Airtable Python Wrapper Documentation

Gui Talarico

Jul 28, 2021

CONTENTS

Version:

For more information about the Airtable API see the Airtable API Docs

CHAPTER

ONE

INSTALLATION

```
>>> pip install airtable-python-wrapper
```

CHAPTER

TWO

INDEX

2.1 Airtable Class

2.1.1 Overview

Airtable Class Instance

```
>>> airtable = Airtable('base_id', 'table_name')
>>> airtable.get_all()
[{id:'rec123asa23', fields': {'Column': 'Value'}, ...}]
```

For more information on Api Key and authentication see the *Airtable Authentication*.

Examples

For a full list of available methods see the *Airtable* class below. For more details on the Parameter filters see the documentation on the available *Parameter Filters* as well as the Airtable API Docs

Record/Page Iterator:

```
>>> for page in airtable.get_iter(view='ViewName',sort='COLUMN_A'):
...     for record in page:
...         value = record['fields']['COLUMN_A']
```

Get all Records:

```
>>> airtable.get_all(view='ViewName',sort='COLUMN_A')
[{id:'rec123asa23', 'fields': {'COLUMN_A': 'Value', ...}, ... ]
```

Search:

```
>>> airtable.search('ColumnA', 'SearchValue')
```

Formulas:

```
>>> airtable.get_all(formula="FIND('DUP', {COLUMN_STR})=1")
```

Insert:

```
>>> airtable.insert({'First Name': 'John'})
```

Delete:

```
>>> airtable.delete('recwPQIfs4wKPyc9D')
```

You can see the Airtable Class in action in this Jupyter Notebook

---

Return Values

Return Values: when records are returned, they will most often be a list of Airtable records (dictionary) in a format similar to this:

```
>>> [{
...     "records": [
...         {
...             "id": "recwPQIfs4wKPyc9D",
...             "fields": {
...                 "COLUMN_ID": "1",
...             },
...             "createdTime": "2017-03-14T22:04:31.000Z"
...         },
...         {
...             "id": "rechOLltN9SpPHq5o",
...             "fields": {
...                 "COLUMN_ID": "2",
...             },
...             "createdTime": "2017-03-20T15:21:50.000Z"
...         },
...         {
...             "id": "rec5eR7IzKSAOBHCz",
...             "fields": {
...                 "COLUMN_ID": "3",
...             },
...             "createdTime": "2017-08-05T21:47:52.000Z"
...         }
...     ],
...     "offset": "rec5eR7IzKSAOBHCz"
... }, ... ]
```

2.1.2 Class API

class airtable.**Airtable**(*base_id*, *table_name*, *api_key*, *timeout=None*)

__init__(*base_id*, *table_name*, *api_key*, *timeout=None*)

Instantiates a new Airtable instance

```
>>> table = Airtable('base_id', "tablename")
```

With timeout:

```
>>> table = Airtable('base_id', "tablename", timeout=(1, 1))
```

Parameters

- **base_id** (str) – Airtable base identifier
- **table_name** (str) – Airtable table name. Value will be url encoded, so use value as shown in Airtable.
- **api_key** (str) – API key.

Keyword Arguments **timeout** (int, Tuple[int, int], optional) – Optional timeout parameters to be used in request. See requests timeout docs.

batch_delete(*record_ids*)

Breaks records into batches of 10 and deletes in batches, following set API Rate Limit (5/sec). To change the rate limit set value of `airtable.API_LIMIT` to the time in seconds it should sleep before calling the function again.

```
>>> record_ids = ['recwPQIfs4wKPyc9D', 'recwDxIfs3wDPyc3F']
>>> airtable.batch_delete(records_ids)
```

Parameters **records** (list) – Record Ids to delete

Returns list of records deleted

Return type records(list)

batch_insert(*records*, *typecast=False*)

Breaks records into chunks of 10 and inserts them in batches. Follows the set API rate. To change the rate limit use `airtable.API_LIMIT = 0.2` (5 per second)

```
>>> records = [{'Name': 'John'}, {'Name': 'Marc'}]
>>> airtable.batch_insert(records)
```

Parameters

- **records** (list) – Records to insert
- **typecast** (boolean) – Automatic data conversion from string values.

Returns list of added records

Return type records (list)

batch_update(*records*, *typecast=False*)

Updates a records by their record id's in batch.

Parameters

- **records** (list) – List of dict: [{"id": record_id, "fields": fields_to_update_dict}]
- **typecast** (boolean) – Automatic data conversion from string values.

Returns list of updated records

Return type records(list)

delete(*record_id*)

Deletes a record by its id

```
>>> record = airtable.match('Employee Id', 'DD13332454')
>>> airtable.delete(record['id'])
```

Parameters **record_id** (str) – Airtable record id

Returns Deleted Record

Return type record (dict)

delete_by_field(*field_name*, *field_value*, ***options*)

Deletes first record to match provided field_name and field_value.

```
>>> record = airtable.delete_by_field('Employee Id', 'DD13332454')
```

Parameters

- **field_name** (str) – Name of field to match (column name).
- **field_value** (str) – Value of field to match.

Keyword Arguments

- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.

Returns Deleted Record

Return type record (dict)

get(*record_id*)

Retrieves a record by its id

```
>>> record = airtable.get('recwPQIfs4wKPyc9D')
```

Parameters **record_id** (str) – Airtable record id

Returns Record

Return type record (dict)

get_all(***options*)

Retrieves all records repetitively and returns a single list.

```
>>> airtable.get_all()
>>> airtable.get_all(view='MyView', fields=['ColA', '-ColB'])
>>> airtable.get_all(maxRecords=50)
[{'fields': ... }, ...]
```

Keyword Arguments

- **max_records** (int, optional) – The maximum total number of records that will be returned. See *MaxRecordsParam*
- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **fields** (str, list, optional) – Name of field or fields to be retrieved. Default is all fields. See *FieldsParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.
- **formula** – Airtable formula. See *FormulaParam*.

get_iter(***options*)

Record Retriever Iterator

Returns iterator with lists in batches according to pageSize. To get all records at once use *get_all*

```
>>> for page in airtable.get_iter():
...     for record in page:
...         print(record)
[{'fields': ... }, ...]
```

Keyword Arguments

- **max_records** (int, optional) – The maximum total number of records that will be returned. See *MaxRecordsParam*
- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **page_size** (int, optional) – The number of records returned in each request. Must be less than or equal to 100. Default is 100. See *PageSizeParam*.
- **fields** (str, list, optional) – Name of field or fields to be retrieved. Default is all fields. See *FieldsParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.
- **formula** – Airtable formula. See *FormulaParam*.

insert(*fields*, *typecast=False*)

Inserts a record

```
>>> record = {'Name': 'John'}
>>> airtable.insert(record)
```

Parameters

- **fields** (dict) – Fields to insert. Must be dictionary with Column names as Key.

- **typecast** (boolean) – Automatic data conversion from string values.

Returns Inserted record

Return type record (dict)

match(*field_name*, *field_value*, ***options*)

Returns first match found in *get_all*

```
>>> airtable.match('Name', 'John')
{'fields': {'Name': 'John'} }
```

Parameters

- **field_name** (str) – Name of field to match (column name).
- **field_value** (str) – Value of field to match.

Keyword Arguments

- **max_records** (int, optional) – The maximum total number of records that will be returned. See *MaxRecordsParam*
- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **fields** (str, list, optional) – Name of field or fields to be retrieved. Default is all fields. See *FieldsParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.

Returns First record to match the field_value provided

Return type record (dict)

record_url(*record_id*)

Builds URL with record id

replace(*record_id*, *fields*, *typecast=False*)

Replaces a record by its record id. All Fields are updated to match the new `fields` provided. If a field is not included in `fields`, value will bet set to null. To update only selected fields, use *update*.

```
>>> record = airtable.match('Seat Number', '22A')
>>> fields = {'PassangerName': 'Mike', 'Passport': 'YASD232-23'}
>>> airtable.replace(record['id'], fields)
```

Parameters

- **record_id** (str) – Id of Record to update
- **fields** (dict) – Fields to replace with. Must be dictionary with Column names as Key.
- **typecast** (boolean) – Automatic data conversion from string values.

Returns New record

Return type record (dict)

replace_by_field(*field_name*, *field_value*, *fields*, *typecast=False*, ***options*)

Replaces the first record to match field name and value. All Fields are updated to match the new `fields` provided. If a field is not included in `fields`, value will bet set to null. To update only selected fields, use *update*.

Parameters

- **field_name** (str) – Name of field to match (column name).
- **field_value** (str) – Value of field to match.
- **fields** (dict) – Fields to replace with. Must be dictionary with Column names as Key.
- **typecast** (boolean) – Automatic data conversion from string values.

Keyword Arguments

- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.

Returns New record

Return type record (dict)

search(*field_name*, *field_value*, *record=None*, ***options*)

Returns all matching records found in *get_all*

```
>>> airtable.search('Gender', 'Male')
[{'fields': {'Name': 'John', 'Gender': 'Male'}, ... ]
```

```
>>> airtable.search('Checkbox Field', 1)
[{'fields': {'Name': 'John', 'Gender': 'Male'}, ... ]
```

Parameters

- **field_name** (str) – Name of field to match (column name).
- **field_value** (str) – Value of field to match.

Keyword Arguments

- **max_records** (int, optional) – The maximum total number of records that will be returned. See *MaxRecordsParam*
- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **fields** (str, list, optional) – Name of field or fields to be retrieved. Default is all fields. See *FieldsParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.

Returns All records that matched field_value

Return type records (list)

update(*record_id*, *fields*, *typecast=False*)

Updates a record by its record id. Only Fields passed are updated, the rest are left as is.

```
>>> record = airtable.match('Employee Id', 'DD13332454')
>>> fields = {'Status': 'Fired'}
>>> airtable.update(record['id'], fields)
```

Parameters

- **record_id** (str) – Id of Record to update

- **fields** (dict) – Fields to update. Must be dictionary with Column names as Key
- **typecast** (boolean) – Automatic data conversion from string values.

Returns Updated record

Return type record (dict)

update_by_field(*field_name*, *field_value*, *fields*, *typecast=False*, ***options*)

Updates the first record to match field name and value. Only Fields passed are updated, the rest are left as is.

```
>>> record = {'Name': 'John', 'Tel': '540-255-5522'}
>>> airtable.update_by_field('Name', 'John', record)
```

Parameters

- **field_name** (str) – Name of field to match (column name).
- **field_value** (str) – Value of field to match.
- **fields** (dict) – Fields to update. Must be dictionary with Column names as Key
- **typecast** (boolean) – Automatic data conversion from string values.

Keyword Arguments

- **view** (str, optional) – The name or ID of a view. See *ViewParam*.
- **sort** (list, optional) – List of fields to sort by. Default order is ascending. See *SortParam*.

Returns Updated record

Return type record (dict)

2.1.3 Source Code

```
import requests
from collections import OrderedDict
import posixpath
import time
from urllib.parse import quote

from .auth import AirtableAuth
from .params import AirtableParams


class Airtable(object):

    VERSION = "v0"
    API_BASE_URL = "https://api.airtable.com/"
    API_LIMIT = 1.0 / 5  # 5 per second
    API_URL = posixpath.join(API_BASE_URL, VERSION)
```

(continues on next page)

(continued from previous page)

```
    MAX_RECORDS_PER_REQUEST = 10

    def __init__(self, base_id, table_name, api_key, timeout=None):
        """
        Instantiates a new Airtable instance

        >>> table = Airtable('base_id', "tablename")

        With timeout:

        >>> table = Airtable('base_id', "tablename", timeout=(1, 1))

        Args:
            base_id(``str``): Airtable base identifier
            table_name(``str``): Airtable table name. Value will be url encoded, so
                use value as shown in Airtable.
            api_key (``str``): API key.

        Keyword Args:
            timeout (``int``, ``Tuple[int, int]``, optional): Optional timeout
                parameters to be used in request. `See requests timeout docs.
                <https://requests.readthedocs.io/en/master/user/advanced/#timeouts>`_

        """
        session = requests.Session()
        session.auth = AirtableAuth(api_key=api_key)
        self.session = session
        self.table_name = table_name
        url_safe_table_name = quote(table_name, safe="")
        self.url_table = posixpath.join(self.API_URL, base_id, url_safe_table_name)
        self.timeout = timeout

    def _process_params(self, params):
        """
        Process params names or values as needed using filters
        """
        new_params = OrderedDict()
        for param_name, param_value in sorted(params.items()):
            param_value = params[param_name]
            ParamClass = AirtableParams._get(param_name)
            new_params.update(ParamClass(param_value).to_param_dict())
        return new_params

    def _chunk(self, iterable, chunk_size):
        """Break iterable into chunks."""
        for i in range(0, len(iterable), chunk_size):
            yield iterable[i : i + chunk_size]

    def _build_batch_record_objects(self, records):
        return [{"fields": record} for record in records]

    def _process_response(self, response):
```

(continues on next page)

(continued from previous page)

```
        try:
            response.raise_for_status()
        except requests.exceptions.HTTPError as exc:
            err_msg = str(exc)

            # Attempt to get Error message from response, Issue #16
            try:
                error_dict = response.json()
            except ValueError:
                pass
            else:
                if "error" in error_dict:
                    err_msg += " [Error: {}]".format(error_dict["error"])
            exc.args = (*exc.args, err_msg)
            raise exc
        else:
            return response.json()

    def record_url(self, record_id):
        """ Builds URL with record id """
        return posixpath.join(self.url_table, record_id)

    def _request(self, method, url, params=None, json_data=None):
        response = self.session.request(
            method, url, params=params, json=json_data, timeout=self.timeout
        )
        return self._process_response(response)

    def _get(self, url, **params):
        processed_params = self._process_params(params)
        return self._request("get", url, params=processed_params)

    def _post(self, url, json_data):
        return self._request("post", url, json_data=json_data)

    def _put(self, url, json_data):
        return self._request("put", url, json_data=json_data)

    def _patch(self, url, json_data):
        return self._request("patch", url, json_data=json_data)

    def _delete(self, url):
        return self._request("delete", url)

    def _delete_batch(self, record_ids):
        if len(record_ids) == 1:
            return self.delete(record_ids[0])

        return self._request("delete", self.url_table, params={"records": record_ids})

    def get(self, record_id):
        """
```

(continues on next page)

(continued from previous page)

```
        Retrieves a record by its id

        >>> record = airtable.get('recwPQIfs4wKPyc9D')

        Args:
            record_id(``str``): Airtable record id

        Returns:
            record (``dict``): Record
        """
        record_url = self.record_url(record_id)
        return self._get(record_url)

    def get_iter(self, **options):
        """
        Record Retriever Iterator

        Returns iterator with lists in batches according to pageSize.
        To get all records at once use :any:`get_all`

        >>> for page in airtable.get_iter():
        ...     for record in page:
        ...         print(record)
        [{'fields': ... }, ...]


    Keyword Args:
            max_records (``int``, optional): The maximum total number of
                records that will be returned. See :any:`MaxRecordsParam`
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            page_size (``int``, optional ): The number of records returned
                in each request. Must be less than or equal to 100.
                Default is 100. See :any:`PageSizeParam`.
            fields (``str``, ``list``, optional): Name of field or fields to
                be retrieved. Default is all fields. See :any:`FieldsParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.
            formula (``str``, optional): Airtable formula.
                See :any:`FormulaParam`.

        Returns:
            iterator (``list``): List of Records, grouped by pageSize

        """
        offset = None
        while True:
            data = self._get(self.url_table, offset=offset, **options)
            records = data.get("records", [])
            time.sleep(self.API_LIMIT)
            yield records
            offset = data.get("offset")
```

(continues on next page)

(continued from previous page)

```
            if not offset:
                break

    def get_all(self, **options):
        """
        Retrieves all records repetitively and returns a single list.

        >>> airtable.get_all()
        >>> airtable.get_all(view='MyView', fields=['ColA', '-ColB'])
        >>> airtable.get_all(maxRecords=50)
        [{'fields': ... }, ...]


    Keyword Args:
            max_records (``int``, optional): The maximum total number of
                records that will be returned. See :any:`MaxRecordsParam`
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            fields (``str``, ``list``, optional): Name of field or fields to
                be retrieved. Default is all fields. See :any:`FieldsParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.
            formula (``str``, optional): Airtable formula.
                See :any:`FormulaParam`.

        Returns:
            records (``list``): List of Records

        >>> records = get_all(maxRecords=3, view='All')

        """
        all_records = []
        for records in self.get_iter(**options):
            all_records.extend(records)
        return all_records

    def match(self, field_name, field_value, **options):
        """
        Returns first match found in :any:`get_all`

        >>> airtable.match('Name', 'John')
        {'fields': {'Name': 'John'} }

        Args:
            field_name (``str``): Name of field to match (column name).
            field_value (``str``): Value of field to match.

        Keyword Args:
            max_records (``int``, optional): The maximum total number of
                records that will be returned. See :any:`MaxRecordsParam`
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
```

(continues on next page)

(continued from previous page)

```
            fields (``str``, ``list``, optional): Name of field or fields to
                be retrieved. Default is all fields. See :any:`FieldsParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.

        Returns:
            record (``dict``): First record to match the field_value provided
        """
        from_name_and_value = AirtableParams.FormulaParam.from_name_and_value
        formula = from_name_and_value(field_name, field_value)
        options["formula"] = formula
        for record in self.get_all(**options):
            return record
        else:
            return {}

    def search(self, field_name, field_value, record=None, **options):
        """
        Returns all matching records found in :any:`get_all`

        >>> airtable.search('Gender', 'Male')
        [{'fields': {'Name': 'John', 'Gender': 'Male'}, ... ]

        >>> airtable.search('Checkbox Field', 1)
        [{'fields': {'Name': 'John', 'Gender': 'Male'}, ... ]

        Args:
            field_name (``str``): Name of field to match (column name).
            field_value (``str``): Value of field to match.

        Keyword Args:
            max_records (``int``, optional): The maximum total number of
                records that will be returned. See :any:`MaxRecordsParam`
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            fields (``str``, ``list``, optional): Name of field or fields to
                be retrieved. Default is all fields. See :any:`FieldsParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.

        Returns:
            records (``list``): All records that matched ``field_value``

        """
        records = []
        from_name_and_value = AirtableParams.FormulaParam.from_name_and_value
        formula = from_name_and_value(field_name, field_value)
        options["formula"] = formula
        records = self.get_all(**options)
        return records

    def insert(self, fields, typecast=False):
```

(continues on next page)

(continued from previous page)

```
        """
        Inserts a record

        >>> record = {'Name': 'John'}
        >>> airtable.insert(record)

        Args:
            fields(``dict``): Fields to insert.
                Must be dictionary with Column names as Key.
            typecast(``boolean``): Automatic data conversion from string values.

        Returns:
            record (``dict``): Inserted record

        """
        return self._post(
            self.url_table, json_data={"fields": fields, "typecast": typecast}
        )

    def batch_insert(self, records, typecast=False):
        """
        Breaks records into chunks of 10 and inserts them in batches.
        Follows the set API rate.
        To change the rate limit use ``airtable.API_LIMIT = 0.2``
        (5 per second)

        >>> records = [{'Name': 'John'}, {'Name': 'Marc'}]
        >>> airtable.batch_insert(records)

        Args:
            records(``list``): Records to insert
            typecast(``boolean``): Automatic data conversion from string values.

        Returns:
            records (``list``): list of added records
        """
        inserted_records = []
        for chunk in self._chunk(records, self.MAX_RECORDS_PER_REQUEST):
            new_records = self._build_batch_record_objects(chunk)
            response = self._post(
                self.url_table, json_data={"records": new_records, "typecast": typecast}
            )
            inserted_records += response["records"]
            time.sleep(self.API_LIMIT)
        return inserted_records

    def update(self, record_id, fields, typecast=False):
        """
        Updates a record by its record id.
        Only Fields passed are updated, the rest are left as is.

        >>> record = airtable.match('Employee Id', 'DD13332454')
```

(continues on next page)

(continued from previous page)

```
        >>> fields = {'Status': 'Fired'}
        >>> airtable.update(record['id'], fields)

        Args:
            record_id(``str``): Id of Record to update
            fields(``dict``): Fields to update.
                Must be dictionary with Column names as Key
            typecast(``boolean``): Automatic data conversion from string values.

        Returns:
            record (``dict``): Updated record
        """
        record_url = self.record_url(record_id)
        return self._patch(
            record_url, json_data={"fields": fields, "typecast": typecast}
        )

    def batch_update(self, records, typecast=False):
        """
        Updates a records by their record id's in batch.

        Args:
            records(``list``): List of dict: [{"id": record_id, "fields": fields_to_
↪update_dict}]
            typecast(``boolean``): Automatic data conversion from string values.

        Returns:
            records(``list``): list of updated records
        """
        updated_records = []
        for chunk in self._chunk(records, self.MAX_RECORDS_PER_REQUEST):
            chunk_records = [{"id": x["id"], "fields": x["fields"]} for x in chunk]
            response = self._patch(
                self.url_table, json_data={"records": chunk_records, "typecast":␣
↪typecast}
            )
            updated_records += response["records"]
        #
        return updated_records

    def update_by_field(
        self, field_name, field_value, fields, typecast=False, **options
    ):
        """
        Updates the first record to match field name and value.
        Only Fields passed are updated, the rest are left as is.

        >>> record = {'Name': 'John', 'Tel': '540-255-5522'}
        >>> airtable.update_by_field('Name', 'John', record)

        Args:
            field_name (``str``): Name of field to match (column name).
```

(continues on next page)

(continued from previous page)

```
            field_value (``str``): Value of field to match.
            fields(``dict``): Fields to update.
                Must be dictionary with Column names as Key
            typecast(``boolean``): Automatic data conversion from string values.

        Keyword Args:
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.

        Returns:
            record (``dict``): Updated record
        """
        record = self.match(field_name, field_value, **options)
        return {} if not record else self.update(record["id"], fields, typecast)

    def replace(self, record_id, fields, typecast=False):
        """
        Replaces a record by its record id.
        All Fields are updated to match the new ``fields`` provided.
        If a field is not included in ``fields``, value will bet set to null.
        To update only selected fields, use :any:`update`.

        >>> record = airtable.match('Seat Number', '22A')
        >>> fields = {'PassangerName': 'Mike', 'Passport': 'YASD232-23'}
        >>> airtable.replace(record['id'], fields)

        Args:
            record_id(``str``): Id of Record to update
            fields(``dict``): Fields to replace with.
                Must be dictionary with Column names as Key.
            typecast(``boolean``): Automatic data conversion from string values.

        Returns:
            record (``dict``): New record
        """
        record_url = self.record_url(record_id)
        return self._put(record_url, json_data={"fields": fields, "typecast": typecast})

    def replace_by_field(
        self, field_name, field_value, fields, typecast=False, **options
    ):
        """
        Replaces the first record to match field name and value.
        All Fields are updated to match the new ``fields`` provided.
        If a field is not included in ``fields``, value will bet set to null.
        To update only selected fields, use :any:`update`.

        Args:
            field_name (``str``): Name of field to match (column name).
            field_value (``str``): Value of field to match.
```

(continues on next page)

(continued from previous page)

```
            fields(``dict``): Fields to replace with.
                Must be dictionary with Column names as Key.
            typecast(``boolean``): Automatic data conversion from string values.

        Keyword Args:
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.

        Returns:
            record (``dict``): New record
        """
        record = self.match(field_name, field_value, **options)
        return {} if not record else self.replace(record["id"], fields, typecast)

    def delete(self, record_id):
        """
        Deletes a record by its id

        >>> record = airtable.match('Employee Id', 'DD13332454')
        >>> airtable.delete(record['id'])

        Args:
            record_id(``str``): Airtable record id

        Returns:
            record (``dict``): Deleted Record
        """
        record_url = self.record_url(record_id)
        return self._delete(record_url)

    def delete_by_field(self, field_name, field_value, **options):
        """
        Deletes first record  to match provided ``field_name`` and
        ``field_value``.

        >>> record = airtable.delete_by_field('Employee Id', 'DD13332454')

        Args:
            field_name (``str``): Name of field to match (column name).
            field_value (``str``): Value of field to match.

        Keyword Args:
            view (``str``, optional): The name or ID of a view.
                See :any:`ViewParam`.
            sort (``list``, optional): List of fields to sort by.
                Default order is ascending. See :any:`SortParam`.

        Returns:
            record (``dict``): Deleted Record
        """
```

(continues on next page)

(continued from previous page)

```
        record = self.match(field_name, field_value, **options)
        record_url = self.record_url(record["id"])
        return self._delete(record_url)

    def batch_delete(self, record_ids):
        """
        Breaks records into batches of 10 and deletes in batches, following set
        API Rate Limit (5/sec).
        To change the rate limit set value of ``airtable.API_LIMIT`` to
        the time in seconds it should sleep before calling the function again.

        >>> record_ids = ['recwPQIfs4wKPyc9D', 'recwDxIfs3wDPyc3F']
        >>> airtable.batch_delete(records_ids)

        Args:
            records(``list``): Record Ids to delete

        Returns:
            records(``list``): list of records deleted

        """
        chunks = self._chunk(record_ids, self.MAX_RECORDS_PER_REQUEST)
        deleted_records = []
        for chunk in chunks:
            response = self._delete_batch(chunk)
            deleted_records += response["records"] if len(chunk) > 1 else [response]
            time.sleep(self.API_LIMIT)
        return deleted_records

    def __repr__(self):
        return "<Airtable table:{}>".format(self.table_name)
```

2.2 Parameter Filters

2.2.1 Overview

Parameter filters are instantiated internally by using the corresponding keywords.

Filter names (kwargs) can be either the API camelCase name (ie `maxRecords`) or the snake-case equivalent (`max_records`).

Refer to the *`Airtable`* class to verify which kwargs can be used with each method.

The purpose of these classes is to 1. improve flexibility and ways in which parameter filter values can be passed, and 2. properly format the parameter names and values on the request url.

For more information see the full implementation below.

2.2.2 Parameter Filters

class airtable.params.**AirtableParams**

class FieldsParam(*value*)
Fields Param

Kwargs: fields=

Only data for fields whose names are in this list will be included in the records. If you don't need every field, you can use this parameter to reduce the amount of data transferred.

Usage:

```
>>> airtable.get(fields='ColumnA')
```

Multiple Columns:

```
>>> airtable.get(fields=['ColumnA', 'ColumnB'])
```

Parameters **fields** (str, list) – Name of columns you want to retrieve.

class FormulaParam(*value*)
Formula Param

Kwargs: formula= or filterByFormula=

The formula will be evaluated for each record, and if the result is not 0, false, "", NaN, [], or #Error! the record will be included in the response.

If combined with view, only records in that view which satisfy the formula will be returned. For example, to only include records where COLUMN_A isn't empty, pass in: "NOT({COLUMN_A}='')"

For more information see Airtable Docs on formulas.

Usage - Text Column is not empty:

```
>>> airtable.get_all(formula="NOT({COLUMN_A}='')")
```

Usage - Text Column contains:

```
>>> airtable.get_all(formula="FIND('SomeSubText', {COLUMN_STR})=1")
```

Parameters **formula** (str) – A valid Airtable formula.

static from_name_and_value(*field_name*, *field_value*)
Creates a formula to match cells from from field_name and value

class MaxRecordsParam(*value*)
Max Records Param

Kwargs: max_records= or maxRecords=

The maximum total number of records that will be returned.

Usage:

```
>>> airtable.get_all(max_records=10)
```

Parameters **max_records** (int) – The maximum total number of records that will be returned.

class PageSizeParam(*value*)

Page Size Param

Kwargs: page_size= or pageSize=

Limits the maximum number of records returned in each request. Default is 100.

Usage:

```
>>> airtable.get_all(page_size=50)
```

Parameters **page_size** (int) – The number of records returned in each request. Must be less than or equal to 100. Default is 100.

class SortParam(*value*)

Sort Param

Kwargs: sort=

Specifies how the records will be ordered. If you set the view parameter, the returned records in that view will be sorted by these fields.

If sorting by multiple columns, column names can be passed as a list. Sorting Direction is ascending by default, but can be reversed by prefixing the column name with a minus sign -, or passing COLUMN_NAME, DIRECTION tuples. Direction options are asc and desc.

Usage:

```
>>> airtable.get(sort='ColumnA')
```

Multiple Columns:

```
>>> airtable.get(sort=['ColumnA', '-ColumnB'])
```

Explicit Directions:

```
>>> airtable.get(sort=[('ColumnA', 'asc'), ('ColumnB', 'desc')])
```

Parameters **fields** (str, list) – Name of columns and directions.

class ViewParam(*value*)

View Param

Kwargs: view=

If set, only the records in that view will be returned. The records will be sorted according to the order of the view.

Usage:

```
>>> airtable.get_all(view='My View')
```

Parameters **view** (str) – The name or ID of a view.

2.2.3 Source Code

```
from collections import OrderedDict
import re


class _BaseParam(object):
    def __init__(self, value):
        self.value = value

    def to_param_dict(self):
        return {self.param_name: self.value}


class _BaseStringArrayParam(_BaseParam):
    """
    Api Expects Array Of Strings:
    >>> ['FieldOne', 'Field2']

    Requests Params Input:
    >>> params={'fields': ['FieldOne', 'FieldTwo']}

    Requests Url Params Encoding:
    >>> ?fields=FieldOne&fields=FieldTwo

    Expected Url Params:
    >>> ?fields[]=FieldOne&fields[]=FieldTwo
    """

    def to_param_dict(self):
        encoded_param = self.param_name + "[]"
        return {encoded_param: self.value}


class _BaseObjectArrayParam(_BaseParam):
    """
    Api Expects Array of Objects:
    >>> [{field: "UUID", direction: "desc"}, {...}]

    Requests Params Input:
    >>> params={'sort': ['FieldOne', '-FieldTwo']}
    or
    >>> params={'sort': [('FieldOne', 'asc'), ('-FieldTwo', 'desc')]}

    Requests Url Params Encoding:
    >>> ?sort=field&sort=direction&sort=field&sort=direction

    Expected Url Params:
    >>> ?sort[0][field]=FieldOne&sort[0][direction]=asc
    """

    def to_param_dict(self):
```

(continues on next page)

(continued from previous page)

```
        """ Sorts to ensure Order is consistent for Testing """
        param_dict = {}
        for index, dictionary in enumerate(self.value):
            for key, value in dictionary.items():
                param_name = "{param_name}[{index}][{key}]".format(
                    param_name=self.param_name, index=index, key=key
                )
                param_dict[param_name] = value
        return OrderedDict(sorted(param_dict.items()))


class AirtableParams(object):
    class MaxRecordsParam(_BaseParam):
        """
        Max Records Param

        Kwargs:
            ``max_records=`` or ``maxRecords=``

        The maximum total number of records that will be returned.

        Usage:

        >>> airtable.get_all(max_records=10)

        Args:
            max_records (``int``): The maximum total number of records that
                will be returned.


        """

        # Class Input > Output
        # >>> filter = MaxRecordsParam(100)
        # >>> filter.to_param_dict()
        # {'maxRecords: 100}

        param_name = "maxRecords"
        kwarg = "max_records"

    class ViewParam(_BaseParam):
        """
        View Param

        Kwargs:
            ``view=``

        If set, only the records in that view will be returned.
        The records will be sorted according to the order of the view.

        Usage:
```

(continues on next page)

(continued from previous page)

```
        >>> airtable.get_all(view='My View')

        Args:
            view (``str``): The name or ID of a view.

        """

        # Class Input > Output
        # >>> filter = ViewParam('Name or Id Of View')
        # >>> filter.to_param_dict()
        # {'view: 'Name or Id Of View'}

        param_name = "view"
        kwarg = param_name

    class PageSizeParam(_BaseParam):
        """
        Page Size Param

        Kwargs:
            ``page_size=`` or ``pageSize=``

        Limits the maximum number of records returned in each request.
        Default is 100.

        Usage:

        >>> airtable.get_all(page_size=50)

        Args:
            page_size (``int``): The number of records returned in each request.
                Must be less than or equal to 100. Default is 100.

        """

        # Class Input > Output
        # >>> filter = PageSizeParam(50)
        # >>> filter.to_param_dict()
        # {'pageSize: 50}

        param_name = "pageSize"
        kwarg = "page_size"

    class FormulaParam(_BaseParam):
        """
        Formula Param

        Kwargs:
            ``formula=`` or ``filterByFormula=``

        The formula will be evaluated for each record, and if the result
        is not 0, false, "", NaN, [], or #Error! the record will be included
```

(continues on next page)

(continued from previous page)

```
        in the response.

        If combined with view, only records in that view which satisfy the
        formula will be returned. For example, to only include records where
        ``COLUMN_A`` isn't empty, pass in: ``"NOT({COLUMN_A}='')"``

        For more information see
        `Airtable Docs on formulas. <https://airtable.com/api>`_

        Usage - Text Column is not empty:

        >>> airtable.get_all(formula="NOT({COLUMN_A}='')")

        Usage - Text Column contains:

        >>> airtable.get_all(formula="FIND('SomeSubText', {COLUMN_STR})=1")

        Args:
            formula (``str``): A valid Airtable formula.

        """

        # Class Input > Output
        # >>> param = FormulaParams("FIND('DUP', {COLUMN_STR})=1")
        # >>> param.to_param_dict()
        # {'formula': "FIND('WW')=1"}

        param_name = "filterByFormula"
        kwarg = "formula"

        @staticmethod
        def from_name_and_value(field_name, field_value):
            """
            Creates a formula to match cells from from field_name and value
            """
            if isinstance(field_value, str):
                field_value = re.sub("(?<!\\\\)'", "\\'", field_value)
                field_value = "'{}'".format(field_value)

            formula = "{{{name}}}={value}".format(name=field_name, value=field_value)
            return formula

    class _OffsetParam(_BaseParam):
        """
        Offset Param

        Kwargs:
            ``offset=``

        If there are more records what was in the response,
        the response body will contain an offset value.
        To fetch the next page of records,
```

(continues on next page)

(continued from previous page)

```
        include offset in the next request's parameters.

        This is used internally by :any:`get_all` and :any:`get_iter`.

        Usage:

        >>> airtable.get_iter(offset='recjAle5lryYOpMKk')

        Args:
            record_id (``str``, ``list``):

        """

        # Class Input > Output
        # >>> filter = _OffsetParam('recqgqThAnETLuH58')
        # >>> filter.to_param_dict()
        # {'offset: 'recqgqThAnETLuH58'}

        param_name = "offset"
        kwarg = param_name

    class FieldsParam(_BaseStringArrayParam):
        """
        Fields Param

        Kwargs:
            ``fields=``

        Only data for fields whose names are in this list will be included in
        the records. If you don't need every field, you can use this parameter
        to reduce the amount of data transferred.

        Usage:

        >>> airtable.get(fields='ColumnA')

        Multiple Columns:

        >>> airtable.get(fields=['ColumnA', 'ColumnB'])

        Args:
            fields (``str``, ``list``): Name of columns you want to retrieve.

        """

        # Class Input > Output
        # >>> param = FieldsParam(['FieldOne', 'FieldTwo'])
        # >>> param.to_param_dict()
        # {'fields[]': ['FieldOne', 'FieldTwo']}

        param_name = "fields"
        kwarg = param_name
```

(continues on next page)

(continued from previous page)

```
    class SortParam(_BaseObjectArrayParam):
        """
        Sort Param

        Kwargs:
            ``sort=``

        Specifies how the records will be ordered. If you set the view
        parameter, the returned records in that view will be sorted by these
        fields.

        If sorting by multiple columns, column names can be passed as a list.
        Sorting Direction is ascending by default, but can be reversed by
        prefixing the column name with a minus sign ``-``, or passing
        ``COLUMN_NAME, DIRECTION`` tuples. Direction options
        are ``asc`` and ``desc``.

        Usage:

        >>> airtable.get(sort='ColumnA')

        Multiple Columns:

        >>> airtable.get(sort=['ColumnA', '-ColumnB'])

        Explicit Directions:

        >>> airtable.get(sort=[('ColumnA', 'asc'), ('ColumnB', 'desc')])

        Args:
            fields (``str``, ``list``): Name of columns and directions.

        """

        # Class Input > Output
        # >>> filter = SortParam([{'field': 'col', 'direction': 'asc'}])
        # >>> filter.to_param_dict()
        # {'sort[0]['field']: 'col', sort[0]['direction']: 'asc'}

        param_name = "sort"
        kwarg = param_name

        def __init__(self, value):
            # Wraps string into list to avoid string iteration
            if hasattr(value, "startswith"):
                value = [value]

            self.value = []
            direction = "asc"

            for item in value:
```

(continues on next page)

(continued from previous page)

```
                if not hasattr(item, "startswith"):
                    field_name, direction = item
                else:
                    if item.startswith("-"):
                        direction = "desc"
                        field_name = item[1:]
                    else:
                        field_name = item

                sort_param = {"field": field_name, "direction": direction}
                self.value.append(sort_param)

    @classmethod
    def _discover_params(cls):
        """
        Returns a dict where filter keyword is key, and class is value.
        To handle param alias (maxRecords or max_records), both versions are
        added.
        """

        try:
            return cls.filters
        except AttributeError:
            filters = {}
            for param_class_name in dir(cls):
                param_class = getattr(cls, param_class_name)
                if hasattr(param_class, "kwarg"):
                    filters[param_class.kwarg] = param_class
                    filters[param_class.param_name] = param_class
            cls.filters = filters
        return cls.filters

    @classmethod
    def _get(cls, kwarg_name):
        """ Returns a Param Class Instance, by its kwarg or param name """
        param_classes = cls._discover_params()
        try:
            param_class = param_classes[kwarg_name]
        except KeyError:
            raise ValueError("invalid param keyword {}".format(kwarg_name))
        else:
            return param_class
```

2.3 Airtable Authentication

2.3.1 Overview

Authentication is handled by the *Airtable* class.

```
>>> airtable = Airtable(base_id, table_name, api_key)
```

Note: You can also use this class to handle authentication for you if you are making your own wrapper:

```
>>> auth = AirtableAuth(api_key)
>>> response = requests.get('https://api.airtable.com/v0/{base_id}/{table_name}',␣
↪auth=auth)
```

2.3.2 Authentication Class

class airtable.auth.**AirtableAuth**(*api_key*)

__init__(*api_key*)
Authentication used by Airtable Class

Parameters **api_key** (str) – Airtable API Key.

2.3.3 Source Code

```
import requests


class AirtableAuth(requests.auth.AuthBase):
    def __init__(self, api_key):
        """
        Authentication used by Airtable Class

        Args:
            api_key (``str``): Airtable API Key.
        """
        self.api_key = api_key

    def __call__(self, request):
        auth_token = {"Authorization": "Bearer {}".format(self.api_key)}
        request.headers.update(auth_token)
        return request
```

- genindex
- modindex

Version:

CHAPTER

THREE

RELEASE NOTES

CHAPTER

FOUR

QUESTIONS

Post them over in the project's Github Page

CHAPTER

FIVE

CONTRIBUTE

```
git clone git@github.com:gtalarico/airtable-python-wrapper.git
cd airtable-python-wrapper.git
python setup.py develop
python setup.py test
```

CHAPTER

SIX

LICENSE

MIT License

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

D

F

G

I

M

R

S

U