate, install it this way:

```
git clone git://github.com/onyxfish/agate.git
cd agate
mkvirtualenv agate
pip install -r requirements.txt
python setup.py develop
tox
```

Note: agate also supports running tests with coverage:

```
nosetests --with-coverage --cover-package=agate
```

3.1.3 Supported platforms

agate supports the following versions of Python:

- Python 2.6 (tests pass, but some dependencies claim not to support it)
- Python 2.7
- Python 3.3
- Python 3.4
- Latest PyPy

It is tested on OSX, but due to it's minimal dependencies should work fine on both Linux and Windows.

3.2 Tutorial

3.2.1 About this tutorial

The best way to learn to use any tool is to actually use it. In this tutorial we will answer some basic questions about a dataset using agate.

The data will be using is a copy of the National Registery of Exonerations made on August 28th, 2015. This dataset lists individuals who are known to have been exonerated after having been wrongly convicted. At the time the data was exported there were 1,651 entries in the registry.

3.2.2 Installing agate

Installing agate is easy:

```
pip install agate
```

Note: You should be installing agate inside a virtualenv. If for some crazy reason you aren't using virtualenv you will need to add a `sudo` to the previous command.

3.2.3 Getting the data

Let's start by creating a clean workspace:

```
mkdir agate_tutorial
cd agate_tutorial
```

Now let's download the data:

```
curl -L -O https://github.com/onyxfish/agate/raw/master/examples/realdata/exonerations-20150828.csv
```

You will now have a file named `exonerations-20150828.csv` in your `agate_tutorial` directory.

3.2.4 Getting setup

First launch the Python interpreter:

```
python
```

Now let's import our dependencies:

```
import csv
import agate
```

Note: You should really be using csvkit to load CSV files, but here we stick with the builtin *csv* module because it comes with Python so everyone already has it.

I also strongly suggest taking a look at proof for building data processing pipelines, but we won't use it in this tutorial to keep things simple.

3.2.5 Defining the columns

There are two ways to specify column types in agate. You can specify a particular type one-by-one, which gives you complete control over how the data is processed, or you can use agate's *TypeTester* to infer types from the data. The latter is more convenient, but it is imperfect, so it's wise to check that the types in infers are reasonable. (For instance, some date formats look exactly like numbers and some numbers are really text.)

You can create a *TypeTester* like this:

```
tester = agate.TypeTester()
```

If you prefer to specify your columns manually you will need to create instances of each type that you are using:

```
text_type = agate.Text()
number_type = agate.Number()
boolean_type = agate.Boolean()
```

Then you define the names and types of the columns that are in our dataset as a sequence of pairs. For the exonerations dataset, you would define:

```
columns = (
    ('last_name', text_type),
    ('first_name', text_type),
    ('age', number_type),
    ('race', text_type),
    ('state', text_type),
    ('tags', text_type),
    ('crime', text_type),
    ('sentence', text_type),
    ('convicted', number_type),
    ('exonerated', number_type),
    ('dna', boolean_type),
    ('dna_essential', text_type),
    ('mistaken_witness', boolean_type),
    ('false_confession', boolean_type),
    ('perjury', boolean_type),
    ('false_evidence', boolean_type),
    ('official_misconduct', boolean_type),
    ('inadequate_defense', boolean_type),
)
```

Note: If specifying column names manually they do not necessarily need to match those found in your CSV file. I've kept them consistent in this example for clarity. If using *TypeTester* column names will be inferred from the headers of your CSV.

3.2.6 Loading data from a CSV

The *Table* is the basic class in agate. A time-saving method is included to create a table from CSV. To infer column types automatically while reading the data:

```
exonerations = agate.Table.from_csv('exonerations-20150828.csv', tester)
```

Note: For larger datasets the *TypeTester* can be slow to evaluate the data. It's best to use it with a tool such as proof so you don't have to run it everytime you work with your data.

Or, to use the column types we created manually:

```
exonerations = agate.Table.from_csv('exonerations-20150828.csv', columns)
```

In either case the `exonerations` variable will now be an instance of *Table*.

Note: If you have data that you've generated in another way you can always pass it in the *Table* constructor directly.

3.2.7 Aggregating column data

Analysis begins with questions, so that's how we'll learn about agate.

Question: **How many exonerations involved a false confession?**

Answering this question involves counting the number of "True" values in the `false_confession` column. When we created the table we specified that the data in this column contained *Boolean* data. Because of this, agate has taken care of coercing the original text data from the CSV into Python's `True` and `False` values.

We'll answer the question using *Count* which is a type of *Aggregation*. Aggregations in agate are used to perform "column-wise" calculations. That is, they derive a new single value from the contents of a column. In the case of *Count*, it will tell us how many times a particular value appears in the column.

An *Aggregation* is applied to a column of a table. You can access the columns of a table using the *Table.columns* attribute.

Putting it together looks like this:

```
num_false_confessions = exonerations.columns['false_confession'].aggregate(agate.Count(True))

print(num_false_confessions)
```

```
211
```

Let's look at another example, this time using a numerical aggregation.

Question: **What was the median age of exonerated indviduals at time of arrest?**

```
median_age = exonerations.columns['age'].aggregate(agate.Median())

print(median_age)
```

Answer:

```
agate.exceptions.NullComputationError
```

Apparently, not every exonerated individual in the data has a value for the `age` column. The *Median* statistical operation has no standard way of accounting for null values, so its caused an error.

Question: **How many individuals do not have an age specified in the data?**

```
num_without_age = exonerations.columns['age'].aggregate(agate.Count(None))

print(num_without_age)
```

Answer:

```
9
```

Only nine rows in this dataset don't have age, so it's still useful to compute a median, but to do this we'll need to filter out those null values first.

Each column in *Table.columns* is an instance of *Column*. As we've seen with *Median*, different aggregations can be applied depending on the type of data in the column and, in this case, the specific values.

If none of the provided aggregations suit your needs you can also easily create your own by subclassing *Aggregation*. See the API documentation for *aggregations* to see all of the implemented types.

3.2.8 Selecting and filtering data

So how can we answer our question about median age? First, we need to filter the data to only those rows that don't contain nulls.

Agate's *Table* class provides a full suite of these "SQL-like" operations, including *Table.select()* for grabbing specific columns, *Table.where()* for selecting particular rows and *Table.group_by()* for grouping rows by common values.

Let's filter our exonerations table to only those individuals that have an age specified.

```
with_age = exonerations.where(lambda row: row['age'] is not None)
```

You'll notice we provide a `lambda` (anonymous) function to the *Table.where()*. This function is applied to each row and if it returns `True`, the row is included in the output table.

A crucial thing to understand about these methods is that they return **new tables**. In our example above `exonerations` was a *Table* instance and we applied *Table.where()*, so `with_age` is a `Table` too. The tables themselves are immutable. You can create new tables, but you can never modify them.

We can verify this did what we expected by counting the rows in the original table and rows in the new table:

```
old = len(exonerations.rows)
new = len(with_age.rows)

print(old - new)
```

```
9
```

Nine rows were removed, which is how many we knew had nulls for the age column.

So, what **is** the median age of these individuals?

```
median_age = with_age.columns['age'].aggregate(agate.Median())

print(median_age)
```

```
26
```

3.2.9 Computing new columns

In addition to "column-wise" calculations there are also "row-wise" calculations. These calculations go through a *Table* row-by-row and derive a new column using the existing data. To perform row calculations in agate we use subclasses of *Computation*.

When one or more instances of *Computation* are applied to a *Table*, a new table is created with additional columns.

Question: **How long did individuals remain in prison before being exonerated?**

To answer this question we will apply the *Change* computation to the `convicted` and `exonerated` columns. All that *Change* does is compute the difference between two numbers. (In this case each of these columns contains an integer year, but agate does have features for working with dates too.)

```
with_years_in_prison = exonerations.compute([
    (agate.Change('convicted', 'exonerated'), 'years_in_prison')
])

median_years = with_years_in_prison.columns['years_in_prison'].aggregate(agate.Median())

print(median_years)
```

```
8
```

The median number of years an exonerated individual spent in prison was 8 years.

Sometimes, the built-in computations, such as *Change* won't suffice. In this case, you can use the generic *Formula* to compute a column based on an arbitrary function. This is somewhat analogous to Excel's cell formulas.

For instance, this example will create a `full_name` column from the `first_name` and `last_name` columns in the data:

```
full_names = exonerations.compute([
    (agate.Formula(text_type, lambda row: '%(first_name)s %(last_name)s' % row), 'full_name')
])
```

For efficiencies sake, agate allows you to perform several computations at once.

```
with_computations = exonerations.compute([
    (agate.Formula(text_type, lambda row: '%(first_name)s %(last_name)s' % row), 'full_name'),
    (agate.Change('convicted', 'exonerated'), 'years_in_prison')
])
```

If *Formula* still is not flexible enough (for instance, if you need to compute a new row based on the distribution of data in a column) you can always implement your own subclass of *Computation*. See the API documentation for *computations* to see all of the supported ways to compute new data.

3.2.10 Sorting and slicing

Question: **Who are the ten exonerated individuals who were youngest at the time they were arrested?**

Remembering that methods of tables return tables, we will use *Table.order_by()* to sort our table:

```
sorted_by_age = exonerations.order_by('age')
```

We can then use *Table.limit()* get only the first ten rows of the data.

```
youngest_ten = sorted_by_age.limit(10)
```

Now let's use *Table.print_table()* to help us pretty the results in a way we can easily review:

```
youngest_ten.print_table(max_columns=7)
```

```
|------------+------------+-----+-----------+-------+---------+---------+------|
|  last_name | first_name | age | race      | state | tags    | crime   | ...  |
|------------+------------+-----+-----------+-------+---------+---------+------|
|  Murray    | Lacresha   | 11  | Black     | TX    | CV, F   | Murder  | ...  |
|  Adams     | Johnathan  | 12  | Caucasian | GA    | CV, P   | Murder  | ...  |
|  Harris    | Anthony    | 12  | Black     | OH    | CV      | Murder  | ...  |
|  Edmonds   | Tyler      | 13  | Caucasian | MS    |         | Murder  | ...  |
|  Handley   | Zachary    | 13  | Caucasian | PA    | A, CV   | Arson   | ...  |
|  Jimenez   | Thaddeus   | 13  | Hispanic  | IL    |         | Murder  | ...  |
|  Pacek     | Jerry      | 13  | Caucasian | PA    |         | Murder  | ...  |
```

```
|  Barr       | Jonathan   | 14  | Black      | IL    | CDC, CV | Murder  | ...  |
|  Brim       | Dominique  | 14  | Black      | MI    | F       | Assault | ...  |
|  Brown      | Timothy    | 14  | Black      | FL    |         | Murder  | ...  |
|-------------+------------+-----+------------+-------+---------+---------+------|
```

If you find it impossible to believe that an eleven year-old was convicted of murder, I encourage you to read the Registry's description of the case.

Note: In the previous example we could have omitted the *Table.limit()* and passed a `max_rows=10` to *Table.print_table()* instead.

3.2.11 Grouping and aggregating

Question: **Which state has seen the most exonerations?**

This question can't be answered by operating on a single column. What we need is the equivalent of SQL's `GROUP BY`. agate supports a full set of SQL-like operations on tables. Unlike SQL, agate breaks grouping and aggregation into two discrete steps.

First, we use *Table.group_by()* to group the data by state.

```
by_state = exonerations.group_by('state')
```

This takes our original *Table* and groups it into a *TableSet*, which contains one table per county. Now we need to aggregate the total for each state. This works in a very similar way to how it did when we were aggregating columns of a single table, except that we'll use the *Length* aggregation to count the total number of values in the column.

```
state_totals = by_state.aggregate([
    ('state', agate.Length(), 'count')
])

sorted_totals = state_totals.order_by('count', reverse=True)

sorted_totals.print_table(max_rows=5)
```

```
|--------+--------|
|  state | count  |
|--------+--------|
|  TX    | 212    |
|  NY    | 202    |
|  CA    | 154    |
|  IL    | 153    |
|  MI    | 60     |
|  ...   | ...    |
|--------+--------|
```

You'll notice we pass a list of tuples to *TableSet.aggregate()*. Each one includes three elements. The first is the column name to aggregate. The second is an instance of some *Aggregation*. The third is the new column name. Unsurpringly, in this case the results appear roughly proportional to population.

Question: **What state has the longest median time in prison prior to exoneration?**

This is a much more complicated question that's going to pull together a lot of the features we've been using. We'll repeat the computations we applied before, but this time we're going to roll those computations up in our group and take the *Median* of each group. Then we'll sort the data and see where people have been stuck in prison the longest.

```
with_years_in_prison = exonerations.compute([
    (agate.Change('convicted', 'exonerated'), 'years_in_prison')
])

state_totals = with_years_in_prison.group_by('state')

medians = state_totals.aggregate([
    ('years_in_prison', agate.Length(), 'count'),
    ('years_in_prison', agate.Median(), 'median_years_in_prison')
])

sorted_medians = medians.order_by('median_years_in_prison', reverse=True)

sorted_medians.print_table(max_rows=5)
```

```
|--------+-------+-------------------------|
|  state | count | median_years_in_prison  |
|--------+-------+-------------------------|
|  DC    | 15    | 27                      |
|  NE    | 9     | 20                      |
|  ID    | 2     | 19                      |
|  VT    | 1     | 18                      |
|  LA    | 45    | 16                      |
|  ...   | ...   | ...                     |
|--------+-------+-------------------------|
```

DC? Nebraska? What accounts for these states having the longest times in prison before exoneration? I have no idea. Given that the group sizes are small, it would probably be wise to look for outliers.

As with `Table.aggregate()` and *`Table.compute()`*, the *`TableSet.aggregate()`*: method takes a list of aggregations to perform. You can aggregate as many columns as you like in a single step and they will all appear in the output table.

3.2.12 Multi-dimensional aggregation

Before we wrap up, let's try one more thing. I've already shown you that you can use *`TableSet`* to group instances of *`Table`*. However, you can also use a *`TableSet`* to group other instances of *`TableSet`*. To put that another way, instances of *`TableSet`* can be *nested*.

The key to nesting data in this way is to use `TableSet.group_by()`. Before we used *`Table.group_by()`* to split data up into a group of tables. Now we'll use `TableSet.group_by()` to further subdivide that data. Let's look at a concrete example.

Question: **Is there a collective relationship between race, age and time spent in prison prior to exoneration?**

I'm not going to explain every stage of this analysis as most of it repeats patterns used previously. The key part to look for is the two separate calls to `group_by`:

```
# Filters rows without age data
only_with_age = with_years_in_prison.where(
    lambda r: r['age'] is not None
)

# Group by race
race_groups = only_with_age.group_by('race')

# Sub-group by age cohorts (20s, 30s, etc.)
race_and_age_groups = race_groups.group_by(
```

```
    lambda r: '%i0s' % (r['age'] // 10),
    key_name='age_group'
)

# Aggregate medians for each group
medians = race_and_age_groups.aggregate([
    ('years_in_prison', agate.Length(), 'count'),
    ('years_in_prison', agate.Median(), 'median_years_in_prison')
])

# Sort the results
sorted_groups = medians.order_by('median_years_in_prison', reverse=True)

# Print out the results
sorted_groups.print_table(max_rows=10)
```

```
|------------------+-----------+-------+-------------------------|
|  race            | age_group | count | median_years_in_prison  |
|------------------+-----------+-------+-------------------------|
|  Native American | 20s       | 2     | 21.5                    |
|                  | 20s       | 1     | 19                      |
|  Native American | 10s       | 2     | 15                      |
|  Native American | 30s       | 2     | 14.5                    |
|  Black           | 10s       | 188   | 14                      |
|  Black           | 20s       | 358   | 13                      |
|  Asian           | 20s       | 4     | 12                      |
|  Black           | 30s       | 156   | 10                      |
|  Caucasian       | 10s       | 76    | 8                       |
|  Caucasian       | 20s       | 255   | 8                       |
|  ...             | ...       | ...   | ...                     |
|------------------+-----------+-------+-------------------------|
```

Well, what are you waiting for? It's your turn!

3.2.13 Where to go next

This tutorial only scratches the surface of agate's features. For many more ideas on how to apply agate, check out the Cookbook, which includes dozens of examples showing how to substitute agate for common patterns used in Excel, SQL, R and more. Also check out the agate's Extensions which add support for reading/writing SQL tables, rendering charts and more.

Also, if you're going to be doing data processing in Python you really ought to check out proof, a library for building data processing pipelines that are repeatable and self-documenting. It will make your code cleaner and save you tons of time.

3.3 Cookbook

3.3.1 Basics

You can always use Python's builtin `csv` to read and write CSV files, but agate also includes shortcuts to save time.

Note: If you have csvkit installed, agate will use it instead of Python's builtin `csv`. The builting module is not unicode-safe for Python 2, so it is strongly suggested that you do install csvkit.

Load table from a CSV

Assuming your file has a single row of headers:

```
text_type = agate.Text()
number_type = agate.Number()

columns = (
    ('city', text_type),
    ('area', number_type),
    ('population', number_type)
)

table = agate.Table.from_csv('population.csv', columns)
```

If your file does not have headers:

```
table = agate.Table.from_csv('population.csv', columns, header=False)
```

Write a table to a CSV

```
table.to_csv('output.csv')
```

Load a table from a SQL database

Use the agate-sql extension.

```
import agatesql

table = agate.Table.from_sql('postgresql:///database', 'input_table')
```

Write a table to a SQL database

Use the agate-sql extension.

```
import agatesql

table.to_sql('postgresql:///database', 'output_table')
```

Guess column types

When loading data into a *Table* instead of specifying each column's type you can instead opt to have agate "guess" what the type of each column is. The advantage of this is that it's much quicker to get started with your analysis. The disadvantage is that it might sometimes guess wrong. Either way, using this feature will never break your code. If agate can't figure out the type of a column it was always fall back to *Text*.

The class which implements the type guessing is *TypeTester*. It supports a `force` argument which allows you to override the type guessing.

```
tester = agate.TypeTester(force={
    'fips': agate.Text()
})

table = agate.Table.from_csv('counties.csv', tester)
```

Note: For larger datasets the *TypeTester* can be slow to evaluate the data. It's best to use it with a tool such as proof so you don't have to run it everytime you work with your data.

Reorder columns

You can reorder the columns in a table by using the *Table.select()* method and specifying the column names in the order you want:

```
new_table = table.select(['3rd_column_name', '1st_column_name', '2nd_column_name'])
```

3.3.2 Filter

By regex

You can use Python's builtin `re` module to introduce a regular expression into a *Table.where()* query.

For example, here we find all states that start with "C".

```
import re

new_table = table.where(lambda row: re.match('^C', row['state']))
```

This can also be useful for finding values that **don't** match your expectations. For example, finding all values in the "phone number" column that don't look like phone numbers:

```
new_table = table.where(lambda row: not re.match('\d{3}-\d{3}-\d{4}', row['phone']))
```

By glob

Hate regexes? You can use glob (a.k.a. `fnmatch`) syntax too!

```
from fnmatch import fnmatch

new_table = table.where(lambda row: fnmatch('C*', row['state'])
```

Values within a range

This snippet filters the dataset to incomes between 100,000 and 200,000.

```
new_table = table.where(lambda row: 100000 < row['income'] < 200000)
```

Dates within a range

This snippet filters the dataset to events during the summer of 2015:

```
import datetime

new_table = table.where(lambda row: datetime.datetime(2015, 6, 1) <= row['date'] <= datetime.datetime
```

If you want to filter to events during the summer of any year:

```
new_table = table.where(lambda row: 6 <= row['date'].month <= 8)
```

Top N percent

To filter a dataset to the top 10% percent of values we first compute the percentiles for the column and then use the result in the *Table.where()* truth test:

```
percentiles = table.columns['salary'].percentiles()
top_ten_percent = table.where(lambda r: r['salary'] >= percentiles[90])
```

Random sample

By combining a random sort with limiting, we can effectively get a random sample from a table.

```
import random

randomized = table.order_by(lambda row: random.random())
sampled = table.limit(10)
```

Ordered sample

With can also get an ordered sample by simply using the `step` parameter of the *Table.limit()* method to get every Nth row.

```
sampled = table.limit(step=10)
```

3.3.3 Sort

Alphabetical

Order a table by the `last_name` column:

```
new_table = table.order_by('last_name')
```

Numerical

Order a table by the `cost` column:

```
new_table = table.order_by('cost')
```

By date

Order a table by the `birth_date` column:

```
new_table = table.order_by('birth_date')
```

Reverse order

The order of any sort can be reversed by using the `reverse` keyword:

```
new_table = table.order_by('birth_date', reverse=True)
```

Multiple columns

Because Python's internal sorting works natively with arrays, we can implement multi-column sort by returning a tuple from the key function.

```
new_table = table.order_by(lambda row: (row['last_name'], row['first_name']))
```

This table will now be ordered by `last_name`, then `first_name`.

Random order

```
import random

new_table = table.order_by(lambda row: random.random())
```

3.3.4 Search

Exact search

Find all individuals with the last_name "Groskopf":

```
family = table.where(lambda r: r['last_name'] == 'Groskopf')
```

Fuzzy search by edit distance

Using an existing Python library for computing the Levenshtein edit distance it is trivially easy to implement a fuzzy string search.

For example, to find all names within 2 edits of "Groskopf":

```
from Levenshtein import distance

fuzzy_family = table.where(lambda r: distance(r['last_name'], 'Groskopf') <= 2)
```

These results will now include all those "Grosskopfs" and "Groskoffs" whose mail I am always getting.

3.3.5 Statistics

Descriptive statistics

agate includes a full set of standard descriptive statistics that can be applied to any column containing *`Number`* data.

```
column = table.columns['salary']

column.aggregate(Sum())
column.aggregate(Min())
column.aggregate(Max())
column.aggregate(Mean())
column.aggregate(Median())
column.aggregate(Mode())
column.aggregate(Variance())
column.aggregate(StDev())
column.aggregate(MAD())
```

Aggregate statistics

You can also generate aggregate statistics for subsets of data (sometimes colloquially referred to as "rolling up".

```
doctors = patients.group_by('doctor')
patient_ages = doctors.aggregate([
    ('age', agate.Length(), 'patient_count')
    ('age', agate.Mean(), 'age_mean'),
    ('age', agate.Median(), 'age_median')
])
```

The resulting table will have four columns: `doctor`, `patient_count`, `age_mean` and `age_median`.

Identify outliers

Use the agate-stats extension to add methods for finding outliers.

```
import agatestats

agatestats.patch()

outliers = table.stdev_outliers('salary', deviations=3, reject=False)
```

By specifying `reject=True` you can instead return a table including only those values **not** identified as outliers.

```
not_outliers = table.stdev_outliers('salary', deviations=3, reject=True)
```

The second, more robust, method for identifying outliers is by identifying values which are more than some number of "median absolute deviations" from the median (typically 3).

```
outliers = table.mad_outliers('salary', deviations=3, reject=False)
```

As with the first example, you can specify `reject=True` to exclude outliers in the resulting table.

3.3.6 Compute new values

Annual change

You could use a *`Formula`* to calculate percent change, however, for your convenience agate has a built-in shortcut. For example, if your spreadsheet has a column with values for each year you could do:

```
new_table = table.compute([
    (Change('2000', '2001'), '2000_change'),
    (Change('2001', '2002'), '2001_change'),
    (Change('2002', '2003'), '2002_change')
])
```

Or, better yet, compute the whole decade using a loop:

```
computations = []

for year in range(2000, 2010):
    change = Change(year, year + 1)
    computations.append((change, '%i_change' % year))

new_table = table.compute(computations)
```

Annual percent change

Want percent change instead of value change? Just swap out the *Aggregation*:

```
computations = []

for year in range(2000, 2010):
    change = PercentChange(year, year + 1)
    computations.append((change, '%i_change' % year))

new_table = table.compute(computations)
```

Indexed/cumulative change

Need your change indexed to a starting year? Just fix the first argument:

```
computations = []

for year in range(2000, 2010):
    change = Change(2000, year + 1)
    computations.append((change, '%i_change' % year))

new_table = table.compute(computations)
```

Of course you can also use *PercentChange* if you need percents rather than values.

Round to two decimal places

agate stores numerical values using Python's `decimal.Decimal` type. This data type ensures numerical precision beyond what is supported by the native `float()` type, however, because of this we can not use Python's builtin `round()` function. Instead we must use `decimal.Decimal.quantize()`.

We can use *Table.compute()* to apply the quantize to generate a rounded column from an existing one:

```
from decimal import Decimal

number_type = agate.Number()

def round_price(row):
```

```
    return row['price'].quantize(Decimal('0.01'))

new_table = table.compute([
    (Formula(number_type, round_price), 'price_rounded')
])
```

To round to one decimal place you would simply change `0.01` to `0.1`.

Difference between dates

Calculating the difference between dates (or dates and times) works exactly the same as it does for numbers:

```
new_table = table.compute([
    (Change('born', 'died'), 'age_at_death')
])
```

Levenshtein edit distance

The Levenshtein edit distance is a common measure of string similarity. It can be used, for instance, to check for typos between manually-entered names and a version that is known to be spelled correctly.

Implementing Levenshtein requires writing a custom *Computation*. To save ourselves building the whole thing from scratch, we will lean on the python-Levenshtein library for the actual algorithm.

```
import agate
from Levenshtein import distance
import six

class LevenshteinDistance(agate.Computation):
    """
    Computes Levenshtein edit distance between the column and a given string.
    """
    def __init__(self, column_name, compare_string):
        self._column_name = column_name
        self._compare_string = compare_string

    def get_computed_column_type(self, table):
        """
        The return value is a numerical distance.
        """
        return agate.Number()

    def prepare(self, table):
        """
        Verify the column is text.
        """
        column = table.columns[self._column_name]

        if not isinstance(column.data_type, agate.Text):
            raise agate.DataTypeError('Can only be applied to Text data.')

    def run(self, row):
        """
        Find the distance, returning null when the input column was null.
        """
        val = row[self._column_name]
```

```
        if val is None:
            return None

        return distance(val, self._compare_string)
```

This code can now be applied to any *Table* just as any other *Computation* would be:

```
new_table = table.compute([
    (LevenshteinDistance('column_name', 'string to compare'), 'distance')
])
```

The resulting column will contain an integer measuring the edit distance between the value in the column and the comparison string.

USA Today Diversity Index

The USA Today Diversity Index is a widely cited method for evaluating the racial diversity of a given area. Using a custom *Computation* makes it simple to calculate.

Assuming that your data has a column for the total population, another for the population of each race and a final column for the hispanic population, you can implement the diversity index like this:

```
class USATodayDiversityIndex(agate.Computation):
    def get_computed_column_type(self, table):
        return agate.Number()

    def run(self, row):
        race_squares = 0

        for race in ['white', 'black', 'asian', 'american_indian', 'pacific_islander']:
            race_squares += (row[race] / row['population']) ** 2

        hispanic_squares = (row['hispanic'] / row['population']) ** 2
        hispanic_squares += (1 - (row['hispanic'] / row['population'])) ** 2

        return (1 - (race_squares * hispanic_squares)) * 100
```

We apply the diversity index like any other computation:

```
with_index = table.compute([
    (USATodayDiversityIndex(), 'diversity_index')
])
```

3.3.7 Rank

There are many ways to rank a sequence of values. agate strives to find a balance between simple, intuitive ranking and flexibility when you need it.

Competition rank

The basic rank supported by agate is standard "competition ranking". In this model the values `[3, 4, 4, 5]` would be ranked `[1, 2, 2, 4]`. You can apply competition ranking using the *Rank* computation:

```
new_table = table.compute([
    (agate.Rank('value'), 'rank')
])
```

Rank descending

Descending competition ranking is specified using the `reverse` argument.

```
new_table = table.compute([
    (agate.Rank('value', reverse=True), 'rank')
])
```

Rank change

You can compute the change from one rank to another by combining the *Rank* and class:.*Change* computations:

```
new_table = table.compute([
    (agate.Rank('value2014'), 'rank2014'),
    (agate.Rank('value2015'), 'rank2015')
])

new_table2 = new_table.compute([
    (agate.Change('rank2014', 'rank2015'), 'rank_change')
])
```

Percentile rank

"Percentile rank" is a bit of a misnomer. Really, this is the percentile in which each value in a column is located. This column can be computed for your data using the *PercentileRank* computation:

```
new_table = table.compute([
    (agate.PercentileRank('value'), 'percentile_rank')
])
```

Note that there is no entirely standard method for computing percentiles. The percentiles computed in this manner may not agree with those generated by other programes. See the *Percentiles* class documentation for implementation details.

3.3.8 Dates and times

agate includes robust support for working columns of data that are instances of `datetime.date`, `datetime.datetime` or `datetime.timedelta`.

Infer a date format

By default, agate will attempt to infer the format of a date column:

```
text_type = agate.Text()
date_type = agate.Date()

columns = (
    ('name', text_type),
```

```
    ('date', date_type)
)

table = agate.Table.from_csv('events.csv', columns)
```

Specify a date format

In some cases, it may not be possible to automatically parse the format of a date. In this case you can specify a `datetime.datetime.strptime()` formatting string to specify how the dates should be parsed. For example, if your dates were formatted as "15-03-15" (March 15th, 2015) then you could specify:

```
date_type = agate.Date('%d-%m-%y')
```

Another use for this feature is if you have a column that contains extraneous data. For instance, imagine that your column contains hours and minutes, but they are always zero. It would make more sense to load that data as type *`Date`* and ignore the extra time information:

```
date_type = agate.Date('%m/%d/%Y 00:00')
```

Specify a timezone

Timezones are hard. Under normal circumstances (no arguments specified), agate will not try to parse timezone information, nor will it apply a timezone to the `datetime.datetime` instances it creates. All the data it constructs will be *naive*. There are two ways to get timezone data into your agate columns.

The first is to use a format string, as shown above, and specify a pattern for timezone information:

```
datetime_type = agate.DateTime('%Y-%m-%d %H:%M:%S%z')
```

The second way is to specify a timezone as an argument to the type constructor:

```
import pytz

eastern = pytz.timezone('US/Eastern')
datetime_type = agate.DateTime(timezone=eastern)
```

In this case all timezones that are processed will be set to have the Eastern timezone. Note, the will be **set**, not converted. You can not use this method to convert your timezones from UTC to another timezone. To do that see *Convert timezones*.

Calculate a time difference

See *Difference between dates*.

Sort by date

See *By date*.

Convert timezones

If you load data from a spreadsheet in one timezone and you need to convert it to another, you can do this using a *`Formula`*. Your datetime column must have timezone data for the following example to work. See *Specify a timezone*.

```
import pytz

us_eastern = pytz.timezone('US/Eastern')
datetime_type = agate.DateTime(timezone=us_eastern)

columns = (
    ('what', text_type),
    ('when', datetime_type)
)

table = agate.Table.from_csv('events.csv', columns)

rome = timezone('Europe/Rome')
timezone_shifter = agate.Formula(lambda r: r['when'].astimezone(rome))

table = agate.Table.compute([
    (timezone_shifter, 'when_in_rome')
])
```

3.3.9 Locales

agate tries very hard to adequately support non-US, non-English users. This means properly handling foreign currencies, date formats, etc. To facilitate this, agate makes a hard distinction between *your* locale and the locale of *the data* you are working with. This allows you to work seamlessly with data from other countries.

Set your locale

Specifying your current locale works the same as with any other Python module. Please see the `locale` documentation for more details. Changes to your locale will automatically change how agate data is printed to the console and serialized to CSV files, but will not effect how data is *parsed*. See *Specify locale of numbers* for that.

Specify locale of numbers

To correctly parse numbers from non-US locales, you can pass a `locale` parameter to the *`Number`* constructor. For example, to parse Dutch numbers (which use a period to separate thousands and a comma to separate decimals):

```
dutch_numbers = agate.Number(locale='de_DE')

columns = (
    ('city', text_type),
    ('population', dutch_numbers)
)

table = agate.Table.from_csv('dutch_cities.csv', columns)
```

3.3.10 Emulate SQL

agate's command structure is very similar to SQL. The primary difference between agate and SQL is that commands like `SELECT` and `WHERE` explicitly create new tables. You can chain them together as you would with SQL, but be aware each command is actually creating a new table.

Note: All examples in this section use the PostgreSQL dialect for comparison.

If you want to read and write data from SQL, see *Load a table from a SQL database*.

SELECT

SQL:

```
SELECT state, total FROM table;
```

agate:

```
new_table = table.select(('state', 'total'))
```

WHERE

SQL:

```
SELECT * FROM table WHERE LOWER(state) = 'california';
```

agate:

```
new_table = table.where(lambda row: row['state'].lower() == 'california')
```

ORDER BY

SQL:

```
SELECT * FROM table ORDER BY total DESC;
```

agate:

```
new_table = table.order_by(lambda row: row['total'], reverse=True)
```

DISTINCT

SQL:

```
SELECT DISTINCT ON (state) * FROM table;
```

agate:

```
new_table = table.distinct('state')
```

Note: Unlike most SQL implementations, agate always returns the full row. Use *`Table.select()`* if you want to filter the columns first.

INNER JOIN

SQL (two ways):

```
SELECT * FROM patient, doctor WHERE patient.doctor = doctor.id;

SELECT * FROM patient INNER JOIN doctor ON (patient.doctor = doctor.id);
```

agate:

```
joined = patients.join(doctors, 'doctor', 'id', inner=True)
```

LEFT OUTER JOIN

SQL:

```
SELECT * FROM patient LEFT OUTER JOIN doctor ON (patient.doctor = doctor.id);
```

agate:

```
joined = patients.join(doctors, 'doctor', 'id')
```

GROUP BY

agate's *Table.group_by()* works slightly different than SQLs. It does not require an aggregate function. Instead it returns *TableSet*. To see how to perform the equivalent of a SQL aggregate, see below.

```
doctors = patients.group_by('doctor')
```

Chain commands together

SQL:

```
SELECT state, total FROM table WHERE LOWER(state) = 'california' ORDER BY total DESC;
```

agate:

```
new_table = table \
    .select(('state', 'total')) \
    .where(lambda row: row['state'].lower() == 'california') \
    .order_by('total', reverse=True)
```

Note: Chaining commands in this way is often not a good idea. Being explicit about each step tends to produce clearer code.

Aggregate functions

SQL:

```
SELECT mean(age), median(age) FROM patients GROUP BY doctor;
```

agate:

```
doctors = patients.group_by('doctor')
patient_ages = doctors.aggregate([
    ('age', agate.Length(), 'patient_count')
    ('age', agate.Mean(), 'age_mean'),
    ('age', agate.Median(), 'age_median')
])
```

The resulting table will have four columns: `doctor`, `patient_count`, `age_mean` and `age_median`.

3.3.11 Emulate Excel

One of agate's most powerful assets is that instead of a wimpy "formula" language, you have the entire Python language at your disposal. Here are examples of how to translate a few common Excel operations.

Simple formulas

If you need to simulate a simple Excel formula you can use the *Formula* class to apply an arbitrary function.

Excel:

```
=($A1 + $B1) / $C1
```

agate:

```
def f(row):
    return (row['a'] + row['b']) / row['c']

new_table = table.compute([
    (Formula(f), 'new_column')
])
```

If this still isn't enough flexibility, you can also create your own subclass of *Computation*.

SUM

```
number_type = agate.Number()

def five_year_total(row):
    columns = ('2009', '2010', '2011', '2012', '2013')

    return sum(tuple(row[c] for c in columns)]

formula = agate.Formula(number_type, five_year_total)

new_table = table.compute([
    (formula, 'five_year_total')
])
```

TRIM

```
new_table = table.compute([
    (Formula(text_type, lambda r: r['name'].strip()), 'name_stripped')
])
```

CONCATENATE

```
new_table = table.compute([
    (Formula(text_type, lambda r: '%(first_name)s %(middle_name)s %(last_name)s' % r), 'full_name')
])
```

IF

```
new_table = table.compute([
    (Formula(boolean_type, lambda r: row['batting_average'] > 0.3), 'mvp_candidate')
])
```

VLOOKUP

```
states = {
    'AL': 'Alabama',
    'AK': 'Alaska',
    'AZ': 'Arizona',
    ...
}

new_table = table.compute([
    (Formula(text_type, lambda r: states[row['state_abbr']]), 'mvp_candidate')
])
```

Pivot tables

You can emulate most of the functionality of Excel's pivot tables using the *TableSet.aggregate()* method.

```
jobs = employees.group_by('job_title')
summary = jobs.aggregate([
    ('salary', agate.Length(), 'employee_count')
    ('salary', agate.Mean(), 'salary_mean'),
    ('salary', agate.Median(), 'salary_median')
])
```

The resulting `summary` table will have four columns: `job_title`, ``employee_count`, `salary_mean` and `salary_median`.

3.3.12 Emulate R

aggregate

R:

```
aggregate(employees$salary, list(job = employees$job), mean)
```

agate:

```
jobs = employees.group_by('job')
aggregates = jobs.aggregate([
    ( 'salary', 'mean')
])
```

3.3.13 Emulate underscore.js

filter

agate's *Table.where()* functions exactly like Underscore's `filter`.

```
new_table = table.where(lambda row: row['state'] == 'Texas')
```

reject

To simulate Underscore's `reject`, simply negate the return value of the function you pass into agate's *`Table.where()`*.

```
new_table = table.where(lambda row: not (row['state'] == 'Texas'))
```

find

agate's *`Table.find()`* works exactly like Undrescore's `find`.

```
row = table.find(lambda row: row['state'].startswith('T'))
```

any

agate's columns have an `Column.any()` method that functions like Underscore's `any`.

```
true_or_false = table.columns['salaries'].any(lambda d: d > 100000)
```

You can also use *`Table.where()`* to filter to columns that pass the truth test.

all

agate's columns have an `Column.all()` method that functions like Underscore's `all`.

```
true_or_false = table.columns['salaries'].all(lambda d: d > 100000)
```

You can also use *`Table.where()`* to filter to columns that pass the truth test.

3.3.14 Rendering charts

Using matplotlib

Here is an example of how you might generate a line chart:

```
import pylab

pylab.plot(table.columns['homeruns'], table.columns['wins'])
pylab.xlabel('Homeruns')
pylab.ylabel('Wins')
pylab.title('How homeruns correlate to wins')

pylab.show()
```

A similar example that draws a histogram:

```
pylab.hist(table.columns['state'])

pylab.xlabel('State')
pylab.ylabel('Count')
pylab.title('Count by state')
```

```
pylab.show()
```

3.4 Extensions

agate's core featureset is designed rely on as few dependencies as possible. However, in real life you're often going to want to interface agate with SQL, numpy or other data pipelines.

3.4.1 How extensions work

In order to support these use-cases, but not make things excessively complicated, agate support's a simple extensibility pattern based on monkey patching. Libraries can be created that patch new methods on *Table* and *TableSet*. For example, agate-sql adds the ability to read and write tables from a SQL database:

```
import agate
import agatesql

agatesql.patch()

# After calling patch the from_sql and to_sql methods are now part of the Table class
table = agate.Table.from_sql('postgresql:///database', 'input_table')
table.to_sql('postgresql:///database', 'output_table')
```

3.4.2 List of extensions

Here is a list of actively supported agate extensions:

- agate-sql: Read and write tables in SQL databases
- agate-stats: Additional statistical methods

3.4.3 Writing your own extensions

Writing your own extensions is straightforward. Create a class that acts as your "patch" and when you call `Table.monkeypatch()` it will dynamically be added as a base class of *Table*.

```
import agate

class ExamplePatch(object):
    def new_method(self):
        print('I do something to a Table when you call me.')
```

Then create a function that applies your patch:

```
def patch()
    agate.Table.monkeypatch(ExamplePatch)
```

The *Table* class will now have all the methods of `ExamplePatch` as though they were defined as part of it.

```
>>> import agate
>>> import myextension
>>> myextension.patch()
>>> table = agate.Table(rows, columns)
```

```
>>> table.new_method()
'I do something to a Table when you call me.'
```

The same pattern also works for adding methods to *TableSet*.

Warning: Extensions are added as **base classes** of *Table* so you can not use them to override the implementation of an existing method. They are perfect for adding features, but if you need to actually modify how agate works, then you'll need to use a subclass. Any shadowed method will be ignored.

3.5 API

3.5.1 agate.aggregations

This module contains the *Aggregation* class and its various subclasses. Each of these classes processes a column's data and returns some value(s). For instance, *Mean*, when applied to a column containing *Number* data, returns a single `decimal.Decimal` value which is the average of all values in that column.

Aggregations are applied to instances of *Column* using the *Column.aggregate()* method. Typically, the column is first retrieved using the *Table.columns* attribute.

Most aggregations can also be applied to instances of *TableSet* using the `Tableset.aggregate()` method, in which case the result will be a new *Table* with a column for each aggregation and a row for each table in the set.

class `agate.aggregations.`**Aggregation**

Bases: `object`

Base class defining an operation that can be executed on a column using `Table.aggregate()` or on a set of columns using `TableSet.aggregate`.

get_cache_key()

Aggregations can optionally define a cache key that uniquely identifies this operation. If they do they future invocations of this aggregation with the same cache key applied to the same column will use the cached value.

get_aggregate_data_type(*column*)

Get the data type that should be used when using this aggregation with a *TableSet* to produce a new column.

Should raise *UnsupportedAggregationError* if this column does not support aggregation into a *TableSet*. (For example, if it does not return a single value.)

run(*column*)

Execute this aggregation on a given column and return the result.

class `agate.aggregations.`**HasNulls**

Bases: *agate.aggregations.Aggregation*

Returns `True` if the column contains null values.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns `bool`

class agate.aggregations.**Any**(*test=None*)
Bases: *agate.aggregations.Aggregation*

Returns `True` if any value in a column passes a truth test. The truth test may be omitted when testing *Boolean* data.

Parameters **test** – A function that takes a value and returns `True` or `False`.

get_aggregate_data_type(*column*)

run(*column*)

Returns `bool`

class agate.aggregations.**All**(*test=None*)
Bases: *agate.aggregations.Aggregation*

Returns `True` if all values in a column pass a truth test. The truth test may be omitted when testing *Boolean* data.

Parameters **test** – A function that takes a value and returns `True` or `False`.

get_aggregate_data_type(*column*)

run(*column*)

Returns `bool`

class agate.aggregations.**Length**
Bases: *agate.aggregations.Aggregation*

Count the total number of values in the column.

Equivalent to calling `len()` on a *Column*.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns `int`

class agate.aggregations.**Count**(*value*)
Bases: *agate.aggregations.Aggregation*

Count the number of times a specific value occurs in a column.

If you want to count the total number of values in a column use *Length*.

Parameters **value** – Any value to be counted, including `None`.

get_aggregate_data_type(*column*)

run(*column*)

Returns `int`

class agate.aggregations.**Min**
Bases: *agate.aggregations.Aggregation*

Compute the minimum value in a column. May be applied to columns containing *DateTime* or *Number* data.

get_aggregate_data_type(*column*)

run(*column*)

Returns `datetime.date`

class agate.aggregations.**Max**
Bases: *agate.aggregations.Aggregation*

Compute the maximum value in a column. May be applied to columns containing *DateTime* or *Number* data.

get_aggregate_data_type(*column*)

run(*column*)

Returns datetime.date

class agate.aggregations.**MaxPrecision**
Bases: *agate.aggregations.Aggregation*

Compute the most decimal places present for any value in this column.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Sum**
Bases: *agate.aggregations.Aggregation*

Compute the sum of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Mean**
Bases: *agate.aggregations.Aggregation*

Compute the mean value of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Median**
Bases: *agate.aggregations.Aggregation*

Compute the median value of a column containing *Number* data.

This is the 50th percentile. See *Percentiles* for implementation details.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Mode**
Bases: *agate.aggregations.Aggregation*

Compute the mode value of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**IQR**
Bases: *agate.aggregations.Aggregation*

Compute the interquartile range of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Variance**
Bases: *agate.aggregations.Aggregation*

Compute the sample variance of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**PopulationVariance**
Bases: *agate.aggregations.Variance*

Compute the population variance of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**StDev**
Bases: *agate.aggregations.Aggregation*

Compute the sample standard of deviation of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**PopulationStDev**
Bases: *agate.aggregations.StDev*

Compute the population standard of deviation of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**MAD**
Bases: *agate.aggregations.Aggregation*

Compute the median absolute deviation of a column containing *Number* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns decimal.Decimal.

class agate.aggregations.**Percentiles**
Bases: *agate.aggregations.Aggregation*

Divides a *Number* column into 100 equal-size groups using the "CDF" method.

See this explanation of the various methods for computing percentiles.

"Zeroth" (min value) and "Hundredth" (max value) percentiles are included for reference and intuitive indexing.

A reference implementation was provided by pycalcstats.

This aggregation can not be applied to a *TableSet*.

get_cache_key()

run(*column*)

Returns An array of decimal.Decimal.

class agate.aggregations.**Quartiles**
Bases: *agate.aggregations.Aggregation*

The quartiles of a *Number* column based on the 25th, 50th and 75th percentiles.

"Zeroth" (min value) and "Fourth" (max value) quartiles are included for reference and intuitive indexing.

See *Percentiles* for implementation details.

This aggregation can not be applied to a *TableSet*.

get_cache_key()

run(*column*)

class agate.aggregations.**Quintiles**
Bases: *agate.aggregations.Aggregation*

The quintiles of a column based on the 20th, 40th, 60th and 80th percentiles.

"Zeroth" (min value) and "Fifth" (max value) quintiles are included for reference and intuitive indexing.

See *Percentiles* for implementation details.

This aggregation can not be applied to a *TableSet*.

get_cache_key()

run(*column*)

class agate.aggregations.**Deciles**
Bases: *agate.aggregations.Aggregation*

The deciles of a column based on the 10th, 20th ... 90th percentiles.

"Zeroth" (min value) and "Tenth" (max value) deciles are included for reference and intuitive indexing.

See *Percentiles* for implementation details.

This aggregation can not be applied to a *TableSet*.

get_cache_key()

run(*column*)

class agate.aggregations.**MaxLength**
Bases: *agate.aggregations.Aggregation*

Calculates the longest string in a column containing *Text* data.

get_aggregate_data_type(*column*)

get_cache_key()

run(*column*)

Returns int.

3.5.2 agate.data_types

This module contains the *DataType* class and its subclasses. These types define how data should be converted during the creation of a *Table*.

A *TypeTester* class is also included which be used to infer data types from column data.

class agate.data_types.**TypeTester**(*force={}*, *locale='en_US'*)
Infer data types for the columns in a given set of data.

Parameters

- **force** – A dictionary where each key is a column name and each value is a *DataType* instance that overrides inference.
- **locale** – A locale to use when evaluating the types of data. See *Number*.

run(*rows*, *column_names*)
Apply inference to the provided data and return an array of `(column_name, column_type)` tuples suitable as an argument to *Table*.

Parameters

- **rows** – The data as a sequence of any sequences: tuples, lists, etc.
- **column_names** – A sequence of column names.

agate.data_types.base.**DEFAULT_NULL_VALUES = ('', 'na', 'n/a', 'none', 'null', '.')**
Default values which will be automatically cast to `None`

class agate.data_types.base.**DataType**(*null_values=('', 'na', 'n/a', 'none', 'null', '.')*)
Bases: object

Base class for data types.

Parameters **null_values** – A sequence of values which should be cast to `None` when encountered with this type.

test(*d*)
Test if a given string value could possibly be an instance of this data type.

cast(*d*)
Coerce a given string value into this column's data type.

agate.data_types.boolean.**DEFAULT_TRUE_VALUES = ('yes', 'y', 'true', 't')**
Default values which will be automatically cast to `True`.

agate.data_types.boolean.**DEFAULT_FALSE_VALUES = ('no', 'n', 'false', 'f')**
Default values which will be automatically cast to `False`.

class agate.data_types.boolean.**Boolean**(*true_values=('yes', 'y', 'true', 't')*, *false_values=('no', 'n', 'false', 'f')*, *null_values=('', 'na', 'n/a', 'none', 'null', '.')*)
Bases: *agate.data_types.base.DataType*

Data type representing boolean values.

Parameters

- **true_values** – A sequence of values which should be cast to `True` when encountered with this type.
- **false_values** – A sequence of values which should be cast to `False` when encountered with this type.

test(*d*)
Test, for purposes of type inference, if a string value could possibly be valid for this column type.

cast(*d*)
Cast a single value to `bool`.

Parameters **d** – A value to cast.

Returns `bool` or `None`.

class agate.data_types.date.**Date**(*date_format=None*, ***kwargs*)
Bases: *agate.data_types.base.DataType*

Data type representing dates only.

Parameters **date_format** – A formatting string for `datetime.datetime.strptime()` to use instead of using regex-based parsing.

test(*d*)
Test, for purposes of type inference, if a string value could possibly be valid for this column type.

cast(*d*)
Cast a single value to a `datetime.date`.

Parameters **date_format** – An optional `datetime.strptime()` format string for parsing datetimes in this column.

Returns `datetime.date` or `None`.

class agate.data_types.date_time.**DateTime**(*datetime_format=None*, *timezone=None*, ***kwargs*)
Bases: *agate.data_types.base.DataType*

Data type representing dates and times.

Parameters

- **datetime_format** – A formatting string for `datetime.datetime.strptime()` to use instead of using regex-based parsing.

- **timezone** – A pytz timezone to apply to each parsed date.

test(*d*)
Test, for purposes of type inference, if a string value could possibly be valid for this column type.

cast(*d*)
Cast a single value to a `datetime.datetime`.

Parameters **date_format** – An optional `datetime.strptime()` format string for parsing datetimes in this column.

Returns `datetime.datetime` or `None`.

agate.data_types.number.**CURRENCY_SYMBOLS = [u'\u060b', u'$', u'\u0192', u'\u17db', u'\xa5', u'\u20a1', u'\u20b1',**
A list of currency symbols sourced from Xe.

class agate.data_types.number.**Number**(*locale='en_US'*, *display_precision=2*, ***kwargs*)
Bases: *agate.data_types.base.DataType*

Data type representing numbers.

Parameters

- **locale** – A locale specification such as `en_US` or `de_DE` to use for parsing formatted numbers.
- **display_precision** – An integer specifying how many decimal places to include when formatting this column for display. (Such as when using `Table.pretty_print`.)

test(*d*)
Test, for purposes of type inference, if a string value could possibly be valid for this column type.

cast(*d*)
Cast a single value to a `decimal.Decimal`.

Returns `decimal.Decimal` or `None`.

Raises *CastError*

class agate.data_types.text.**Text**(*null_values=('', 'na', 'n/a', 'none', 'null', '.')*)
Bases: *agate.data_types.base.DataType*

Data type representing text.

test(*d*)
Test, for purposes of type inference, if a string value could possibly be valid for this column type.

cast(*d*)
Cast a single value to `unicode()` (`str()` in Python 3).

Parameters **d** – A value to cast.

Returns `unicode()` (`str()` in Python 3) or `None`

3.5.3 agate.columns

class agate.columns.**Column**(*table*, *index*)
Proxy access to column data. Instances of *Column* should not be constructed directly. They are created by *Table* instances.

Parameters

- **table** – The table that contains this column.
- **index** – The index of this column in the table.

index
This column's index in its parent table.

name
This column's name in its parent table.

data_type
This column's data type.

get_data()
Get the data contained in this column as a `tuple`.

get_data_without_nulls()
Get the data contained in this column with any null values removed.

get_data_sorted()
Get the data contained in this column sorted.

aggregate(*aggregation*)
Apply a *Aggregation* to this column and return the result. If the aggregation defines a

class agate.columns.**ColumnMapping**(*table*)
Proxy access to *Column* instances for *Table*.

Parameters **table** – *Table*.

class agate.columns.**ColumnIterator**(*table*)
Iterator over *Column* instances within a *Table*.

Parameters **table** – *Table*.

3.5.4 agate.computations

This module contains the *Computation* class and its subclasses. Computations allow for row-wise calculation of new data for *Table*. For instance, the *PercentChange* subclass takes two column names as arguments and computes the percentage change between them for each row.

Computations are applied to tables using the *Table.compute()* method. For efficiencies sake, this method accepts a sequence of operations, which are applied simultaneously.

If the basic computations supplied in this module are not suitable to your needs the *Formula* subclass can be used to apply an arbitrary function to the data in each row. If this is still not suitable, *Computation* can be subclassed to fully customize it's behavior.

class agate.computations.**Computation**
Bases: `object`

Base class for row-wise computations on a *Table*.

get_computed_data_type(*table*)
Returns an instantiated *DataType* which will be appended to the table.

prepare(*table*)
Called with the table immediately prior to invoking the computation with rows. Can be used to compute column-level statistics for computations. By default, this does nothing.

run(*row*)
When invoked with a row, returns the computed new column value.

class agate.computations.**Formula**(*data_type*, *func*)
Bases: *agate.computations.Computation*

A simple drop-in computation that can apply any function to rows.

get_computed_data_type(*table*)

run(*row*)

class agate.computations.**Change**(*before_column_name*, *after_column_name*)
Bases: *agate.computations.Computation*

Computes change between two columns.

get_computed_data_type(*table*)

prepare(*table*)

run(*row*)

class agate.computations.**PercentChange**(*before_column_name*, *after_column_name*)
Bases: *agate.computations.Computation*

Computes percent change between two columns.

get_computed_data_type(*table*)

prepare(*table*)

run(*row*)

class agate.computations.**Rank**(*column_name*, *comparer=None*, *reverse=None*)
Bases: *agate.computations.Computation*

Computes rank order of the values in a column.

Uses the "competition" ranking method: if there are four values and the middle two are tied, then the output will be `[1, 2, 2, 4]`.

Null values will always be ranked last.

Parameters

- **column_name** – The name of the column to rank.
- **comparer** – An optional comparison function. If not specified ranking will be ascending, with nulls ranked last.
- **reverse** – Reverse sort order before ranking.

get_computed_data_type(*table*)

prepare(*table*)

run(*row*)

class agate.computations.**PercentileRank**(*column_name*, *comparer=None*, *reverse=None*)
Bases: *agate.computations.Rank*

Assign each value in a column to the percentile into which it falls.

See *Percentiles* for implementation details.

prepare(*table*)

run(*row*)

3.5.5 agate.exceptions

This module contains various exceptions raised by agate.

exception agate.exceptions.**NullCalculationError**
Exception raised if a calculation which can not logically account for null values is attempted on a *Column* containing nulls.

exception agate.exceptions.**DataTypeError**
Exception raised if a process, such as an *Aggregation*, is attempted with an invalid data type.

exception agate.exceptions.**UnsupportedAggregationError**
Exception raised if an aggregation is attempted which is not supported. For example if a *Percentiles* is used on a *TableSet*.

exception agate.exceptions.**CastError**
Exception raised when a column value can not be cast to the correct type.

exception agate.exceptions.**ColumnDoesNotExistError**(*k*)
Exception raised when trying to access a column that does not exist.

Parameters **k** – The key used to access the non-existent *Column*.

exception agate.exceptions.**RowDoesNotExistError**(*i*)
Exception raised when trying to access a row that does not exist.

Parameters **i** – The index used to access the non-existent *Row*.

3.5.6 agate.rows

This module contains agate's *Row* implementation and various related classes. In common usage nothing in this module should need to be instantiated directly.

class agate.rows.**Row**(*table*, *i*)
Proxy to row data.

Values within a row can be accessed by column name or column index.

Parameters

- **table** – The Table that contains this row.
- **i** – The index of this row in the Table.

class agate.rows.**RowSequence**(*table*)
Proxy access to rows by index.

Parameters **table** – The *Table* that contains the rows.

class agate.rows.**RowIterator**(*table*)
Iterator over row proxies.

Parameters **table** – The *Table* of which to iterate.

class agate.rows.**CellIterator**(*row*)
Iterator over row cells.

Parameters **row** – The class:*Row* over which to iterate.

3.5.7 agate.table

This module contains the *Table* object, which is the central data structure in agate. Tables are created by supplying row data, column names and subclasses of *DataType* to the constructor. Once instantiated tables are **immutable**. This concept is central to agate. The table of the data may not be accessed or modified directly.

Various methods on the *Table* simulate "SQL-like" operations. For example, the *Table.select()* method reduces the table to only the specified columns. The *Table.where()* method reduces the table to only those rows that pass a truth test. And the *Table.order_by()* method sorts the rows in the table. In all of these cases the output is new *Table* and the existing table remains unmodified.

Tables are not themselves iterable, but the columns of the table can be accessed via *Table.columns* and the rows via *Table.rows*.

agate.table.**allow_tableset_proxy**(*func*)
Decorator to flag that a given Table method can be proxied as a TableSet method.

class agate.table.**Table**(*rows*, *column_info*)
A dataset consisting of rows and columns.

Parameters

- **rows** – The data as a sequence of any sequences: tuples, lists, etc.
- **column_info** – A sequence of pairs of column names and types. The latter must be instances of *DataType*.

Attr columns A *ColumnMapping* for accessing the columns in this table.

Attr rows A *RowSequence* for accessing the rows in this table.

classmethod from_csv(*path*, *column_info*, *header=True*, ***kwargs*)
Create a new table for a CSV. This method will use csvkit if it is available, otherwise it will use Python's builtin csv module.

kwargs will be passed through to csv.reader().

If you are using Python 2 and not using csvkit, this method is not unicode-safe.

Parameters

- **path** – Filepath or file-like object from which to read CSV data.
- **column_info** – A sequence of pairs of column names and types. The latter must be instances of *DataType*. Or, an instance of *TypeTester* to infer types.
- **header** – If *True*, the first row of the CSV is assumed to contains headers and will be skipped.

to_csv(*path*, ***kwargs*)
Write this table to a CSV. This method will use csvkit if it is available, otherwise it will use Python's builtin csv module.

kwargs will be passed through to csv.writer().

If you are using Python 2 and not using csvkit, this method is not unicode-safe.

Parameters **path** – Filepath or file-like object to write to.

column_types
Get an ordered list of this table's column types.

Returns A tuple of *Column* instances.

column_names
Get an ordered list of this table's column names.

Returns A tuple of strings.

rows
Get this tables *RowSequence*.

columns
Get this tables *ColumnMapping*.

data
Get the data underlying this table.

select(*column_names*)
Reduce this table to only the specified columns.

Parameters **column_names** – A sequence of names of columns to include in the new table.

Returns A new *Table*.

where(*test*)
Filter a to only those rows where the row passes a truth test.

Parameters **test** (function) – A function that takes a *Row* and returns True if it should be included.

Returns A new *Table*.

find(*test*)
Find the first row that passes a truth test.

Parameters **test** (function) – A function that takes a *Row* and returns True if it matches.

Returns A single *Row* or None if not found.

order_by(*key*, *reverse=False*)
Sort this table by the key. This can be either a column_name or callable that returns a value to sort by.

Parameters

- **key** – Either the name of a column to sort by or a function that takes a row and returns a value to sort by.
- **reverse** – If True then sort in reverse (typically, descending) order.

Returns A new *Table*.

limit(*start_or_stop=None*, *stop=None*, *step=None*)
Filter data to a subset of all rows.

See also: Python's slice().

Parameters

- **start_or_stop** – If the only argument, then how many rows to include, otherwise, the index of the first row to include.
- **stop** – The index of the last row to include.
- **step** – The size of the jump between rows to include. (*step=2* will return every other row.)

Returns A new *Table*.

distinct(*key=None*)
Filter data to only rows that are unique.

Parameters **key** – Either 1) the name of a column to use to identify unique rows or 2) a function that takes a row and returns a value to identify unique rows or 3) None, in which case the entire row will be checked for uniqueness.

Returns A new *Table*.

join(*right_table*, *left_key*, *right_key=None*, *inner=False*)

Performs the equivalent of SQL's "left outer join", combining columns from this table and from `right_table` anywhere that the output of `left_key` and `right_key` are equivalent.

Where there is no match for `left_key` the left columns will be included with the right columns set to `None` unless the `inner` argument is specified. (See arguments for more.)

If `left_key` and `right_key` are column names, only the left column will be included in the output table.

Column names from the right table which also exist in this table will be suffixed "2" in the new table.

Parameters

- **right_table** – The "right" table to join to.
- **left_key** – Either the name of a column from the this table to join on, or a `function` that takes a row and returns a value to join on.
- **right_key** – Either the name of a column from :code:table‘ to join on, or a `function` that takes a row and returns a value to join on. If `None` then `left_key` will be used for both.
- **inner** – Perform a SQL-style "inner join" instead of a left outer join. Rows which have no match for `left_key` will not be included in the output table.

Returns A new *Table*.

classmethod merge(*tables*)

Merge an array of tables with identical columns into a single table. Each table must have exactly the same column types. Their column names need not be identical. The first table's column names will be the ones which are used.

Parameters **tables** – An array of *Table*.

Returns A new *Table*.

group_by(*key*, *key_name=None*, *key_type=None*)

Create a new *Table* for unique value and return them as a *TableSet*. The `key` can be either a column name or a function that returns a value to group by.

Note that when group names will always be coerced to a string, regardless of the format of the input column.

Parameters

- **key** – Either the name of a column from the this table to group by, or a `function` that takes a row and returns a value to group by.
- **key_name** – A name that describes the grouped properties. Defaults to the column name that was grouped on or "group" if grouping with a key function. See *TableSet* for more.
- **key_type** – An instance some subclass of *DataType*. If not provided it will default to a :class‘.Text‘.

Returns A *TableSet* mapping where the keys are unique values from the `key` and the values are new *Table* instances containing the grouped rows.

compute(*computations*)

Compute new columns by applying one or more *Computation* to each row.

Parameters **computations** – An iterable of pairs of new column names and *Computation* instances.

Returns A new *Table*.

counts(*key*, *key_name=None*, *key_type=None*)
Count the number of occurrences of each distinct value in a column. Creates a new table with only the value and the count. This is effectively equivalent to doing a group_by followed by an aggregation with a *Length* aggregator.

Parameters

- **key** – Either the name of a column from the this table to count, or a `function` that takes a row and returns a value to count.
- **key_name** – A name that describes the counted properties. Defaults to the column name that was counted or "group" if counting with a key function.
- **key_type** – An instance some subclass of *DataType*. If not provided it will default to a :class'.Text'.

bins(*column_name*, *count=10*, *start=None*, *end=None*)
Generates (approximately) evenly sized bins for the values in a column. Bins may not be perfectly even if the spread of the data does not divide evenly, but all values will always be included in some bin.

Parameters

- **column_name** – The name of the column to bin. Must be of type *Number*
- **count** – The number of bins to create. If not specified then each value will be counted as its own bin.
- **start** – The minimum value to start the bins at. If not specified the minimum value in the column will be used.
- **end** – The maximum value to end the bins at. If not specified the maximum value in the column will be used.

Returns A new *Table* with *bin* (*Text*) and *count* (*Number*) columns.

print_table(*max_rows=None*, *max_columns=None*, *output=<open file '<stdout>', mode 'w'>*)
Print a well-formatted preview of this table to the console or any other output.

Parameters

- **max_rows** – The maximum number of rows to display before truncating the data.
- **max_columns** – The maximum number of columns to display before truncating the data.
- **output** – A file-like object to print to. Defaults to `sys.stdout`.

print_bars(*label_column_name*, *value_column_name*, *domain=None*, *width=120*, *output=<open file '<stdout>', mode 'w'>*)
Print a text-based bar chart of the columns names *label_column_name* and *value_column_name*.

Parameters

- **label_column_name** – The column containing the label values.
- **value_column_name** – The column containing the bar values.
- **domain** – A 2-tuple containing the minimum and maximum values for the chart's x-axis. The domain must be large enough to contain all values in the column.
- **width** – The width, in characters, to use for the bar chart. Defaults to *120*.
- **output** – A file-like object to print to. Defaults to `sys.stdout`.

monkeypatch(*patch_cls*)
Dynamically add `patch_cls` as a base class of this class.

Parameters **patch_cls** – The class to be patched on.

3.5.8 agate.tableset

This module contains the *TableSet* class which abstracts an set of related tables into a single data structure. The most common way of creating a *TableSet* is using the *Table.group_by()* method, which is similar to SQL's `GROUP BY` keyword. The resulting set of tables each have identical columns structure.

TableSet functions as a dictionary. Individual tables in the set can be accessed by using their name as a key. If the table set was created using *Table.group_by()* then the names of the tables will be the group factors found in the original data.

TableSet replicates the majority of the features of *Table*. When methods such as `TableSet.select()`, `TableSet.where()` or `TableSet.order_by()` are used, the operation is applied to *each* table in the set and the result is a new *TableSet* instance made up of entirely new *Table* instances.

TableSet instances can also contain other TableSet's. This means you can chain calls to *Table.group_by()* and `TableSet.group_by()` and end up with data grouped across multiple dimensions. *TableSet.aggregate()* on nested TableSets will then group across multiple dimensions.

class `agate.tableset.`**TableMethodProxy**(*tableset*, *method_name*)
A proxy for *TableSet* methods that converts them to individual calls on each *Table* in the set.

class `agate.tableset.`**TableSet**(*group*, *key_name='group'*, *key_type=None*)
An group of named tables with identical column definitions. Supports (almost) all the same operations as *Table*. When executed on a *TableSet*, any operation that would have returned a new *Table* instead returns a new *TableSet*. Any operation that would have returned a single value instead returns a dictionary of values.

Parameters

- **tables** – A dictionary of string keys and `Table` values.
- **key_name** – A name that describes the grouping properties. Used as the column header when the groups are aggregated. Defaults to the column name that was grouped on.
- **key_type** – An instance some subclass of *DataType*. If not provided it will default to a :class‘.Text‘.

key_name
Get the name of the key this TableSet is grouped by. (If created using *Table.group_by()* then this is the original column name.)

key_type
Get the *DataType* this TableSet is grouped by. (If created using *Table.group_by()* then this is the original column type.)

classmethod from_csv(*dir_path*, *column_info*, *header=True*, ***kwargs*)
Create a new *TableSet* from a directory of CSVs. This method will use csvkit if it is available, otherwise it will use Python's builtin csv module.

`kwargs` will be passed through to `csv.reader()`.

If you are using Python 2 and not using csvkit, this method is not unicode-safe.

Parameters

- **dir_path** – Path to a directory full of CSV files. All CSV files in this directory will be loaded.
- **column_info** – A sequence of pairs of column names and types. The latter must be instances of *DataType*. Or, an instance of *TypeTester* to infer types.
- **header** – If *True*, the first row of the CSV is assumed to contains headers and will be skipped.

to_csv(*dir_path*, ***kwargs*)
Write this each table in this set to a separate CSV in a given directory. This method will use csvkit if it is available, otherwise it will use Python's builtin csv module.

`kwargs` will be passed through to `csv.writer()`.

If you are using Python 2 and not using csvkit, this method is not unicode-safe.

Parameters **dir_path** – Path to the directory to write the CSV files to.

column_types
Get an ordered list of this *TableSet*'s column types.

Returns A `tuple` of *Column* instances.

column_names
Get an ordered list of this *TableSet*'s column names.

Returns A `tuple` of strings.

merge()
Convert this TableSet into a single table. This is the inverse of *Table.group_by()*.

Returns A new `Table`.

aggregate(*aggregations=[]*)
Aggregate data from the tables in this set by performing some set of column operations on the groups and coalescing the results into a new *Table*.

`aggregations` must be a list of tuples, where each has three parts: a `column_name`, a *Aggregation* instance and a `new_column_name`.

Parameters **aggregations** – An list of triples in the format `(column_name, aggregation, new_column_name)`.

Returns A new *Table*.

get($k\left[, d\right]$) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

monkeypatch(*patch_cls*)
Dynamically add `patch_cls` as a base class of this class.

Parameters **patch_cls** – The class to be patched on.

values() → list of D's values

3.5.9 agate.utils

This module contains a collection of utility classes and functions used in agate.

class `agate.utils.`**Patchable**
Adds a monkeypatching extensibility pattern to subclasses.

Calling `Class.monkeypatch(AnotherClass)` will dynamically add `AnotherClass` as a base class of `Class`. This effective is global–even existing instances of the class will have the new methods.

This can only be used to add new methods. It can not be used to override the implementation of an existing method on the patched class.

classmethod monkeypatch(*patch_cls*)
Dynamically add `patch_cls` as a base class of this class.

Parameters **patch_cls** – The class to be patched on.

class agate.utils.**NullOrder**
Dummy object used for sorting in place of None.

Sorts as "greater than everything but other nulls."

class agate.utils.**Quantiles**(*quantiles*)
A class representing quantiles (percentiles, quartiles, etc.) for a given column of Number data.

locate(*value*)
Identify which quantile a given value is part of.

agate.utils.**memoize**(*func*)
Dead-simple memoize decorator for instance methods that take no arguments.

This is especially useful since so many of our classes are immutable.

agate.utils.**median**(*data_sorted*)
Finds the median value of a given series of values.

Parameters **data_sorted** – The values to find the median of. Must be sorted.

agate.utils.**max_precision**(*values*)
Given a series of values (such as a *Column*) returns the most significant decimal places present in any value.

Parameters **values** – The values to analyze.

agate.utils.**make_number_formatter**(*decimal_places*)
Given a number of decimal places creates a formatting string that will display numbers with that precision.

agate.utils.**round_to_magnitude**(*n*)
Round a value to the nearest whole magnitude.

3.6 Contributing to agate

3.6.1 Principles

agate is a intended to fill a very particular programming niche. It should not be allowed to become as complex as numpy or pandas. Please bear in mind the following principles when contemplating an addition:

- Humans have less time than computers. Always optimize for humans.
- Most datasets are small and simple. Never optimize for "big data".
- Text is data. It must always be a first-class citizen.
- Python gets it right. Make it work like Python does.
- Humans lives are nasty, brutish and short. Make things easy.
- Mutability leads to confusion. Processes that alter data must create new copies.

3.6.2 How agate works

Here are a few notes regarding agate's internal architecture for interested developers. Users shouldn't need to care about these things.

- Methods on *Table* instances such as *Table.select()*, *Table.where()* and *Table.order_by()* return new *Table* instances.
- Operations on *Column* instances access data from their parent *Table* and return appropriate native data structures. Columns are not portable between instances of *Table*!
- When a *Table* is instantiated the data provided is copied into a tuple so that it becomes immutable within the context of the *Table*.
- *Table* instances that are "forked" (created by table operations) will share *Row* instances for memory efficiency. This is safe because row data is immutable. Methods that create new data will copy the row data first. (e.g. *Table.compute()*)
- *ColumnMapping*, *RowSequence*, *Column*, and *Row* have **read only** access to a Table's private variables. They are purely a formal abstraction and for purposes of encapsulation they can be treated as a single unit.
- *Column* instances lazily construct a copy of their data from their parent Table and then cache it. They will also cache the result of common operations such as filtering null values. This caching is safe because the underlying data is immutable.

3.6.3 Process for contributing code

Contributors should use the following roadmap to guide them through the process of submitting a contribution:

1. Fork the project on Github.
2. Check out the issue tracker and find a task that needs to be done and is of a scope you can realistically expect to complete in a few days. Don't worry about the priority of the issues at first, but try to choose something you'll enjoy. You're much more likely to finish something to the point it can be merged if it's something you really enjoy hacking on.
3. Comment on the ticket letting everyone know you're going to be hacking on it so that nobody duplicates your effort. It's also good practice to provide some general idea of how you plan on resolving the issue so that other developers can make suggestions.
4. Write tests for the feature you're building. Follow the format of the existing tests in the test directory to see how this works. You can run all the tests with the command `nosetests`. (Or `tox` to run across all versions of Python.)
5. Write the code. Try to stay consistent with the style and organization of the existing codebase. A good patch won't be refused for stylistic reasons, but large parts of it may be rewritten and nobody wants that.
6. As you are coding, periodically merge in work from the master branch and verify you haven't broken anything by running the test suite.
7. Write documentation. Seriously.
8. Once it works, is tested, and has documentation, submit a pull request on Github.
9. Wait for it to either be merged or to receive a comment about what needs to be fixed.
10. Rejoice.

3.6.4 Legalese

To the extent that they care, contributors should keep in mind that the source of agate and therefore of any contributions are licensed under the permissive MIT license. By submitting a patch or pull request you are agreeing to release your code under this license. You will be acknowledged in the AUTHORS file. As the owner of your specific contributions you retain the right to privately relicense your specific code contributions, however, the released version of the code can never be retracted.

3.7 Release process

This is the release process for agate:

1. Verify all unit tests pass with fresh environments: `tox -r`.
2. Verify 100% test coverage: `nosetests --with-coverage --cover-package=agate`.
3. Ensure any new modules have been added to setup.py's `packages` list.
4. Make sure the example script still works: `python example.py`.
5. Ensure `CHANGELOG` is up to date.
6. Create a release tag: `git tag -a x.y.z -m "x.y.z release."`
7. Push tags upstream: `git push --tags`
8. Upload to PyPI: `python setup.py sdist bdist_wheel upload`.
9. Flag the release to build on RTFD.
10. Update the "default version" on RTFD to the latest.
11. Rev to latest version: `docs/conf.py`, `setup.py` and `CHANGELOG` need updates.
12. Commit revision: `git commit -am "Update to version x.y.z for development."`.

CHAPTER 4

Authors

The following individuals have contributed code to agate:

- Christopher Groskopf
- Jeff Larson
- Eric Sagara
- John Heasly
- Mick O'Brien
- Nikhil Sonnad
- Matt Riggott

CHAPTER 5

License

The MIT License

Copyright (c) 2014 Christopher Groskopf and contributers

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.

CHAPTER 6

Changelog

6.1 0.11.0

- Column constructor no longer takes a data_type argument.
- Column.index and Column.name properties added.
- Table.counts implemented. (#271)
- Table.bins implemented. (#267, #227)
- Table.join now raises ColumnDoesNotExistError. (#264)
- Table.select now raises ColumnDoesNotExistError.
- computations.ZScores moved into agate-stats.
- computations.Rank cmp argument renamed comparer.
- aggregations.MaxPrecision added. (#265)
- Table.print_bars added.
- Table.pretty_print renamed Table.print_table.
- Reimplement Table method proxying via @allow_tableset_proxy decorator. (#263)
- Add agate-stats references to docs.
- Move stdev_outliers, mad_outliers and pearson_correlation into agate-stats. (#260)
- Prevent issues with applying patches multiple times. (#258)

6.2 0.10.0

- Add reverse and cmp arguments to Rank computation. (#248)
- Document how to use agate-sql to read/write SQL tables. (#238, #241)
- Document how to write extensions.
- Add monkeypatching extensibility pattern via utils.Patchable.
- Reversed order of argument pairs for Table.compute. (#249)
- TableSet.merge method can be used to ungroup data. (#253)
- Columns with identical names are now suffixed “2” after a Table.join.

- Duplicate key columns are no longer included in the result of a Table.join. (#250)
- Table.join right_key no longer necessary if identical to left_key. (#254)
- Table.inner_join is now more. Use *inner* keyword to Table.join.
- Table.left_outer_join is now Table.join.

6.3 0.9.0

- Add many missing unit tests. Up to 99% coverage.
- Add property accessors for TableSet.key_name and TableSet.key_type. (#247)
- Table.rows and Table.columns are now behind properties. (#247)
- Column.data_type is now a property. (#247)
- Table[Set].get_column_types() is now the Table[Set].column_types property. (#247)
- Table[Set].get_column_names() is now the Table[Set].column_names property. (#247)
- Table.pretty_print now displays consistent decimal places for each Number column.
- Discrete data types (Number, Date etc) are now right-aligned in Table.pretty_print.
- Implement aggregation result caching. (#245)
- Reimplement Percentiles, Quartiles, etc as aggregations.
- UnsupportedAggregationError is now used to disable TableSet aggregations.
- Replaced several exceptions with more general DataTypeError.
- Column type information can now be accessed as Column.data_type.
- Eliminated Column subclasses. Restructured around DataType classes.
- Table.merge implemented. (#9)
- Cookbook: guess column types. (#230)
- Fix issue where all group keys were being cast to text. (#235)
- Table.group_by will now default key_type to the type of the grouping column. (#234)
- Add Matt Riggott to AUTHORS. (#231)
- Support file-like objects in Table.to_csv and Table.from_csv. (#229)
- Fix bug when applying multiple computations with Table.compute.

6.4 0.8.0

- Cookbook: dealing with locales. (#220)
- Cookbook: working with dates and times.
- Add timezone support to DateTimeType.
- Use pytimeparse instead of python-dateutil. (#221)
- Handle percents and currency symbols when casting numbers. (#217)
- Table.format is now Table.pretty_print. (#223)

- Rename TextType to Text, NumberType to Number, etc.
- Rename agate.ColumnType to agate.DataType (#216)
- Rename agate.column_types to agate.data_types.
- Implement locale support for number parsing. (#116)
- Cookbook: ranking. (#110)
- Cookbook: date change and date ranking. (#113)
- Add tests for unicode support. (#138)
- Fix computations.ZScores calculation. (#123)
- Differentiate sample and population variance and stdev. (#208)
- Support for overriding column inference with "force".
- Competition ranking implemented as default. (#125)
- TypeTester: robust type inference. (#210)

6.5 0.7.0

- Cookbook: USA Today diversity index.
- Cookbook: filter to top x%. (#47)
- Cookbook: fuzzy string search example. (#176)
- Values to coerce to true/false can now be overridden for BooleanType.
- Values to coerce to null can now be overridden for all ColumnType subclasses. (#206)
- Add key_type argument to TableSet and Table.group_by. (#205)
- Nested TableSet's and multi-dimensional aggregates. (#204)
- TableSet.aggregate will now use key_name as the group column name. (#203)
- Added key_name argument to TableSet and Table.group_by.
- Added Length aggregation and removed count from TableSet.aggregate output. (#203)
- Fix error messages for RowDoesNotExistError and ColumnDoesNotExistError.

6.6 0.6.0

- Fix missing package definition in setup.py.
- Split Analysis off into the proof library.
- Change computation now works with DateType, DateTimeType and TimeDeltaType. (#159)
- TimeDeltaType and TimeDeltaColumn implemented.
- NonNullAggregation class removed.
- Some private Column methods made public. (#183)
- Rename agate.aggegators to agate.aggregations.
- TableSet.to_csv implemented. (#195)

- TableSet.from_csv implemented. (#194)
- Table.to_csv implemented (#169)
- Table.from_csv implemented. (#168)
- Added Table.format method for pretty-printing tables. (#191)
- Analysis class now implements a caching workflow. (#171)

6.7 0.5.0

- Table now takes (column_name, column_type) pairs. (#180)
- Renamed the library to agate. (#179)
- Results of common column operations are now cached using a common memoize decorator. (#162)
- Deprecated support for Python version 3.2.
- Added support for Python wheel packaging. (#127)
- Add PercentileRank computation and usage example to cookbook. (#152)
- Add indexed change example to cookbook. (#151)
- Add annual change example to cookbook. (#150)
- Column.aggregate now invokes Aggregations.
- Column.any, NumberColumn.sum, etc. converted to Aggregations.
- Implement Aggregation and subclasses. (#155)
- Move ColumnType subclasses and ColumnOperation subclasses into new modules.
- Table.percent_change, Table.rank and Table.zscores reimplemented as Computers.
- Computer implemented. Table.compute reimplemented. (#147)
- NumberColumn.iqr (inter-quartile range) implemented. (#102)
- Remove Column.counts as it is not the best way.
- Implement ColumnOperation and subclasses.
- Table.aggregate migrated to TableSet.aggregate.
- Table.group_by now supports grouping by a key function. (#140)
- NumberColumn.deciles implemented.
- NumberColumn.quintiles implemented. (#46)
- NumberColumn.quartiles implemented. (#45)
- Added robust test case for NumberColumn.percentiles. (#129)
- NumberColumn.percentiles reimplemented using new method. (#130)
- Reorganized and modularized column implementations.
- Table.group_by now returns a TableSet.
- Implement TableSet object. (#141)

6.8 0.4.0

- Upgrade to python-dateutil 2.2. (#134)
- Wrote introductory tutorial. (#133)
- Reorganize documentation (#132)
- Add John Heasly to AUTHORS.
- Implement percentile. (#35)
- no_null_computations now accepts args. (#122)
- Table.z_scores implemented. (#123)
- DateTimeColumn implemented. (#23)
- Column.counts now returns dict instead of Table. (#109)
- ColumnType.create_column renamed _create_column. (#118)
- Added Mick O'Brien to AUTHORS. (#121)
- Pearson correlation implemented. (#103)

6.9 0.3.0

- DateType.date_format implemented. (#112)
- Create ColumnType classes to simplify data parsing.
- DateColumn implemented. (#7)
- Cookbook: Excel pivot tables. (#41)
- Cookbook: statistics, including outlier detection. (#82)
- Cookbook: emulating Underscore's any and all. (#107)
- Parameter documention for method parameters. (#108)
- Table.rank now accepts a column name or key function.
- Optionally use cdecimal for improved performance. (#106)
- Smart naming of aggregate columns.
- Duplicate columns names are now an error. (#92)
- BooleanColumn implemented. (#6)
- TextColumn.max_length implemented. (#95)
- Table.find implemented. (#14)
- Better error handling in Table.__init__. (#38)
- Collapse IntColumn and FloatColumn into NumberColumn. (#64)
- Table.mad_outliers implemented. (#93)
- Column.mad implemented. (#93)
- Table.stdev_outliers implemented. (#86)
- Table.group_by implemented. (#3)

- Cookbook: emulating R. (#81)
- Table.left_outer_join now accepts column names or key functions. (#80)
- Table.inner_join now accepts column names or key functions. (#80)
- Table.distinct now accepts a column name or key function. (#80)
- Table.order_by now accepts a column name or key function. (#80)
- Table.rank implemented. (#15)
- Reached 100% test coverage. (#76)
- Tests for Column._cast methods. (#20)
- Table.distinct implemented. (#83)
- Use assertSequenceEqual in tests. (#84)
- Docs: features section. (#87)
- Cookbook: emulating SQL. (#79)
- Table.left_outer_join implemented. (#11)
- Table.inner_join implemented. (#11)

6.10 0.2.0

- Python 3.2, 3.3 and 3.4 support. (#52)
- Documented supported platforms.
- Cookbook: csvkit. (#36)
- Cookbook: glob syntax. (#28)
- Cookbook: filter to values in range. (#30)
- RowDoesNotExistError implemented. (#70)
- ColumnDoesNotExistError implemented. (#71)
- Cookbook: percent change. (#67)
- Cookbook: sampleing. (#59)
- Cookbook: random sort order. (#68)
- Eliminate Table.get_data.
- Use tuples everywhere. (#66)
- Fixes for Python 2.6 compatibility. (#53)
- Cookbook: multi-column sorting. (#13)
- Cookbook: simple sorting.
- Destructive Table ops now deepcopy row data. (#63)
- Non-destructive Table ops now share row data. (#63)
- Table.sort_by now accepts a function. (#65)
- Cookbook: pygal.

- Cookbook: Matplotlib.
- Cookbook: VLOOKUP. (#40)
- Cookbook: Excel formulas. (#44)
- Cookbook: Rounding to two decimal places. (#49)
- Better repr for Column and Row. (#56)
- Cookbook: Filter by regex. (#27)
- Cookbook: Underscore filter & reject. (#57)
- Table.limit implemented. (#58)
- Cookbook: writing a CSV. (#51)
- Kill Table.filter and Table.reject. (#55)
- Column.map removed. (#43)
- Column instance & data caching implemented. (#42)
- Table.select implemented. (#32)
- Eliminate repeated column index lookups. (#25)
- Precise DecimalColumn tests.
- Use Decimal type everywhere internally.
- FloatColumn converted to DecimalColumn. (#17)
- Added Eric Sagara to AUTHORS. (#48)
- NumberColumn.variance implemented. (#1)
- Cookbook: loading a CSV. (#37)
- Table.percent_change implemented. (#16)
- Table.compute implemented. (#31)
- Table.filter and Table.reject now take funcs. (#24)
- Column.count implemented. (#12)
- Column.counts implemented. (#8)
- Column.all implemented. (#5)
- Column.any implemented. (#4)
- Added Jeff Larson to AUTHORS. (#18)
- NumberColumn.mode implmented. (#18)

6.11 0.1.0

- Initial prototype

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

a

A

B

C

D

F

G

H

I

J

K

U

V

W