Southern New Hampshire University

CS-340: Assignment 5-1: Writeup to Project 1

Prepared on: October 5, 2025

Prepared for: Prof. Jeff H. Sanford

Prepared by: Alexander Ahmann

Contents

1 Background

Two fictious entities, *Global Rain* and *Grazioso Salvare*, are presented to give a motivating problem for students of client-server software development.[1] Global Rain is described as "a software engineering company that specializes in custom software design and development," and Grazioso Salvare is described as an organisation that trains rescue animals.

This writeup concerns itself with myself assuming the role of lead developer for Global Rain and solving problems for Grazioso Salvare.[2] In particular, the latter has defined an objective of training dogs to rescue people and other animals in dangerous conditions. A sub-problem for Grazioso Salvare is to identify the "ideal dog" that can be trained for such rescue missions. A component to the solution to the problem of identifying these "ideal dogs" involves data modelling and analysis.[3]

The problem to be solved involves identifying, specifically, the features that make for the ideal rescue dog. Two examples of features that can help narrow down ideal rescue dogs are age and breed.[4] Such a problem is best approached with a computer and information system: a database server is used to store documentation and records of individual dogs, and a client-side utility is used to access the database, and filter and process the data.

2 Problem Statement

This writeup documents my solution to *Project 1*, which involves the setup and configuration of a *MongoDB* non-relational database management system (DBMS), and the creation of a Python module that interfaces with the aforementioned database.

A general outline of the problems to be solved involve: inventing a "Python CRUD module" that acts as a reusable library that interfaces with *Grazioso Salvare*'s MongoDB instance to perform *create, read, update*, and *delete* (CRUD) operations, to test this library's functionality with a basic test harness implemented as a *Jupyter Notebook*,[5] and to produce basic documentation in the form of a `README` file that briefly describes the newly invented "Python CRUD module."

[1] This hypothetical scenario is described in the "Scenario," of CS-340 (n.d., §2).

[2] This is a paraphrase of the "Scenario" in CS-340 (n.d., §2).

[3] The dataset to be analysed has been supplied by Animal Services (2016).

[4] As CS-340 (n.d., §2) notes, dogs no older than 2 years of age, and certain breeds of dogs, are more likely to express success in missions in environments involving "water, mountain, or wilderness" rescue, or locating people based on their scent.

[5] See https://jupyter.org/about

2.1 Learning Objectives

This project is, of course, a fictious motivating problem for educational purposes. In this writeup, I intend to demonstrate the following competencies: "*[application of] database systems concepts and principles to develop a client/server application*" and "*[creating] a database that can interface with client-side code.*"[6] I also intend to demonstrate competency in writing documentation.

3 Solutions, Part 1: Review of Previous Work

The first two tasks reminds the student to setup the needed infrastructure and to have a basic Python "CRUD class" developed from previous assignments, and this section serves as a review of the relevant stuff in them.

I have already worked out solutions to tasks 1 and 2 in previous assignments.[7] I have imported the Animal Services (2016) into the MongoDB instance with the `mongoimport` utility, created various indexes to optimize data modelling queries, created a new user account specifically to work with the dataset that I am concerned with modelling, and began working on a Python module to perform *create*, *read*, *update*, and *delete* (CRUD) operations against the MongoDB instance.

3.1 Task 1: Importing the CSV dataset into a MongoDB instance, and reformatting it into a BSON document

> "The Austin Animal Center Outcomes data set is pre-loaded in Codio as `aac_shelter_outcomes.csv`. In the Module Three milestone, you imported the data as `aac` into MongoDB by *inserting a CSV file using the appropriate MongoDB import tool.* The data set is located in the `datasets` folder in Codio. Complete the import using the `mongoimport` tool, and take screenshots of both the import command and its execution. You will include these screenshots in your README file later." —CS-340 (n.d., §3.1)

I have already completed this step in a previous assignment (Ahmann, 2025a), but would still like to review what I did to import the dataset, and

[6] Paraphrased from the rubric's "Competencies" section (CS-340, n.d., §1).

[7] See Ahmann (2025a,b) for an more comprehensive discussion; I will quote specific parts of these writeups going further into this section.

what resulted. I used the exact command depicted by plate 1 to import the CSV dataset and reformat its entries as BSON documents.

Plate 1: Variant of `mongoimport` utility used

```
mongoimport --db aac
  --collection animals
  --type csv
  --file ./aac_shelter_outcomes.csv
  --headerline
```

The `--db aac` argument instructs `mongoimport` to import the data into the `aac` database, the `--collection animals` instructs to import into the `animals` collection, and `--file ./aac_shelter_outcomes.csv` instructs `mongoimport` to read from the `aac_shelter_outcomes.csv`. Furthermore, the `--type csv` argument tells `mongoimport` that the input file will be a *comma seperated value* dataset, and the `--headerline` flag instructs the tool to treat the first row as a list of features for each of the new documents.

After running the `mongoimport` command with the given flags, ten-thousand (10,000) documents was added to the `aac.animals` collection from the given CSV file. These CSV-to-BSON documents will be the subject of basic data modelling and analysis.

3.2 Task 2: The Implementation of Role-Based Access Controls in the MongoDB instance

For the purposes of cybersecurity and keeping the *MongoDB* instance organised, *a role-based access control* (RoBAC) approach will be used to keep it working properly. I will create a new user account on the MongoDB instance and give it only read-write access to the `aac` database. I intend to store the user credentials in the `admin` database, and switched to it by first issuing the command `use admin`. The exact command that I used to create the new user is depicted in plate 2.

Plate 2: MongoDB query used to create `aacuser`

```
db.createUser({
  user: "aacuser",
  pwd: passwordPrompt(),
  roles: [ { role: "readWrite", db: "aac" } ]
})
```

The `user` field specifies the username, which I set to `aacuser` and the `roles` field specifies the kinds of permissions to grant the new user, and to which databases: which in this case are `readWrite` and `aac`, respectively. To securely set the password, I used the `passwordPrompt()` function to grab the password from a terminal input (masked by asterisks), as opposed to specifying it in a command. When setting up the `aacuser`, I set the password to `WingsofRedemption`.

Figure 1 shows the databases in the MongoDB instance, including the `admin` database, and figure 2 shows the execution of the query that adds the `aacuser` into the system, with appropriate privileges and password request through the `passwordPrompt()`.

4 Solutions, Part 2: The Current Task at Hand

The main work that I did for this module was improving on the `read()` and `create()` functions of the Python CRUD module, and creating the `update()` and `delete()` methods for said CRUD module. I also created made changes to the given test harness, and improved upon existing tests, in the form of a Jupyter Notebook. Other work that I did includes rewriting existing portions of code, created either by me or given by the course as "starter code," and writing up documentation in the form of a *README* file.[8]

4.1 Task 3: Further Implementing CRUD Functionality into the Custom Python MongoDB module

This first project involves the invention of a Python CRUD module that consists of four basic functions: a `read()` function, a `create()` function, an

[8] Along with this more detailed writeup, of course.

`update()` function, and a `delete()` function:

- `read()`: queries the collection in question with a given filter, and returns a `list` of documents that meet the filter's criteria.
- `create()`: adds a new document to the collection based on a dictionary that will act as a "field–>value" data structure to be stored as a *BSON document*. This function returns a `bool` value— specifically a `True` if the procedure worked out correctly, or a `False` if there was a failure in the procedure.
- `update()`: identifies documents in a collection on the basis of a filter, and then changes their values based on a given key/value pair. Returns an `int` value of the number of documents that was affected.
- `delete()`: deletes documents based on a criteria specified by a filter. Returns an `int` value specifying the number of documents that were affected.

Plate 3.1 depicts the "skeleton" of the Python CRUD module.

Plate 3.1: Basic "Skeleton" for CRUD Module

```
1.  from pymongo import MongoClient
2.  from bson.objectid import ObjectId

3.  class AnimalShelter(object):
4.      def __init__(self):

5.          USER = 'aacuser'
6.          PASS = 'WingsofRedemption'
7.          HOST = 'localhost'
8.          PORT = 27017
9.          DB = 'aac'
10.         COL = 'animals'
```

The `MongoClient` and `ObjectId` are imported,[9] with the latter being a Python implementation of MongoDB's "ObjectId" data type used for storing unique identifiers for documents in a collection. Such an `ObjectId` will

[9] Plate 3.1, lines 1-2

be useful when identifying specific documents that might share common values in their respective fields. Next, an `AnimalShelter` class is declared,[10] which exploits the notion of inheritance to derive basic MongoDB commands from the `MongoClient`. Or, the `AnimalShelter` class is an extension of the `MongoClient`, which introduces additional functions "tailored" for interfacing with the `aac` database.

Plate 3.1: Basic "Skeleton" for CRUD Module (cont.)

```
11.         self.client = MongoClient
    ('mongodb://%s:%s@%s:%d' % (USER,PASS,HOST,PORT))
12.         self.database = self.client['%s' % (DB)]
13.         self.collection = self.database['%s' % (COL)]

14.     def create(self, data: dict) -> bool:
15.         pass

16.     def read(self, data: dict) -> list:
17.         pass

18.     def update(self, query: dict, data:
    dict) -> int:
19.         pass

20.     def delete(self, query: dict) -> int:
21.         pass
```

The class is then initialized[11] to get configuration options for a username,[12] a password,[13] a host to connect to,[14] a port number that is listening for a MongoDB connection,[15] the database to connect to,[16] and the

[10]Plate 3.1, line 3.
[11]Plate 3.1, lines 4-13
[12]Plate 3.1, Line 5, the username `USER` in this case is `aacuser`.
[13]Plate 3.1, Line 6, the password `PASS` in this case is `WingsofRedemption`.
[14]Plate 3.1, Line 7, the `HOST` in this the `localhost`.
[15]Plate 3.1, Line 8, the `PORT` in this case is `27017`.
[16]Plate 3.1, Line 9, the database `DB` in this case is `aac`.

document collection to work with.[17] After specifiying the configurations, `AnimalShelter`'s `__init__` procedure then connects to the database,[18] and then configures the current database to be used and the collection to interact with.[19] After this initialization process, the `create()`, `read()`, `update()`, and `delete()` are defined,[20] and elaborating on their functionality is the "main topic" of this writeup.

I have already developed `read() and create()` functions, though I did make slight changes to them— more so with the latter. Plate 3.2.1 depicts `create()`'s code.

Plate 3.2.1: Refined solution to `create()`

```
1.  def create(self, data: dict) -> bool:
2.      """
3.      A function that inserts a document into a
    specified
4.        MongoDB database and collection
5.      Args:
6.        data (dictionary): the actual dictionary,
    which is to be
7.          inserted as a BSON document.
8.      Returns:
9.        bool: A "True" if the operation was
    successful, otherwise a "False" if the
10. operation was not successful.
11.     """
```

I would like to direct the reader's attention to two feature of good Python function creation: the use of *type hints* and *document strings*. Type hints act as documentation, unique to the Python language,[21] for developers telling them to give a specific kind of Python data type to the function.[22] Document

[17]Plate 3.1, Line 10, the collection in this case is `animals`.

[18]Plate 3.1, Line 11.

[19]Plate 3.1, Lines 11 and 12.

[20]Plate 3.1, `create()` on lines 14 and 15, `read()` on lines 16 and 17, `update()` on lines 18 and 19, and `delete()` on lines 20-21.

[21]From my cursory research, type hints seem to be specific to Python. Though it is possible that there exists other languages that implement type hints.

[22]See Python 3.13.7 Documentation (n.d.). *typing– Support for type hints* Retrieved on Oct. 4, 2025 from: https://docs.python.org/3/library/typing.html

strings further assist developers who want to use the *Python CRUD Module* by giving a brief description of the function's purpose, and its needed inputs and given outputs.[23] They are typically placed just below the declaration of a new `class` or `def` objects, enclosed between two triple-quotes, one triple-quote at the beginning of the document string, and another at its end.

Plate 3.2.1: Refined solution to `create()` (cont.)

```
12.     try:
13.         if data is None:
14.             raise Exception("The data type should
     not be 'None'")
15.         elif not isinstance(data, dict):
16.             raise Exception("The data type should
     be a dictionary")
17.         else:
18.             obj_id = ObjectId()
19.             document = {"_id":obj_id}
20.             document.update(data)
21.             self.database.animals
    .insert_one(document)
22.             return True
23.     except Exception as e:
24.         print("Exception raised: {0}".format(e))
25.         return False
26.    return False
```

The `create()` function takes an argument called `data`,[24] which is a Python dictionary, and returns a `bool` value resulting in `True` if the insert operation completed successfully, and a `False` if the insert operation failed.[25] The procedure by which to insert, or "`create()`," a new BSON document into the `animals.aac` collection begins with a set up of an exception handling via the `try/except` block.[26] In the case of an exception, an error message

[23] "Document Strings" are usually called "Docstrings" in a Python context; also see the Python PEP (n.d.). *PEP 257 — Docstring Conventions.* Retrieved on Oct. 4, 2025 from: https://peps.python.org/pep-0257/

[24] Plate 3.2.1, Line 1.

[25] Plate 3.2.1, Line 26.

[26] Plate 3.2.1, Lines 12 and 23.

is printed out into the terminal and a `False` boolean value is returned.[27] When this function executes, conditional logic is employed to make sure that the `data` variable is not set to `None`.[28] If it is, an exception is raised. Next, such conditional logic is then used to check if the `data` variable is a Python dictionary.[29] If `data` is another kind of data type, an exception is raised.[30]

If the given `data` is indeed a Python dictionary, the insert procedure starts:[31] a new `ObjectId` is created, and then stored in the `obj_id`, another newly defined dictionary. Next, a new dictionary called `document` is defined with the newly created `obj_id`. Finally, the `document` is inserted into the `aac.animals` collection via the `self.database.animals.insert_one (document)` command. If it was executed without any error, `create()` will return a `True`.[32] Otherwise, if an exception was raised, it will return a `False`.[33]

The next function that I implemented is the `read()` function, as depicted in plate 3.2.2.[34] Like the `create()` function, the `read()` function has type hints to assist developers on properly using the function. Furthermore, the same conditional logic is used to ensure that given parameters are with the appropriate data type.

---

[27] Plate 3.2.1, Lines 24 and 25.

[28] Plate 3.2.1, Lines 13–22.

[29] Plate 3.2.1, Lines 15 and 16.

[30] This is done because of Python's weak-typing design: Programming languages typically raise an exception if an incorrect data type is passed into a function's argument. But Python "breaks this mold" by allowing for any kind of data type to be passed to its functions. This allows for more flexability, but can also lead to unintended consequences in runtime, so checks for proper typing needs to be implemented in order to reduce the risk of such stuff happening.

[31] Plate 3.2.1, Lines 18–21.

[32] Plate 3.2.1, Line 22.

[33] Plate 3.2.1, Line 26.

[34] `read()` has a document string, but I stripped it for the sake of succinctness. Consult with the supplementary codebase to view the full, unabridged code.

Plate 3.2.2: Refined solution to `read()`

```
1.  def read(self, query: dict) -> list:
2.      results = []
3.      try:
4.          if not isinstance(query, dict):
5.              raise Exception
    ("'query' should be a dictionary")
6.          else:
7.              query_results = self.collection
    .find(query)
8.              results.extend([doc for doc
    in query_results])
9.              return results
10.     except Exception as e:
11.         print("Exception raised: {0}".format(e))
12.     return results
```

In particular, the given `read()` function takes only one parameter, the `query`, and expects it to be a dictionary.[35] I first defined a Python `list` called `results`,[36] which is to be returned after the function's execution.[37] Then, exception handling is implemented via a `try/except` block,[38] and begins to process the `query` parameter and make it work with PyMongo's `find()` function.[39] Assuming that that `query` is a dictionary, this script will execute the `self.collection.find()` command parameterized by the given `query`. The selected documents are stored in `query_results`,[40] then appends them to the aforementioned defined `results`,[41] and finally returns said `results` list with the resulting documents.[42] If an exception is indeed raised, it will simply return an empty `results` list and print out an error message.[43]

The worked solutions to `create()` and `read()` functions have been refined from previous attempts. In engineering, regardless of mechanical, electron-

[35]Plate 3.2.2, Line 1.
[36]Plate 3.2.2, Line 2.
[37]Plate 3.2.2, Line 12.
[38]Plate 3.2.2, Lines 3 and 10.
[39]In the `try` half of the exception handler.
[40]Plate 3.2.2, Line 10.
[41]Plate 3.2.2, Line 8
[42]Plate 3.2.2, Line 9.
[43]Plate 3.2.2, Lines 11 and 12.

ical, or software, it is good to identify defects in a technical solution, and then eliminate them and further refine said solution to work more efficiently with less errors. I have done so by removing redundant code and adding document strings to assist developers interested in using the *Python CRUD Module*.[44]

Plate 3.2.3: Worked solution to `update()`

```
1.  def update(self, query: dict, data: dict) -> int:
2.      total_affected = 0
3.      try:
4.          if not isinstance(query, dict):
5.              raise Exception
    ("parameter 'query' should be a dictionary.")
6.          elif not isinstance(data, dict):
7.              raise Exception
    ("parameter 'data' should be a dictionary.")
8.          result = self.collection.update_many(
9.              query, { "$set": data }
10.         )
11.         total_affected += result.modified_count
12.     except Exception as e:
13.         print("Exception raised: {0}".format(e))
14.     return total_affected
```

The next subtask is to create the `update()` and `delete()` functions *ex nihilo*. Plate 3.2.3 depicts the code for the `update()` function.[45] Like the previously refined `create()` and `read()` functions, this one uses document strings and type hints to inform the developers on what the function does, and how to properly use it. In particular, `update()` takes two parameters: `query` and `data`, both of which are dictionaries, and returns an integer.[46]

[44]See Ahmann (2025b, Plates 4.1 and 4.2) for previous versions of the `create()` and `read()` functions.

[45]Note that extraneous stuff like the document string and comments have been eliminated from this plate, for the sake of succinctness.

[46]See the function definition in plate 3.2.3, line 1.

Plate 3.2.4: Worked solution to `delete()`

```
1.  def delete(self, query: dict) -> int:
2.      total_affected = 0
3.      try:
4.          if not isinstance(query, dict):
5.              raise Exception
    ("parameter 'query' should be a dictionary.")
6.          result = self.collection.delete_many(query)
7.          total_affected += result.deleted_count
8.      except Exception as e:
9.          print("Exception raised: {0}".format(e))
10.     return total_affected
```

The `query` dictionary is a filter criteria to work out which documents to apply the update procedure to, and the `data` dictionary specifies the fields and their respective values by which to update the pre-existing fields with.[47] `update()` returns an integer specifying the count of documents that were affected by the update procedure.

`update()` starts off by defining a `total_affected` variable,[48] an integer set to zero— which will be returned after the function is done executing.[49] Like the `create()` and `read()` functions, the main logic to work out the return value is enclosed in a `try/except` block.[50] An error message is printed out if an exception is raised.[51] The function first confirms if the `query` and `data` parameters are dictionaries.[52] If they are not, an exception is raised.[53]

Otherwise, PyMongo's `update_many()` command is executed, and various outcomes are stored in the `result` variable.[54] The `query` variable is passed as the `update_many()`'s first argument as a filter criteria, and `data` is passed into another dictionary with the `key/value` pair of `$set/data`.

Next, the `total_affected` variable is incremented by `result.modified_count`,[55] or the count of documents that have been affected by the update

[47] Or perhaps add new fields if they do not exist? I should look into that.
[48] Plate 3.2.3, Line 2.
[49] Plate 3.2.3, Line 14.
[50] Plate 3.2.3, Lines 3 and 12.
[51] Plate 3.2.3, Line 13.
[52] Plate 3.2.4, Lines 5 and 6.
[53] Specifically on line 6 of plate 3.2.4.
[54] Plate 3.2.3, Line 8.
[55] Plate 3.2.3, Line 11.

transaction. Assuming that the procedure executed without a raised exception, and that there existed at least one document that met the criteria defined in `query`, a non-zero integer will be returned.[56] Otherwise, the integer zero will be returned.

Finally, the creation of a `delete()` function is in order. Plate 3.2.4 depicts an abridged version of my implementation of `delete()`. Like previous CRUD functions, this function uses type hints, a document string,[57] and exception handling to mitigate issues caused by runtime errors. In particular, the function takes in one parameter, a Python dictionary called `query`,[58] and outputs an integer.[59] The function begins by defining a variable called `total_affected`,[60] which is initially set to zero, and is to be returned after the function's execution.

As usual, exception handling is used to mitigate potential runtime errors. A `try/except` block is used to handle them,[61] with an error message printed should an exception be raised.[62] The main logic to delete records and calculate the resulting value for `total_affected` is done under the `try` block. An exception is raised if the `query` is not a dictionary,[63] and if the `query` is an actual dictionary, PyMongo's `delete_many()` command is executed, and then stored into the `result` variable.[64]

The `result.deleted_count`, which is the number of documents that this database transaction affected, is appended onto the `total_affected` variable.[65] Assuming that there were no exceptions raised, and that the `query` filter was able to match at least one document to its criteria, then a non-zero integer will be returned. Otherwise, a zero integer will be returned.

...

This completes the current work needed to refine and further develop the *Python CRUD Module*. This module can now, for the most part, be used by any developer interested in accessing the `aac` database — and the `animals` collection in particular.

[56] Plate 3.2.3, Line 14.

[57] I stripped out the document string for the sake of succinctness; see the supplementary codebase for a full listing.

[58] 3.2.4, Line 1

[59] Plate 3.2.4, Line 10.

[60] Plate 3.2.4, Line 10

[61] Plate 3.2.4, Lines 3 and 8

[62] Plate 3.2.4, Line 9.

[63] Plate 3.2.4, Lines 4 and 5.

[64] Plate 3.2.4, Line 6

[65] Plate 3.2.4, Line 7.

4.2 Task 4: Testing the CRUD functionality in the custom module, and further codebase refinements

For the purposes of quality assessment, I have devised a test harness consisting of a simple set of test cases, implemented in the *Jupyter Notebook*. A previous test harness written with the Jupyter Notebook in a previous assignment (Ahmann, 2025b), and the work done in this module improves upon the previously done work with the test harness.

I would be redundant to describe the basic testing process for each of the CRUD class functions. So, instead, I will just outline the general procedure for the testing procedure. Plate 4 depicts an algorithmic procedure as a representation of the general process for testing "CRUD class" functions.

Plate 4: Generic Testing Procedure

test_cases ← dictionary of hard-coded test cases
for $key, value = val \in$ test_cases **do**
    `results` ← `CRUD.function`(val[p1],val[p2], ... val[pN])
    Print "Results for { `key` }: { `results` }"
end for

The general testing method is to define a Python dictionary of `key`/`value` pairs — where the label of the test case is the key, and the test cases, which could be a single variable, or a `list` of variables, acting as parameters for the function under testing. Then, a `for`-loop is used to execute each test case pushing the `val` parameter into the function being tested, and printing out results. The analyst is to manually confirm whether-or-not the tests were successful.

I have executed the Jupyter Notebook, and after many iterations when I ran into errors, I was able to demonstrate that the Python CRUD Module is functioning *at least at a basic standard*. I have attached screenshots of the Jupyter Notebook tests that I ran to this writeup to demonstrate a basic standard of a working "CRUD Class."

Figure 3 shows the setup of the JupyterLab notebook: importing needed libraries, instantiating the `AnimalShelter` class from the newly invented Python CRUD Module, setting up constants and functions for handling the "nitty gritty" work of making the tests readable, and setting up the test case for the first function: the `read()`. Figures 4 and 5 show the results for two `read()` test cases, and demonstrate that they work properly. Figures 5—9

show the setup and results of test cases for the `create()` functions: they were all successful. Figure 11 shows a similar setup with test cases for the `update()` function, with success. And finally, figure 12 shows the test cases for the `delete()` function, its results, and report success.

5 Discussion

When making the Python CRUD class, I implemented standard coding procedures. I introduced proper naming for functions and variables, exception handling, and introducing comments to explain certain snippets of code.[66] I also introduced type hints and document strings to hopefully "raise the bar" in quality coding.[67] The basic procedure that is common to all of the CRUD class functions is that they take in a `query` filter in the form of a `dict` data-type, perform a database transaction with the `query`,[68] store the transaction results in a variable, and then return the variable to its parent function.

While this Python CRUD module meets a basic level of competence, further work can be done in order to improve it. One possible improvement could be to use a *linting* utility to scan the CRUD module's source code and identify defects that the software engineering team did not identify.[69] After the linter identifies bad coding practices, the codebase can be refactored and refined. More unit tests can be written, and a unit testing framework should be employed to keep track of the successess and failures that came about from testing,[70] not just through a developer manually identifying failures. To improve the kinds of unit tests, a *software fuzzer* can be employed to devise test cases that are more likely to "break" an application. Software fuzzers are adept at identifying not just software crashes, but potential vulnerability vectors that a malicious hacker might use to take over the web application in question. They will definately ensure reliability *and even* security of a target

[66] As required by the project guidelines (CS-340, n.d.).

[67] I know type hints and document strings from experience; furthermore, I consulted these additional high quality coding practices from the following resource: Dhamane, G. (Aug. 7, 2025) *The Python Developer's Survival Kit— 15 Best Practices That Separate Pros from Beginners*. Level Up Coding. Retrieved on Oct. 4, 2025 from: https://levelup.gitconnected.com/the-python-developers-survival-kit-15-best-practices-that-separate-pros-from-beginners-7d87ba661814

[68] And possibly other parameters, if more information is needed.

[69] From cursory research, `pylint` appears to be promising: https://pylint.readthedocs.io/en/stable/

[70] From cursory research, the Python `unittest` framework looks promising: https://docs.python.org/3/library/unittest.html

web application.

6 Summary

In this project, the first phase of developing a data-driven web application has been accomplished. In particular, the following gives a final outline of the progress made:

- A *MongoDB* database management system (DBMS) has been setup, with proper role-based access controls to ensure system reliability and security.
- A CSV dataset was imported into the MongoDB instance, and converted from its tabular format to a BSON format.
- Further refinements were made to the already existing `create()` and `read()` functions, and new `update()` and `delete()` functions were created *ex nihilo*.
- Further refinements were made to the given Jupyter Notebook, which functions as a test harness for the Python CRUD module.
- While I was able to demonstrate a "CRUD class" that works under basic, laboratory controlled conditions, further work will be needed to refine the codebase to ensure maximum reliability and security— especially when deployed to computer systems where "harsh" phenomena is more likely to occur.

References

Ahmann, A. (2025a). *CS-340: Assignment 3-1: Module 3 Journal*. Homework Assignment.

Ahmann, A. (2025b). *CS-340: Assignment 4-1: Module 4 Journal*. Homework Assignment.

Animal Services (2016). Austin Animal Center Outcomes (Version 3.1). *City of Austin, Texas Open Data*. https://doi.org/10.26000/025.000001

CS-340 (n.d.). *Project One Guidelines and Rubric*.

Giamas, A. (2022). *Mastering MongoDB 6.x: Expert Techniques to Run High-volume and Fault-tolerant Database Solutions Using MongoDB 6.x*. Birmingham, UK: Packt Publishing. https://research.ebsco.com/linkprocessor/plink?id=a5bcc20e-3306-36b5-ad4f-0d0bd1f1567e

A Figures Depicting Screenshots that Demonstrate Task Completion

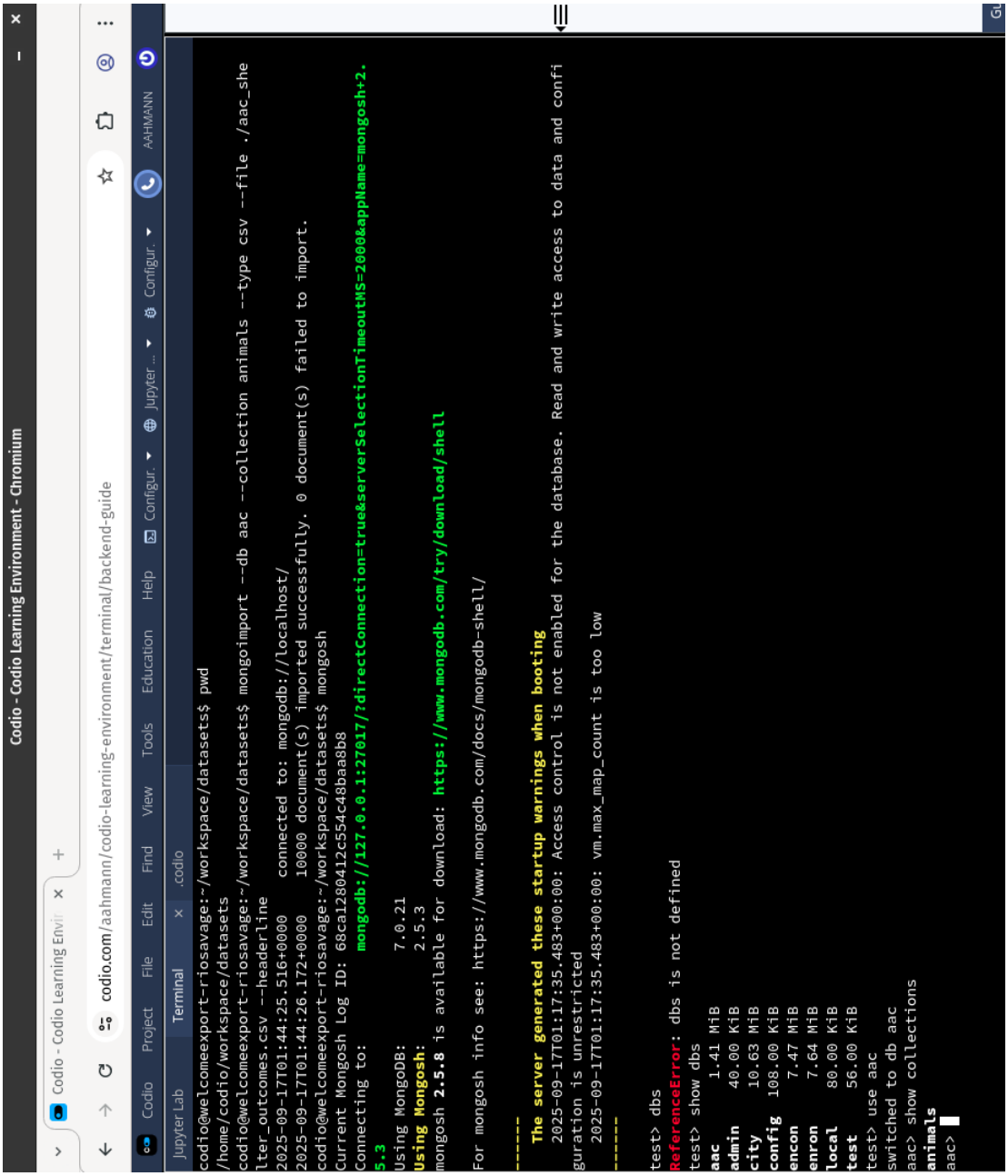
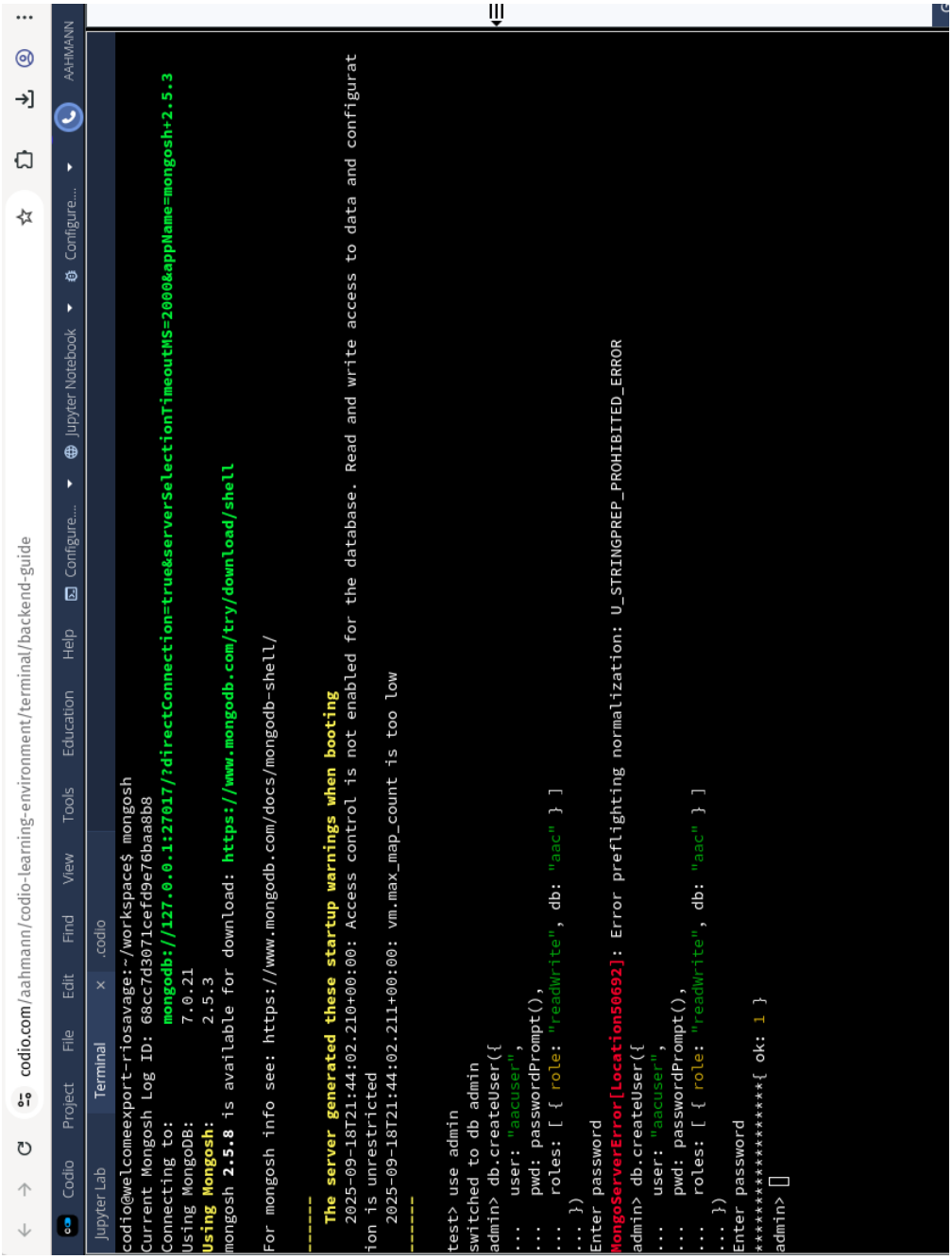


Figure 1:


codio.com/aahmann/codio-learning-environment/terminal/backend-guide
Codio
Project
File
Edit
Find
View
Tools
Education
Help
Configure...
Jupyter Notebook
Configure...
AAHMANN
Jupyter Lab
Terminal
.codio
codio@welcomeexport-riosavage:~/workspace$ mongosh
Current Mongosh Log ID: 68cc7d3071cefd9e76baa8b8
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.3
Using MongoDB: 7.0.21
Using Mongosh: 2.5.3
mongosh 2.5.8 is available for download: https://www.mongodb.com/try/download/shell
For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

The server generated these startup warnings when booting
2025-09-18T21:44:02.210+00:00: Access control is not enabled for the database. Read and write access to data and configurat
ion is unrestricted
2025-09-18T21:44:02.211+00:00: vm.max_map_count is too low
------
test> use admin
switched to db admin
admin> db.createUser({
... user: "aacuser",
... pwd: passwordPrompt(),
... roles: [{ role: "readWrite", db: "aac" }]
... })
Enter password
MongoServerError[Location50692]: Error preflighting normalization: U_STRINGPREP_PROHIBITED_ERROR
admin> db.createUser({
... user: "aacuser",
... pwd: passwordPrompt(),
... roles: [{ role: "readWrite", db: "aac" }]
... })
Enter password
**********{ ok: 1 }
admin>

Figure 2:

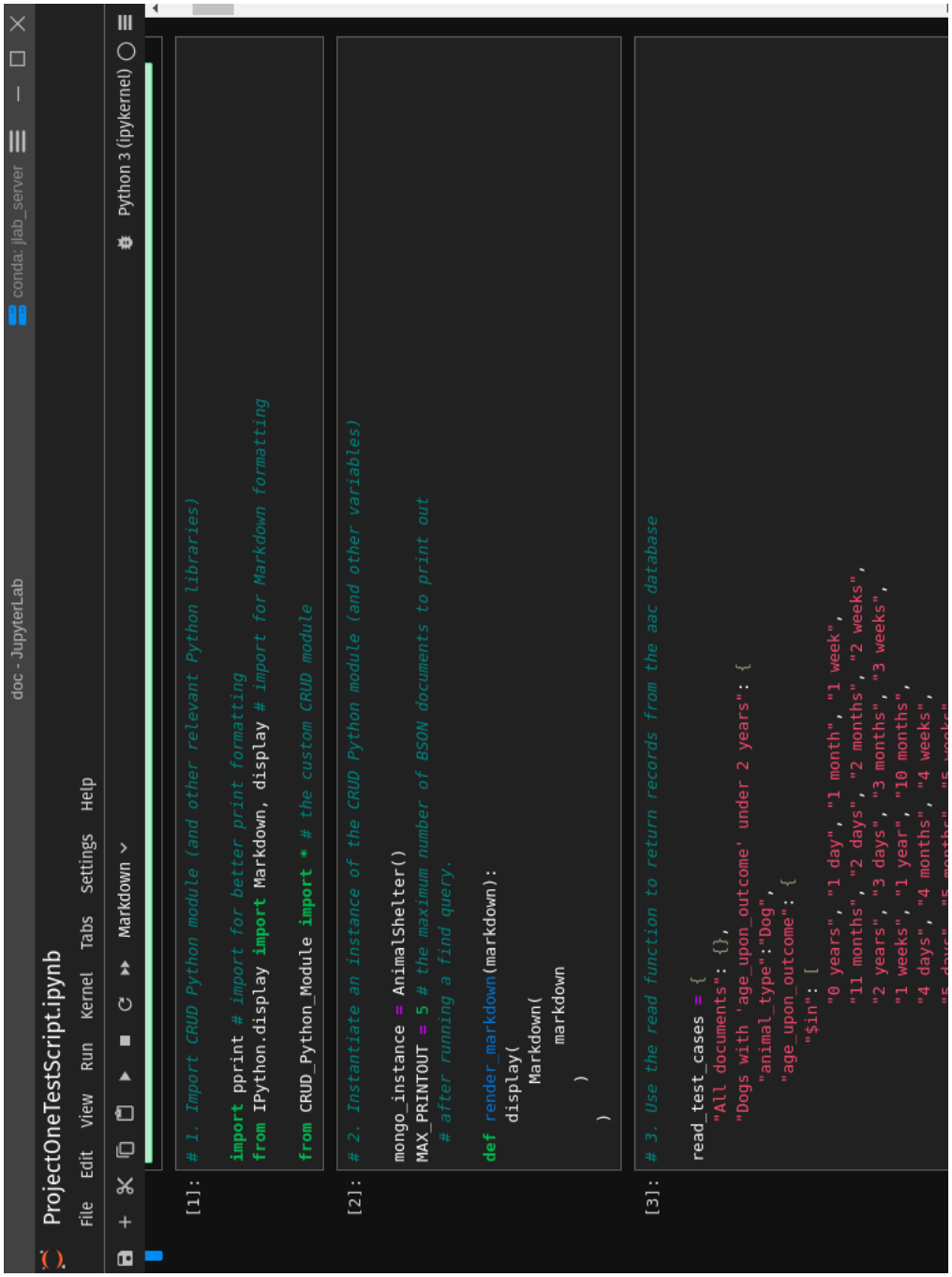

doc - JupyterLab
conda: jlab_server
ProjectOneTestScript.ipynb
File Edit View Run Kernel Tabs Settings Help
Markdown
Python 3 (ipykernel)
[1]: # 1. Import CRUD Python module (and other relevant Python libraries)
import pprint # import for better print formatting
from IPython.display import Markdown, display # import for Markdown formatting
from CRUD_Python_Module import * # the custom CRUD module
[2]: # 2. Instantiate an instance of the CRUD Python module (and other variables)
mongo_instance = AnimalShelter()
MAX_PRINTOUT = 5 # the maximum number of BSON documents to print out
# after running a find query.
def render_markdown(markdown):
display(
Markdown(
markdown
)
)
[3]: # 3. Use the read function to return records from the aac database
read_test_cases = {
"All documents": {},
"Dogs with 'age_upon_outcome' under 2 years": {
"animal_type":"Dog",
"age_upon_outcome": {
"$in": [
"0 years", "1 day", "1 month", "1 week",
"11 months", "2 days", "2 months", "2 weeks",
"2 years", "3 days", "3 months", "3 weeks",
"1 weeks", "1 year", "10 months",
"4 days", "4 months", "4 weeks",

Figure 3:

```
    else:
        pprint.pp(results) # print out all the documents (which
          # will have a count that is less than 5).
```

1. Test cases for `.read()` function:

Results for test case (read): "All documents":

Total documents for All documents: 10000

Sample output for All documents:

```
[{'_id': ObjectId('68ca1279e4eff538217d6190'),
  'rec_num': 1,
  'age_upon_outcome': '3 years',
  'animal_id': 'A746874',
  'animal_type': 'Cat',
  'breed': 'Domestic Shorthair Mix',
  'color': 'Black/White',
  'date_of_birth': '2014-04-10',
  'datetime': '2017-04-11 09:00:00',
  'monthyear': '2017-04-11T09:00:00',
  'name': '',
  'outcome_subtype': 'SCRP',
  'outcome_type': 'Transfer',
  'sex_upon_outcome': 'Neutered Male',
  'location_lat': 30.5066578739455,
  'location_long': -97.3408780722188,
  'age_upon_outcome_in_weeks': 156.767857142857},
 {'_id': ObjectId('68ca1279e4eff538217d6191'),
  'rec_num': 9,
  'age_upon_outcome': '3 years',
  'animal_id': 'A720214',
  'animal_type': 'Dog',
  'breed': 'Labrador Retriever Mix',
  'color': 'Red/White',
  'date_of_birth': '2013-02-04',
```

Figure 4:

Figure 5:

Results for test case (read): "Dogs with 'age_upon_outcome' under 2 years":

Total documents for Dogs with 'age_upon_outcome' under 2 years: 3687

Sample output for Dogs with 'age_upon_outcome' under 2 years:


```
[{'_id': ObjectId('68ca1279e4eff538217d6193'),
  'rec_num': 11,
  'age_upon_outcome': '1 year',
  'animal_id': 'A721199',
  'animal_type': 'Dog',
  'breed': 'Dachshund Wirehair Mix',
  'color': 'Tan/White',
  'date_of_birth': '2015-02-23',
  'datetime': '2016-02-27 17:49:00',
  'monthyear': '2016-02-27T17:49:00',
  'name': 'Belle',
  'outcome_subtype': '',
  'outcome_type': 'Adoption',
  'sex_upon_outcome': 'Spayed Female',
  'location_lat': 30.7290272761146,
  'location_long': -97.3753328216134,
  'age_upon_outcome_in_weeks': 52.8203373015873},
 {'_id': ObjectId('68ca1279e4eff538217d6194'),
  'rec_num': 12,
  'age_upon_outcome': '1 year',
  'animal_id': 'A664843',
  'animal_type': 'Dog',
  'breed': 'Pit Bull Mix',
  'color': 'Brown/White',
  'date_of_birth': '2013-06-09',
  'datetime': '2014-08-18 17:24:00',
  'monthyear': '2014-08-18T17:24:00',
  'name': 'Sherlock',
  'outcome_subtype': 'Partner',
  'outcome_type': 'Transfer',
  'sex_upon_outcome': 'Neutered Male',
  'location_lat': 30.4515549397366,
```

```
[4]: # 4. Use the create function to create a new record in the aac database

insert_test_cases = {

    "Test Case #1": {
        'tid': 'test_case_1',
        'rec_num': 20001,
        'age_upon_outcome': '1 year',
        'animal_id': 'TA736551',
        'animal_type': 'Dog',
        'breed': 'Labrador Retriever/Australian Cattle Dog',
        'color': 'Black',
        'date_of_birth': '2015-10-12',
        'datetime': '2016-11-27 18:00:00',
        'monthyear': '2016-11-27T18:00:00',
        'name': '*',
        'outcome_subtype': '',
        'outcome_type': 'Adoption',
        'sex_upon_outcome': 'Spayed Female',
        'location_lat': 30.4443212820182,
        'location_long': -97.7326980338793,
        'age_upon_outcome_in_weeks': 58.9642857142857
    },

    "Test Case #2": {
        "tid": 'test_case_2',
        'rec_num': 20002,
        'age_upon_outcome': '5 months',
        'animal_id': 'TA693288',
        'animal_type': 'Cat',
        'breed': 'Domestic Shorthair Mix',
        'color': 'Orange',
        'date_of_birth': '2013-09-28',
        'datetime': '2013-12-09 18:36:00',
        'monthyear': '2013-12-09T18:36:00',
```

Figure 6:

```
            'zip':90210
        }
    } # Regarding the next two test cases: the different fields in both
      # test cases demonstrate the flexability in what kinds of data that MongoDB
      # collections can store, and how fields can vary from document-to-document.
}

render_markdown("## 2.1. Test cases for ``.create()`` function:")
for k, v in insert_test_cases.items():
    render_markdown("### Test case (create): {0}".format(k))
    print("\nDocument to be inserted: {0}\n".format(v))

    successful = mongo_instance.create(v)
    if successful:
        render_markdown("__Successfully inserted new document.__")
    else:
        render_markdown("__Failed to insert new document.__")

test_documents_queries = {
    "Test Case #1": {"tid":"test_case_1"},
    "Test Case #2": {"tid":"test_case_2"},
    "Test Case #3": {"tid":"test_case_3"},
    "Test Case #4": {"tid":"test_case_4"},
    "Test Case #5": {"tid":"test_case_5"},
} # To avoid breaking the dataset, I will test the update /
  # delete functions against the newly created test cases

render_markdown("## 2.2. Contents of the test cases:")

for k, v in test_documents_queries.items():
    render_markdown("### Specifically, contents of: {0}".format(k))

    results = mongo_instance.read(v)
    pprint.pp(results)
```

Figure 7:

2.1. Test cases for `.create()` function:

Test case (create): Test Case #1

```
Document to be inserted: {'tid': 'test_case_1', 'rec_num': 20001, 'age_upon_outcome': '1 year', 'animal_id': 'TA73
6551', 'animal_type': 'Dog', 'breed': 'Labrador Retriever/Australian Cattle Dog', 'color': 'Black', 'date_of_birt
h': '2015-10-12', 'datetime': '2016-11-27 18:00:00', 'monthyear': '2016-11-27T18:00:00', 'name': '*', 'outcome_sub
type': '', 'outcome_type': 'Adoption', 'sex_upon_outcome': 'Spayed Female', 'location_lat': 30.4443212820182, 'loc
ation_long': -97.7326980338793, 'age_upon_outcome_in_weeks': 58.9642857142857}
```

Successfully inserted new document.

Test case (create): Test Case #2

```
Document to be inserted: {'tid': 'test_case_2', 'rec_num': 20002, 'age_upon_outcome': '5 months', 'animal_id': 'TA
693288', 'animal_type': 'Cat', 'breed': 'Domestic Shorthair Mix', 'color': 'Orange', 'date_of_birth': '2013-09-2
8', 'datetime': '2013-12-09 18:36:00', 'monthyear': '2013-12-09T18:36:00', 'name': '*Brain', 'outcome_subtype':
'', 'outcome_type': 'Adoption', 'sex_upon_outcome': 'Spayed Female', 'location_lat': 30.4527678292931, 'location_l
ong': -97.4620507167676, 'age_upon_outcome_in_weeks': 10.3964285714286}
```

Successfully inserted new document.

Test case (create): Test Case #3

```
Document to be inserted: {'tid': 'test_case_3', 'age_upon_outcome': '5 months', 'animal_id': 'TA693288', 'animal_t
ype': 'Cat', 'breed': 'Domestic Shorthair Mix', 'color': 'Orange', 'date_of_birth': '2013-09-28', 'datetime': '201
3-12-09 18:36:00', 'name': '*Moning', 'outcome_type': 'Adoption', 'sex_upon_outcome': 'Spayed Female'}
```

Successfully inserted new document.

Test case (create): Test Case #4

Figure 8:

2.2. Contents of the test cases:

Specifically, contents of: Test Case #1


```
[{'_id': ObjectId('68e159c1effe604f01f4d3a8'),
  'tid': 'test_case_1',
  'rec_num': 20001,
  'age_upon_outcome': '1 year',
  'animal_id': 'TA736551',
  'animal_type': 'Dog',
  'breed': 'Labrador Retriever/Australian Cattle Dog',
  'color': 'Black',
  'date_of_birth': '2015-10-12',
  'datetime': '2016-11-27 18:00:00',
  'monthyear': '2016-11-27T18:00:00',
  'name': '*',
  'outcome_subtype': '',
  'outcome_type': 'Adoption',
  'sex_upon_outcome': 'Spayed Female',
  'location_lat': 30.4443212820182,
  'location_long': -97.7326980338793,
  'age_upon_outcome_in_weeks': 58.9642857142857}]
```


Specifically, contents of: Test Case #2


```
[{'_id': ObjectId('68e159c1effe604f01f4d3a9'),
  'tid': 'test_case_2',
  'rec_num': 20002,
  'age_upon_outcome': '5 months',
  'animal_id': 'TA693288',
  'animal_type': 'Cat',
  'breed': 'Domestic Shorthair Mix',
  'color': 'Orange',
  'date_of_birth': '2013-09-28',
  'datetime': '2013-12-09 18:36:00',
  'monthyear': '2013-12-09T18:36:00',
  'name': '*Brain',
  'outcome_subtype': ''
```


Figure 9:

```
[5]: # 5. Use the update function to update an existing record in the aac database

update_test_cases = {
    "Test Case #1":[
        {"tid":"test_case_1"}, {"age_upon_outcome": "6 months"}
    ],

    "Test Case #2":[
        {"tid":"test_case_4"}, {"sector": "still a lolcow gamer"}
    ],

    "Test Case #3":[
        {"tid":"test_case_5"}, {"sector": "still a lolcow gamer"}
    ]
}

render_markdown("## 3.1. Test cases for ``.update()`` function:")
for k, v in update_test_cases.items():
    render_markdown("### Test case (update): {0}".format(k))
    results = mongo_instance.update(
        v[0], v[1]
    )
    render_markdown("__Number of documents affected: {0}__\n\n".format(results))
    render_markdown("__Contents of the documents affected:__\n\n")
    after = mongo_instance.read(v[0])

    if len(after) > MAX_PRINTOUT:
        pprint.pp(after[0:MAX_PRINTOUT])
    else:
        pprint.pp(after)
```

3.1. Test cases for `.update()` function:

Figure 10:

3.1. Test cases for `.update()` function:

Test case (update): Test Case #1

Number of documents affected: 1

Contents of the documents affected:


```
[{'_id': ObjectId('68e159c1effe604f01f4d3a8'),
  'tid': 'test_case_1',
  'rec_num': 20001,
  'age_upon_outcome': '6 months',
  'animal_id': 'TA736551',
  'animal_type': 'Dog',
  'breed': 'Labrador Retriever/Australian Cattle Dog',
  'color': 'Black',
  'date_of_birth': '2015-10-12',
  'datetime': '2016-11-27 18:00:00',
  'monthyear': '2016-11-27T18:00:00',
  'name': '*',
  'outcome_subtype': '',
  'outcome_type': 'Adoption',
  'sex_upon_outcome': 'Spayed Female',
  'location_lat': 30.4443212820182,
  'location_long': -97.7326980338793,
  'age_upon_outcome_in_weeks': 58.9642857142857}]
```


Test case (update): Test Case #2

Number of documents affected: 1

Contents of the documents affected:


```
[{'_id': ObjectId('68e159c1effe604f01f4d3ab'),
  'tid': 'test_case_4',
  'certificate_number': 666,
  'business_name': 'DarkSydePhil',
  'date': 'April 20, 1984',
  'result': 'Shoutout to HeyArVy'
```


Figure 11:

```
[6]: # 6. Use the delete function to delete an existing record in the aac database

delete_test_cases = {
    "Test Case 1": {"tid": "test_case_1"},
    "Test Case 2": {"tid": "test_case_2"},
    "Test Case 3": {"tid": "test_case_3"},
    "Test Case 4": {"tid": "test_case_4"},
    "Test Case 5": {"tid": "test_case_5"}
}

render_markdown("## 4.1. Test cases for ``.update()`` function:")
for k, v in delete_test_cases.items():
    render_markdown("### Test case (delete): {0}".format(k))
    results = mongo_instance.delete(v)

    render_markdown("Number of documents affected: {0}".format(results))
```

4.1. Test cases for `.update()` function:

Test case (delete): Test Case 1

Number of documents affected: 1

Test case (delete): Test Case 2

Number of documents affected: 1

Test case (delete): Test Case 3

Number of documents affected: 1

Test case (delete): Test Case 4

Number of documents affected: 1

Figure 12: