# Pypeline Documentation

***Release 0.2***

**Kyle Corbitt**

May 09, 2014

Contents


**1 Contents** **3**
- 1.1 Installation . . . 3
- 1.2 Quick Start . . . 3
- 1.3 API Reference . . . 6

**2 Links** **11**

Pypeline DB is designed to simplify the creation and management of datasets. It has a friendly and easy-to-master API backed by the power of LevelDB. This allows it to manage datasets too large to fit in RAM without sacrificing data access performance.

**Pypeline is great for:**

- Exploring data without eating all your RAM
- Transforming data with maps, filters and reductions
- Stopping you from losing or overwriting your data (unless you explicitly ask it to)

It's also easy to export a dataset from Pypeline to Pandas for further analysis.

CHAPTER 1

# Contents

## 1.1 Installation

### 1.1.1 Installing LevelDB

Pypeline DB relies on LevelDB and the Plyvel driver. Before installing Pypeline, you should make sure you have LevelDB installed and that the shared library is available. On Debian-based Linux distributions like Ubuntu this is as simple as `sudo apt-get install libleveldb1 libleveldb-dev`.

Additionally, make sure that your system is capable of installing Python modules with C extensions. On Ubuntu this can be accomplished with `sudo apt-get install python-dev`.

Further instructions can be found in the Plyvel docs.

### 1.1.2 Installing Pypeline

The simplest way of installing this package is with pip. Simply run `pip install pypeline-db`.

Alternatively, you can download the source and run `python setup.py install`

Check to make sure the package was properly installed by running `python -c 'import pypeline'`. If no output is generated you're good to go.

## 1.2 Quick Start

### 1.2.1 The Basics

You only have to understand two concepts to get started with Pypeline DB: databases and collections. A *Database* is just a simple wrapper for a LevelDB database on disk, which is stored as a directory. A *collection* is a group of documents in a database that are grouped together and can be worked with collectively. Ready? Let's get started.

```
>>> import pypeline
>>> db = pypeline.DB("test_database.pypeline", create_if_missing=True)
>>> collection = db.collection('collection_1')
>>> collection
pypeline.DB.Collection('collection_1')
```

This is pretty straightforward stuff. By calling pypeline.DB we create a new database to store our collections in (if you're opening a preexisting database the `create_if_missing` argument is unnecessary). We then create a new collection that we can refer to as 'collection_1'.

```
>>> db.collection('collection_2')
pypeline.DB.Collection('collection_2')
>>> db.collections()
[u'collection_1', u'collection_2']
```

The database keeps track of the collections within it. If you forget what your collection was called, it's easy to find it again just by calling `DB.collections()`.

### 1.2.2 Dealing with Collections

Of course, a collection is no good to us empty, so let's learn how to add something to it.

```
>>> for x in range(5):
...     collection.append(x)
...
>>> print [record for record in collection]
[0, 1, 2, 3, 4]
>>> collection.append_all([{'a': 'b'}, 'string', [1,2]])
>>> print [record for record in collection]
[0, 1, 2, 3, 4, {u'a': u'b'}, u'string', [1, 2]]
```

There are a couple of interesting things to see here. First of all, collections implement `append`. This works just like appending to a list, and writes your data straight to the database. And just like a list in Python, you can append several types of data. Dicts, lists, and primitives like ints and strings all work fine – anything that is JSON- serializable. `append_all` takes anything that is iterable, like a list, range, or even another collection, and appends every instance within it. (This is also a good way of combining multiple collections into a single one).

Secondly, Pypeline collections are *iterable*. That means all the familiar syntax like `for x in collection` will work just like you would expect.

```
>>> collection[0]
0
>>> collection.delete(0)
>>> print collection[0:2]
[1, 2]
>>> collection[0] = 5
>>> print collection[0]
5
```

Collections allow for the same familiar slice and indexing format as other python objects. This is very memory efficient because only the record(s) you request will be loaded into memory. Objects in a collection can also be deleted by index.

Collections can be copied:

```
>>> c3 = db.copy_collection('collection_1', 'collection_3')
>>> c3[:]
[5, 2, 3, 4, {u'a': u'b'}, u'string', [1, 2]]
>>> c3[0] = 1
>>> print c3[0]
1
>>> print collection[0]
5
```

Collection copies are "deep" copies, so changing a value in one collection won't affect it in the one it came from. There are two separate copies of the data on disk.

### 1.2.3 Reloading from Disk

One of the advantages of Pypeline is that you don't have to worry about data loss – everything you write to it is immediately backed up to disk. Let's reload our database just to test it out.

```
>>> db.close()
>>> del db, collection, c3
>>> db = pypeline.DB("test_database.pypeline")
>>> db.collections()
[u'collection_1', u'collection_2', u'collection_3']
>>> print [record for record in db.collection('collection_1')]
[5, 2, 3, 4, {u'a': u'b'}, u'string', [1, 2]]
```

As you can see, the database maintains its state with no problems.

### 1.2.4 Manipulating Data (Higher-order Functions)

Transforming data manually with operations such as `collection[0] = some_function(collection[0])` works just fine, but Pypeline provides more powerful and convenient ways of running operations on your collections. These are the operations `map`, `filter` and `reduce`, as well as the convenience operation `random_subset`. These work the same way as the built-in Python functions of the same name, except that they operate on a collection and allow you to set the output either to the same collection or to a new one. As always, the memory footprint of these operations is minimal because not all the data is loaded at once.

```
>>> c4 = db.collection('collection_4')
>>> c4.append_all(range(10))
>>> def map_add_one(x):
...     return x+1
...
>>> c5 = c4.map(map_add_one, 'collection_5')
>>> print c5[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> def filter_less_5(x):
...     return x < 5
...
>>> c5.filter(filter_less_5, None)
pypeline.DB.Collection('collection_5')
>>> print c5[:]
[1, 2, 3, 4]
>>> def reduce_sum(x, y):
...     return x+y
...
>>> c6 = c5.reduce(reduce_sum, 'collection_6')
>>> c6[:]
[10]
>>> c7 = c4.random_subset(5, 'collection_7')
>>> c7[:]
[0, 1, 4, 5, 7]
```

All of these functions take as an argument the name of the function to apply as well as the name of the collection to write the results to. The destination collection *will be overwritten* by these operations, so it's best to choose a new name and then append it to an existing collection if that's what you'd like to do. If the collection given is ‘‘None‘‘(as in the Filter example) the current collection will be overwritten.

With this introduction you're now ready to get started using Pypeline! If you have further questions, be sure to check the *API Reference* docs or open an issue on the project Github.

### 1.2.5 Importing into Pandas

Pandas is an invaluable tool for data analysis, and exporting data from Pypeline to Pandas is easy. Because Pypeline makes no assumptions about the format of your data, this requires a bit of manual glue to get right. An easy approach that does not require loading all the data into RAM is creating a temporary CSV file to act as a go-between.

```
>>> import pandas, os, tempfile, csv
>>> c8 = db.collection('collection_8')
>>> csv_file = tempfile.NamedTemporaryFile(delete=False)
>>> for x in range(5):
...     c8.append([x, x+1])
>>> writer = csv.DictWriter(csv_file, fieldnames=['first', 'second'])
>>> writer.writeheader()
>>> for record in c8:
...     writer.writerow({'first': record[0], 'second': record[1]})
>>> csv_file.close()
>>> csv_path = csv_file.name
>>> dataframe = pandas.io.parsers.read_csv(csv_path)
>>> print dataframe
   first  second
0      0       1
1      1       2
2      2       3
3      3       4
4      4       5
>>> os.remove(csv_path)
```

This snippet creates a temporary file that we'll use to store our data as CSV, a format that pandas can import from. We then create our "data," which is just a silly example in this case, and insert it into the collection. Using Python's built-in CSV utilities we then open the file and save our collection to the CSV file row by row. Finally, we close the file, import it into pandas, and delete it.

The dataframe now contains all the data from our collection and is ready for further analysis.

## 1.3 API Reference

This documents the current version of the Pypeline DB API.

### 1.3.1 pypeline.DB

**class** `pypeline.`**`DB`**(*database_path*, ***kwargs*)

The pypeline LevelDB database. This class contains collections and provides some system-level organization.

All arguments beyond the database path are passed directly to the underlying plyvel DB constructor. More details can be found at https://plyvel.readthedocs.org/en/latest/api.html#DB

Arguments: *database_path* – The path to the folder for database storage

**`collection`**(*collection_name*, *reset_collection=False*, *create_if_missing=True*, *error_if_exists=False*)

Returns the collection stored at *collection_name*, or creates it if it doesn't exist.

Arguments:
`collection_name` – the name of the collection to return

Keyword arguments:
`reset_collection` – when True any existant data in the collection is deleted before it is returned
`create_if_missing` – when False a ValueError is raised if the collection doesn't exist
`error_if_exists` – When True a ValueError is raised if the collection already exists

**collections**()
Returns a list of keys of collections contained in the database.

**copy_collection**(*old_collection*, *new_collection*, *start=None*, *end=None*, ***kwargs*)
Copies all instances in the old_collection into the new_collection

Arguments:
`old_collection` – The string name of the old collection
`new_collection` – The string name of the new collection

Keyword arguments:
`start` – (Optional) The index to begin copying from old_collection
`end` – (Optional) The index to end copying from old_collection
`create_if_missing` – when False a ValueError is raised if the new collection doesn't exist (default: True)
`error_if_exists` – When True a ValueError is raised if the collection already exists (default: False)

**delete**(*collection_name*)
Deletes a collection.

Arguments:
`collection_name` – the name of the collection to delete

**close**()
Closes the database.

**open**()
Opens the database.

### 1.3.2 pypeline.Collection

**class** `pypeline.`**Collection**(*database*, *items_set*, *name*)
A collection of records stored in a database

This class should never be instantiated directly. Use the `DB.collection()` method instead

**append**(*record*)
Appends a single record.

Arguments:
`record` – Any JSON-serializable python object (dicts, lists, ints, strings, etc.)

**refresh**()
Reloads the collection from the database.

**delete**(*index*)
Deletes an item from the collection.

Arguments:
`index` – Index of the item to be deleted.

**delete_all**()
Deletes all items in the collection

**append_all**(*iterable*)
Appends every item in the iterable to the collection

**map**(*function*, *new_collection*, ***kwargs*)
Maps a collection to a new collection with a provided function.

Arguments:
`function` – The function used for mapping.
`new_collection` – The name of the collection to insert the new values into. Any existing values will be deleted. If `None`, values are mapped to the same collection.

Keyword arguments:
`create_if_missing` – when False a ValueError is raised if the new collection doesn't exist
`error_if_exists` – When True a ValueError is raised if the new collection already exists

**filter**(*function*, *new_collection*, ***kwargs*)
Filters a collection into a new collection with a given function.

Arguments:
`function` – The function used for filtering.
`new_collection` – The name of the collection to insert the new values into. Any existing values will be deleted. If `None`, values are filtered in the same collection.

Keyword arguments:
`create_if_missing` – when False a ValueError is raised if the new collection doesn't exist
`error_if_exists` – When True a ValueError is raised if the new collection already exists

**reduce**(*function*, *new_collection*, *initializer=None*, ***kwargs*)
Reduces a collection into a new collection with a given function.

Arguments:

`function` – The function used for reducing.

`new_collection` – The name of the collection to insert the new value into. Any existing values will be deleted. If `None`, the current collection is replaced with the reduction output.

Keyword arguments:

`create_if_missing` – when False a ValueError is raised if the new collection doesn't exist

`error_if_exists` – When True a ValueError is raised if the new collection already exists

**random_subset**(*number*, *new_collection*, ***kwargs*)

Produces a random subset of a given collection and inserts it into a new collection.

Arguments:

`new_collection` – The name of the collection to insert the new values into. Any existing values will be deleted. If *None*, the subset is stored to the current collection.

Keyword arguments:

`create_if_missing` – when False a ValueError is raised if the new collection doesn't exist

`error_if_exists` – When True a ValueError is raised if the new collection already exists

**iterator**(*start=None*, *end=None*)

Returns a collection iterator

CHAPTER 2

# Links

- Source
- PyPi Package
- Docs
- Blog Post