Pyrtable

Release 0.7.14

Jul 31, 2021

Contents:

CHAPTER 1

Introduction

1.1 Installation

```
$ pip install pyrtable
```

If you want to add support for timezone-aware timestamps (highly recommended):

```
$ pip install 'pyrtable[pytz]'
```

1.2 Quick Start

Note: Notice that this tutorial will not work out-of-the-box, as you would need a corresponding Airtable table having columns with same names and value types as described below. However, you can adapt the examples below with your own existing bases or create one to experiment with Pyrtable.

To use Pyrtable you will first need to subclass the `BaseRecord` class. Objects of your subclass represent records on the corresponding Airtable table, while the subclass itself is used to interact with the table itself (mostly to fetch records). See the examples below:

```
import enum
from pyrtable.record import BaseRecord
from pyrtable.fields import StringField, DateField, SingleSelectionField, \
        SingleRecordLinkField, MultipleRecordLinkField

class Role(enum.Enum):
    DEVELOPER = 'Developer'
    MANAGER = 'Manager'
    CEO = 'C.E.O.'
```

(continues on next page)

(continued from previous page)

```
class EmployeeRecord(BaseRecord):
    class Meta:
        base_id = 'appABCDE12345'  # @TODO change this value
        table_id = 'Employees'     # @TODO change this value

    @classmethod
    def get_api_key(cls):
        return 'keyABCDE12345'     # @TODO change this value

    name = StringField('Name')
    birth_date = DateField('Birth date')
    office = SingleRecordLinkField('Office', linked_class='OfficeRecord')
    projects = MultipleRecordLinkField(
            'Allocated in projects', linked_class='ProjectRecord')
    role = SingleSelectionField('Role', choices=Role)
```

Further information about the structure of `BaseRecord` subclasses (and how to fill these "`@TODO`" values) can be seen in *how to define record classes*. The reference for the field classes are available *here*.

At this point, retrieving records from Airtable is quite easy:

```
for employee in EmployeeRecord.objects.all():
    print('Employee %s is working on %d projects' %
          (employee.name, len(employee.projects)))
    if employee.role == Role.DEVELOPER:
        print('S/he may understand the difference between loops and conditionals!')
```

Creating, updating and deleting records are also easy:

```
# Creating a record
new_employee = EmployeeRecord(
        name='John Doe',
        birth_date=datetime.date(1980, 5, 10),
        role=Role.DEVELOPER)
new_employee.save()

# Updating a record
new_employee.role = Role.MANAGER
new_employee.save()

# Deleting a record
new_employee.delete()
```

CHAPTER 2

Defining record classes

To use Pyrtable you will first need to subclass the `BaseRecord` class, once for each table you need to access. Objects of these subclasses represent records on the corresponding Airtable table, while the subclasses themselves are used to interact with the server (mostly to fetch records).

Each one of these subclasses follow the structure below:

```
class MyTableRecord(BaseRecord):
    class Meta:
        base_id = '<BASE ID>'
        table_id = '<TABLE ID>'

    # @TODO You need to provide credentials (the API Key) somehow

    # Fields definitions are created by instantiating *Field classes, such as:
    #name = StringField('Name')
    #birth_date = DateField('Birth date')
    #office = SingleRecordLinkField('Office', linked_class='OfficeRecord')
```

Details about the missing bits are provided below.

2.1 Values for `base_id` and `table_id`

2.1.1 `base_id`

Open the desired base in Airtable, go to "Help > API documentation" (top-right corner) and search for a paragraph containing "The ID of this base is *base_id*".

2.1.2 `table_id`

The `table_id` is the name of the table itself. You can double-click the table name in the top tables to ease copying to the clipboard. Avoid using extraneous characters in the name, as these may render Pyrtable inoperative (accented

characters, spaces, dots, hyphens, underlines are all OK).

2.2 Fields definitions

Refer to *this page* for details about using field classes to define the properties that link to Airtable fields.

2.3 Authentication methods

To actually interact with the Airtable server, Pyrtable needs to know the *API Key*. Airtable has a support page explaining how to obtain this key.

2.3.1 Providing the API Key to the record class

The basic way to provide the API Key to Pyrtable is to implement a class method that returns the key:

```
class MyTableRecord(BaseRecord):
    @classmethod
    def get_api_key(cls):
        return '<API KEY>'

    # other class stuff here
```

If this class method accepts a *base_id* parameter, then the caller will fill it – this may be used, e.g., for a dictionary-based lookup:

```
class MyTableRecord(BaseRecord):
    @classmethod
    def get_api_key(cls, base_id):
        return {
            '<BASE_ID_1>': '<API KEY_1>',
            '<BASE_ID_2>': '<API KEY_2>',
        }[base_id]

    # other class stuff here
```

Warning: Putting the raw API Key in the source code itself is a *bad security practice*, as anyone with access to your code will have **full R/W access to all your Airtable bases**. API Keys are as sensitive as passwords; they should be securely stored in separate, private files or using OS keychain services. See the `APIKeyFromSecretsFileMixin` below.

2.3.2 Reading the API Key from a file

Note: This method requires the PyYAML package installed.

Using this approach is surprisingly easy. You only need to add the `APIKeyFromSecretsFileMixin` mixin when defining the class:

```
class MyTableRecord(APIKeyFromSecretsFileMixin, BaseRecord):
    class Meta:
        base_id = '<BASE ID>'
        table_id = '<TABLE ID>'

    # Fields definitions go here
```

Pyrtable will then search for a file named `airtable_secrets.yaml` in one of the following directories:

- `./config` subdirectory (under the current directory), or
- `/etc/airtable`

This file is a YAML file with one of more key-value pairs, where each key is a base ID and the corresponding value is the API Key used to access that base. At the end, the file will contain one or more lines as follows:

```
appFGHIJ67890fghij: keyABCDE12345abcde
```

2.3.3 Reading the API Key from an environment variable

This is an alternative to using `APIKeyFromSecretsFileMixin` and particularly useful for running Docker containers where all bases are accessible under the same API Key:

```
class MyTableRecord(BaseRecord):
    class Meta:
        base_id = '<BASE ID>'
        table_id = '<TABLE ID>'

    @classmethod
    def get_api_key(cls):
        return os.getenv('AIRTABLE_API_KEY')

    # Fields definitions go here
```

Now just provide the API Key through the `AIRTABLE_API_KEY` environment variable, e.g., using the corresponding Docker command-line option or the corresponding Docker Compose configuration key.

2.4 Don't Repeat Yourself!

In the most common scenario, a Python project will interact with several tables across a single Airtable base. That means that `base_id` value will be the same for all `BaseRecord` subclasses.

To avoid unnecessary code repetition, you can create a superclass for all record classes of the same base. This superclass will only contain the definition of `base_id` and the selected authentication method. See the example:

```
class MyBaseRecord(APIKeyFromSecretsFileMixin, BaseRecord):
    class Meta:
        base_id = '<BASE ID>'


class MyTableRecord(MyBaseRecord):
    class Meta:
        table_id = '<TABLE ID>'
```

(continues on next page)

(continued from previous page)

```
    # Fields definitions go here


class MyOtherTableRecord(MyBaseRecord):
    class Meta:
        table_id = '<OTHER TABLE ID>'

    # Fields definitions go here
```

Notice that `table_id` is specific to the actual record classes, while `base_id` is common for all of them.

Of course this superclass can also be designed to read the API Key from an environment variable:

```
class MyBaseRecord(BaseRecord):
    class Meta:
        base_id = '<BASE ID>'

    @classmethod
    def get_api_key(cls):
        return os.getenv('AIRTABLE_API_KEY')
```

CHAPTER 3

Field classes

Field classes are used to declare properties that map onto Airtable fields (columns). You instantiate field classes when *defining your record classes*, providing the corresponding Airtable field name as the first argument – see the example below:

```
class PersonRecord(BaseRecord):
    name = StringField('Name', read_only=True)
    birth_date = DateField('Birth date')
    number_of_children = IntegerField('Children')
```

Airtable always allows empty values to any cell. In general, these are represented as `None` in Python, with exceptions noted below (for instance, `StringField` will always map empty cells onto empty strings).

You don't need to map all Airtable fields; it's OK to declare only some of the fields in Python.

Neither `id` nor `created_timestamp` can be used as field names as they are reserved for two special fields described below.

3.1 The `id` field

All record classes have an automatically generated `.id` field. It will hold either the Airtable record identifier (a string) or, for deleted and created-but-not-saved records, the `None` value.

3.2 The `created_timestamp` field

All record classes have an automatically generated `.created_timestamp` field. It will hold the record creation timestamp as informed by Airtable, or `None` if it was not yet saved.

3.3 Field arguments

Some arguments are used in all field types. These are documented below.

3.3.1 field_name

A string. The name of the field as defined in Airtable.

Beware of field names — not all characters are supported by Pyrtable, even if they are accepted in Airtable. Currently only letters (including diacritics), numbers, spaces, dots, hyphens, underlines are accepted.

3.3.2 read_only

A boolean (optional, defaults to `False`). If `True`, changes to that field are forbidden in Python. You can use this to guarantee that Pyrtable will never update the corresponding Airtable field. Read-only fields are still writeable when creating records.

3.4 Field types

3.4.1 AttachmentField

class AttachmentField(field_name, read_only=True, **options)

Note: Currently only reading operations are implemented, so using `read_only=True` is mandatory.

Holds a collection of uploaded files. Each uploaded file is represented by an instance of the `Attachment` class that contains the following properties:

- `id` (`str`): Airtable unique identifier of this attachment;
- `url` (`str`): URL that can be used to download the file;
- `filename` (`str`): Name of the uploaded file;
- `size` (`int`): Size of the file, in bytes;
- `type` (`str`): Mimetype of the file;
- `width` (`int`, optional): If the file is an image, its width in pixels;
- `height` (`int`, optional): If the file is an image, its height in pixels;
- `thumbnails`: If the file is an image, this is an object with three properties: `small`, `large` and `full`. Each one of these properties point to an object that in turn has three properties: `url`, `width` and `height`. One can use these properties to access thumbnails for the uploaded image.

The `Attachment` class also has two methods to download the corresponding file:

- `download()`: downloads the file and returns the in-memory representation as a `bytes` instance;
- `download_to(path)`: downloads the file and and stores it as a local file whose path is given by the `path` argument.

This property follows `collections.abc.Sized` and `collections.abc.Iterable` semantics, so the following operations are allowed:

```
class PersonRecord(BaseRecord):
    profile_pictures = AttachmentField('Images', read_only=True)

person = # ...fetch a PersonRecord from the server

# Counting the number of attached images
print(len(person.profile_pictures))

# Iterating over attached images
for picture in person.profile_pictures:
    if picture.width is not None and picture.height is not None:
        print('There is a %dx%d image' % (image.width, image.height))
```

3.4.2 BooleanField

class BooleanField(field_name, **options)

Holds a `bool` value. This field never holds `None`, as empty values are mapped to `False`.

3.4.3 DateField

class DateField(field_name, **options)

Holds a `datetime.date` value.

3.4.4 DateTimeField

class DateTimeField(field_name, **options)

Holds a `datetime.datetime` value. If the pytz package is installed, values will be timezone aware.

3.4.5 FloatField

class FloatField(field_name, **options)

Holds a `float` value.

3.4.6 IntegerField

class IntegerField(field_name, **options)

Holds an `int` value.

3.4.7 MultipleRecordLinkField

class MultipleRecordLinkField(field_name, linked_class, **options)

Holds zero or more record references, possibly from another Airtable table. `linked_class` is either the record class (i.e., a `BaseRecord` subclass) or a string containing full Python module path to that class (e.g., `'mypackage.mymodule.MyTableRecord'`).

This property follows `collections.abc.Iterable` and `collections.abc.MutableSet` semantics, so the following operations are allowed:

```
class EmployeeRecord(BaseRecord):
    projects = MultipleRecordLinkField('Projects', linked_class=ProjectRecord)

employee = # ...fetch an EmployeeRecord from the server

# Counting the number of linked records
print(len(employee.projects))

# Checking if a value is/isn't selected
if revolutionary_project in employee.projects:
    print('Congratulations, you have worked in our best project!')
if flopped_project not in employee.projects:
    print('You are not to be blamed. This time.')

# Iterating over selected values
for project in employee.projects:
    print('Our employee %s is working on the project %s' %
          (employee.name, project.name))
```

To change the value of this property there are some ways:

```
employee.projects.add(project)
employee.projects.discard(project)
employee.projects.set(iterable_projects)
```

Notice that the last method accepts an iterable, such as lists, tuples, and sets. There are also some shortcuts:

```
employee.projects += project
employee.projects -= project
```

Pyrtable also creates a companion property with `'_ids'` suffix that holds a collection record IDs. So, in the example above the record IDs can be printed as follows:

```
print('Linked record IDs: %s' % ', '.join(employee.record_ids))
```

Accessing a `MultipleRecordLinkField` property at runtime is an expensive operation for the first time, as it requires fetching each set of linked records from the Airtable server. Once the records are fetched they are cached in memory, so subsequent accesses are fast. Pyrtable also provides mechanisms to *cache foreign records in advance*.

3.4.8 MultipleSelectionField

class MultipleSelectionField(field_name, choices=None, **options)

Holds zero or more values from a predefined set (Airtable calls it a "Multiple select" field) that is mapped onto a Python enum (a subclass of `enum.Enum`). The enum class is given as a second argument named `choices` — check `SingleSelectionField` for a detailed description and examples.

If `choices` is not given or is `None`, the field maps values into strings.

> **Warning:** Due to limitations of the Airtable API, it is currently not possible to use commas in any of the options for multiple select fields. This may confuse Pyrtable in some operations and may cause data loss!

This property follows `collections.abc.Iterable` and `collections.abc.MutableSet` semantics, so the following operations are allowed:

```
# Counting the number of values selected
print(len(record.multiple_selection_field))

# Checking if a value is/isn't selected
if value in record.multiple_selection_field:
    print('The value %r is currently selected.' % value)
if value not in record.multiple_selection_field:
    print('The value %r currently not selected.' % value)

# Iterating over selected values
for value in record.multiple_selection_field:
    print('Selected value: %r' % value)
```

To change the value of this property there are some ways:

```
record.multiple_selection_field.add(value)
record.multiple_selection_field.discard(value)
record.multiple_selection_field.set(iterable)
```

Notice that the last method accepts an iterable, such as lists, tuples, and sets. There are also some shortcuts:

```
record.multiple_selection_field += value
record.multiple_selection_field -= value
```

3.4.9 SingleRecordLinkField

class SingleRecordLinkField(field_name, linked_class, **options)

Holds a reference to another record, possibly from another Airtable table. `linked_class` is either the record class (i.e., a `BaseRecord` subclass) or a string containing full Python module path to that class (e.g., `'mypackage.mymodule.MyTableRecord'`).

Pyrtable also creates a companion property with `'_ids'` suffix that holds a reference to the record ID. So, for example:

```
class EmployeeRecord(BaseRecord):
    office = SingleRecordLinkField('Office',
                                   linked_class='OfficeRecord')
```

then all objects of `EmployeeRecord` class will also have a `obj.office_id` that holds the ID of the office record. Accessing this property does not hit the Airtable field.

Accessing a `SingleRecordLinkField` property at runtime is an expensive operation for the first time, as it requires fetching each linked record from the Airtable server. Once the linked record is fetched it is cached in memory, so subsequent accesses are fast. Pyrtable also provides mechanisms to *cache foreign records in advance*.

3.4.10 SingleSelectionField

class SingleSelectionField(field_name, choices, **options)

Holds a single value from a predefined set (Airtable calls it a "Single select" field) that is mapped onto a Python enum (a subclass of `enum.Enum`). The enum class is given as a second argument named `choices` — see below:

```
class Role(enum.Enum):
    DEVELOPER = 'Developer'
    MANAGER = 'Manager'
    CEO = 'C.E.O.'

class EmployeeRecord(BaseRecord):
    role = SingleSelectionField('Role', choices=Role)
```

3.4.11 StringField

class StringField(field_name, **options)

Holds a `str` value. Unlike other field types, this field never holds `None`; nonexistent values are always translated into empty strings.

CHAPTER 4

Retrieving and saving records

Once you have *created your Pyrtable classes*, it's now time to actually communicate with the server.

There are three ways to fetch records: you can get *all records of a table*, *only those that meet a given criteria*, or *a single record from a record ID*.

You can also update the server by *updating records*, *creating records* and *deleting records*.

From now on, the record class will be referred as `MyTableRecord`.

4.1 Retrieving all records

`MyTableRecord.objects.all()` can be used to traverse all records of the corresponding Airtable table:

```
# Assuming that MyTableRecord has a `name` field
for record in MyTableRecord.objects.all():
    print('Record with ID %s has a name %s.' % (record.id, record.name))
```

Notice that `MyTableRecord.objects.all()` will give an *iterator*, not a list of records itself. This means that calling this method will not hit the server — that will happen every time you iterate over that. In other words:

```
# This will not yet fetch records with the server
all_records_query = MyTableRecord.objects.all()

# This loop will fetch data from the server
for record in all_records_query:
    print('Record with ID %s has a name %s.' % (record.id, record.name))

# This will fetch data all over again
for record in all_records_query:
    print('Person named %s has %d years old' % (record.name, record.age))
```

If you want to fetch data only once, you need to make a list out of the iterator right on the beginning:

```
# This will hit the server and fetch all records
all_records = list(MyTableRecord.objects.all())

# From now on, `all_records` holds all records -
# iterating over it will not fetch data from the server.
```

4.2 Retrieving some records

If you want to fetch only records that match given criteria, you can use `MyTableRecord.objects.filter()`. It's also an iterator, so fetching will not happen until you actually iterate over elements:

```
# Filter by equality
query = MyTableRecord.objects.filter(first_name='John')
query = MyTableRecord.objects.filter(age=30)
query = MyTableRecord.objects.filter(is_admin=True)
query = MyTableRecord.objects.filter(role=Role.MANAGER)

# Filter MultipleSelectionField fields
query = MyTableRecord.objects.filter(role__contains=(Role.DEVELOPER, Role.MANAGER))
query = MyTableRecord.objects.filter(role__excludes=(Role.DEVELOPER, Role.MANAGER))

# Filter by inequality:
# - "not equals"
query = MyTableRecord.objects.filter(first_name_ne='John')
# - "greater than"
query = MyTableRecord.objects.filter(age__gt=30)
# - "greater than or equals"
query = MyTableRecord.objects.filter(age__gte=30)
# - "less than"
query = MyTableRecord.objects.filter(age__lt=30)
# - "less than or equals"
query = MyTableRecord.objects.filter(age__lte=30)
# - "is empty"
query = MyTableRecord.objects.filter(age__empty=True)

# Multiple criteria can be specified - they are ANDed together
query = MyTableRecord.objects.filter(
        first_name='John', last_name='Doe', age__gt=30)
```

Filters can be further narrowed before iteration, so the following pattern is perfectly valid:

```
def get_admins(managers_only=False):
    query = MyTableRecord.objects.filter(is_admin=True)
    if managers_only:
        query = query.filter(role=Role.MANAGER)

    # Server will be queried here
    return list(query)
```

Actually `MyTableRecord.objects.all()` also has a `.filter()` method, so you can start with "all" (meaning "no filters") and narrow them down before hitting the server:

```
def get_employees(admin_only=False, managers_only=False):
    query = MyTableRecord.objects.all()
```

(continues on next page)

(continued from previous page)

```
    if admin_only:
        query = query.filter(is_admin=True)
    if managers_only:
        query = query.filter(role=Role.MANAGER)

    # Server will be queried here
    return list(query)
```

4.2.1 Extended syntax and ORing criteria

The basic usage of `MyTableRecord.objects.filter()` — using property names as named arguments — will not allow one to use alternative criteria, as all of them will be ANDed together. To use that, the `Q` operator can be used to encapsulate independent criteria that can be combined with the `||` (double-pipe) operator:

```
from pyrtable.filters import Q

query = MyTableRecord.objects.filter(
        Q(first_name='John') || Q(first_name='Jane'))
```

The `Q` operator will also accept `&&` (double-ampersand) to combine with AND and `~` (tilde) to invert (negate) the enclosed criteria:

```
from pyrtable.filters import Q

# These are all the same:
query = MyTableRecord.objects.filter(
        first_name='John', last_name='Doe', age__ne=30)
query = MyTableRecord.objects.filter(
        Q(first_name='John') && Q(last_name='Doe') && Q(age__ne=30))
query = MyTableRecord.objects.filter(
        Q(first_name='John') && Q(last_name='Doe') && ~Q(age=30))
```

4.3 Retrieving a single record

If you have the Airtable record ID, you can use `MyTableRecord.objects.get(id)` to retrieve the corresponding record. However, referencing a record by its ID is not required for common use cases.

4.4 Updating records

Changing record properties is allowed for any field not declared with `read_only=True`. However, you must tell Pyrtable that you want to persist these changes in the server. To do that you call the record's `.save()` method:

```
# Create a query to fetch people named "John Doe"
# (remember, this does not hit the server yet)
query = MyTableRecord.objects.filter(
    first_name='John', last_name='Doe')

# Get the first record that matches the filtering criteria
record = next(iter(query))
```

(continues on next page)

(continued from previous page)

```
# Change some values
record.last_name = 'Chloe'
record.age = 35

# Send (persist) changes to the server
record.save()
```

Pyrtable is clever enough to avoid sending a server request if no changes were made in the record:

```
record.age += 10
record.age -= 10
# The last operation reverted the former one:
# at the end the record did not change at all.
# The next call will *not* send a server request:
record.save()
```

4.5 Creating records

To create a record, you first populate its field values and then call the `.save()` method:

```
# Create the object and set the properties one by one
new_record = MyTableRecord()
new_record.first_name = 'John'
new_record.last_name = 'Doe'
new_record.age = 35

# You can also set (some) properties when creating the object
new_record = MyTableRecord(
    first_name='John', last_name='Doe', age=35)

# Create the record in the server
new_record.save()
```

4.6 Deleting records

A record can be deleted from the server by calling its `.delete()` method:

```
query = MyTableRecord.objects.filter(
    first_name='John', last_name='Doe')
record = next(iter(query))

# Delete this record
record.delete()
```

CHAPTER 5

Caching records

By default, Pyrtable does not use any caching mechanism. In other words, any query operation will hit the Airtable server to fetch fresh data. In the example below the server will be queried twice:

```
class EmployeeRecord(BaseRecord):
    class Meta:
        # Meta data

    name = StringField('Name')

if __name__ == '__main__':
    # At this point there is no communication with the server --
    # query is being built but not iterated over:
    employees_query = EmployeeRecord.objects.all()

    # Now data will be fetched from the server:
    for employee in employees_query:
        print(employee.name)

    # Since no caching mechanism is active,
    # data will be fetched *again* from the server:
    for employee in employees_query:
        print(employee.name)
```

This can be reduced to a single server hit by storing the query results beforehand:

```
class EmployeeRecord(BaseRecord):
    class Meta:
        # Meta data

    name = StringField('Name')

if __name__ == '__main__':
    # Notice that the query results (not the query itself)
    # is now being stored, so all server communication happens here
```

(continues on next page)

(continued from previous page)

```
    employees = list(EmployeeRecord.objects.all())

    # This happens without server communication
    for employee in employees:
        print(employee.name)

    # This also happens without server communication
    for employee in employees:
        print(employee.name)
```

However, in any case operations can be extremely slow when referring to linked records, i.e., those contained in `SingleRecordLinkField` and `MultipleRecordLinkField` fields. In these cases, the default strategy is to make a new server request *for each linked record*. If you are working on a table with several linked records this will obviously become a waste of resources, especially if some records are linked many times!

To overcome this, Pyrtable offers a caching strategy for linked records. Instead of loading them one by one, you can first fetch all records from the linked table(s), then work normally over the "linking" table. The following example illustrates this strategy:

```
# This table contains the records that will be referred to
# in another table
class EmployeeRecord(BaseRecord):
    class Meta:
        # Meta data

    name = StringField('Name')

# This table has a field that links to the first one
class ProjectRecord(BaseRecord):
    class Meta:
        # Meta data

    team = MultipleRecordLinkField('Team Members',
                                   linked_class='EmployeeRecord')

if __name__ == '__main__':
    from pyrtable.context import set_default_context, SimpleCachingContext

    # Set up the caching mechanism
    caching_context = SimpleCachingContext()
    set_default_context(caching_context)

    # Fetch and cache all records from the Employee table
    caching_context.pre_cache(EmployeeRecord)

    # From now on references to ``.team`` field
    # will not require server requests
    for project in ProjectRecord.objects.all():
        print(project.name,
              ', '.join(employee.name for employee in project.team))
```

5.1 When caching will happen?

Besides calling `caching_context.pre_cache(RecordClass)`, this mechanism will also cache *any* record that is fetched from the server. So, after using `set_default_context(SimpleCachingContext())` any

linked records will be fetched only once.

Note: If you read the source code you will notice that calling `caching_context.pre_cache(EmployeeRecord)` is the same as simply fetching all table records (as they will be cached). In other words, this call is equivalent to `list(EmployeeRecord.objects.all())`.

5.2 Controlling which tables are cached

Caching all tables may be too much depending on your scenario. This default behaviour can be tuned using constructor arguments for the `SimpleCachingContext` class:

`class SimpleCachingContext(allow_classes=None, exclude_classes=None)`

`allow_classes`, if specified, is a list of record classes that will always be cached. Any classes not listed will not be cached.

`exclude_classes`, on the other hand, is a list of record classes that will never be cached. Any classes not listed will be cached.

5.3 The CachingContext.pre_cache() method

This method can actually receive several arguments. Each argument specifies what is to be cached:

- If the argument is a subclass of `BaseRecord`, then all records will be fetched (by calling `.objects.all()`) and cached.
- If the argument is a query (e.g., `MyTableRecord.objects.filter(...)`), then the records will be fetched and cached.
- If the argument is a single record object (with a non-null `.id`), then this record will be stored in the cache without being fetched from the server.

Pyrtable is a Python 3 library to interface with Airtable's REST API.

There are other Python projects to deal with Airtable. However, most of them basically offer a thin layer to ease authentication and filtering – at the end, the developer still has to deal with JSON encoding/decoding, pagination, request rate limits, and so on.

Pyrtable is a high-level, ORM-like library that hides all these details. It performs automatic mapping between Airtable records and Python objects, allowing CRUD operations while aiming to be intuitive and fun. Programmers used to Django will find many similarities and will be able to interface with Airtable bases in just a couple of minutes.

CHAPTER 6

What does it look like?

Ok, let's have a taste of how one can define a class that maps into records of a table:

```
import enum
from pyrtable.record import BaseRecord
from pyrtable.fields import StringField, DateField, SingleSelectionField, \
        SingleRecordLinkField, MultipleRecordLinkField

class Role(enum.Enum):
    DEVELOPER = 'Developer'
    MANAGER = 'Manager'
    CEO = 'C.E.O.'

class EmployeeRecord(BaseRecord):
    class Meta:
        # Open "Help > API documentation" in Airtable and search for a line
        # starting with "The ID of this base is XXX".
        base_id = 'appABCDE12345'
        table_id = 'Employees'

    @classmethod
    def get_api_key(cls):
        # The API key can be generated in you Airtable Account page.
        # DO NOT COMMIT THIS STRING!
        return 'keyABCDE12345'

    name = StringField('Name')
    birth_date = DateField('Birth date')
    office = SingleRecordLinkField('Office', linked_class='OfficeRecord')
    projects = MultipleRecordLinkField(
            'Allocated in projects', linked_class='ProjectRecord')
    role = SingleSelectionField('Role', choices=Role)
```

After that, common operations are pretty simple:

```
# Iterating over all records
for employee in EmployeeRecord.objects.all():
    print("%s is currently working on %d project(s)" % (
            employee.name, len(employee.projects)))

# Filtering
for employee in EmployeeRecord.objects.filter(
        birth_date__gte=datetime.datetime(2001, 1, 1)):
    print("%s was born in this century!" % employee.name)

# Creating, updating and deleting a record
new_employee = EmployeeRecord(
    name='John Doe',
    birth_date=datetime.date(1980, 5, 10),
    role=Role.DEVELOPER)
new_employee.save()

new_employee.role = Role.MANAGER
new_employee.save()

new_employee.delete()
```

Notice that we don't deal with Airtable column or table names once record classes are defined.

CHAPTER 7

Beyond the basics

Keep in mind that Airtable is *not* a database system and is not really designed for tasks that need changing tons of data. In fact, only fetch (list) operations are batched – insert/update/delete operations are limited to a single record per request, and Airtable imposes a 5 requests per second limit even for paid accounts. You will need a full minute to update 300 records!

That said, Pyrtable will respect that limit and will also track dirty fields to avoid unnecessary server requests, rendering `.save()` calls as no-ops for unchanged objects. So, the following code can actually hit the server zero times inside the `for` loop:

```
all_records = list(EmployeeRecord.objects.all())

# Do operations that change some records here

for record in all_records:
    # Only changed objects will be sent to the server
    # No need to keep track of which records were changed
    record.save()
```

This also works with multiple threads, so the following pattern can be used to update and/or create several records:

```
from concurrent.futures.thread import ThreadPoolExecutor

all_records = list(EmployeeRecord.objects.all())

# Do operations that change some records here

with ThreadPoolExecutor(max_workers=10) as executor:
    for record in all_records:
        # Only changed objects will be sent to the server
        # No need to keep track of which records were changed
        executor.submit(record.save)
```

Or, if you want a really nice tqdm progress bar:

```
from tqdm import tqdm

with ThreadPoolExecutor(max_workers=10) as executor:
    for _ in tqdm(executor.map(lambda record: record.save(), all_records),
                  total=len(all_records), dynamic_ncols=True, unit='',
                  desc='Updating Airtable records'):
        pass
```

Pyrtable also has some extra tools to *cache data* and to store authentication keys *in JSON/YAML files* or *in an environment variable*. Remember to never commit sensitive data to your repository, as Airtable authentication allows **full R/W access to all your bases** with a single API key!

CHAPTER 8

Compatibility

Pyrtable is compatible with Python 3.8 and 3.9. Python 2.x is not supported at all.

CHAPTER 9

Questions, bug reports, improvements

Want to try it out, contribute, suggest, offer a hand? Great! The project is available at https://github.com/vilarneto/pyrtable.

CHAPTER 10

License

Pyrtable is released under MIT license.

CHAPTER 11

Indices and tables

- genindex
- search

Index

S

T

U