CS-340: Assignment 4-1: Module 4 Journal
Prepared on: September 29, 2025
Prepared for: Prof. Jeff H. Sanford
Prepared by: Alexander Ahmann

Contents

1 Introduction

The following writeup documents the progress made in the development of the first project. In particular, I have improved a given Python module containing the "CRUD class"— which is a collection of functions by which to perform *create*, *read*, *update* and *delete* transactions, to interface with the MongoDB instance. [1]

[1] See Giamas (2022, chapter 3) to learn more about CRUD transactions and operations.

In its current incarnation, the "CRUD class" connects to a local MongoDB instance and logs in specifically with the username: `aacuser` and the password `WingsofRedemption`, and has the ability to interfacing with the `aac` database with `readWrite` privileges — though this might change in the future to login with whatever credentials given by a programmer using the CRUD class. Furthermore, a rudimentary test-harness has been created with the *Jupyter Notebook*[2] to assess the quality of the refined "CRUD class" module. With a few test cases, I was able to demonstrate that the "CRUD class" can perform basic tasks. Possible mistakes in the programming and implementation of certain parts of the codebase are discussed.

1.1 Brief description of the figures attached

Figures 1–7 are all screenshots depicting the basic test harness created in *Jupyter Notebook*. Specifically, figures 1 and 2 depict the "setup" of the test harness, where I proceed to import needed Python modules, and the results of inserting two arbitrary documents into the `aac.animals` collection, and figures 3 to 7 depict the "setup" and execution of querying the MongoDB instance with various filters.

1.2 Review of previous work (Task 1)

> "The *Austin Animal Center* (AAC) Outcomes data set is preloaded in Codio as `aac_shelter_outcomes.csv` in the `datasets` directory. In the Module Three milestone, you imported the data as `aac` in the MongoDB terminal. If you did not import the database, please do so now and ensure that you create the aacuser using the steps provided in note section above. Use the database name `aac` and collection name `animals`.[3] — CS-340 (n.d., §1)

The prompt above describes the work needed to configure the infrastructure needed for the *MongoDB* instance. Such work has already been completed in a previous assignments,[4] though I do think that briefly describing the setup of a new Mongo database, and the configuration of the MongoDB infrastructure to include a custom user with appropriate *role-based access*

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

[3] Note that this is work that has already been done. This subsection exists to document these tasks as to give additional context to the reader.

[4] See Ahmann (2025).

controls (RoBAC). This description is to give readers additional context that will help them understand the problem set and my solutions to them.

The important stuff is that the following tasks:

- I imported a *comma-seperated values* (CSV) dataset, and converted its tabular format to *MongoDB*'s *BSON* format. The CSV-to-BSON dataset is stored as a collection.

- I created an end-user account with the appropriate permissions needed to access the newly imported dataset, and applied the principle of least privilege to limit the scope of access to this new user.

I also created a single-field and a partial-field index for query optimization, but such stuff is irrelevant to the work to be done for this module, so I will leave it to the reader to consult the previous assignment if they are interested in the process of MongoDB index-making.

Regarding the importing of the CSV dataset, I used a variant of the `mongoimport` utility to convert the tabular CSV dataset into a MongoDB BSON document as depicted by plate 1.

Plate 1: `mongoimport` command.

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

The `--db` flag instructs the command to import the CSV dataset into the `aac` database, the `--collection` flag instructs the command to specifically store the CSV dataset in the `animals` collection, the `--type` flag instructs the command to treat the input as a CSV dataset, the `--file` flag instructs the command to read the `aac_shelter_outcomes.csv` for import, and the `--headerline` flag instructs the command to use the column values in the first row of the target CSV file as labels for the keys in each document of the collection.

When the `mongoimport` command was executed, it was successful in importing the target CSV file into the `animals` collection of the aac database.

After importing the database and optimizing `find*()` operations with indexes, I proceeded to set up a user account that has `readWrite` permissions for the `aac` database. The exact MongoDB query that I used to create the new MongoDB user account is shown in plate 2.[5]

Plate 2: Query to Create `aacuser`.

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

Specifically, the `user` field specifies the username, which I set to `aacuser`, and the `roles` field contains a list of `role` and `db` keys contain the values for the permissions and databases that are granted to `aacuser`. The `pwd` field is a bit more interesting, since I used the `passwordPrompt()` to get user input for the password. This is a secure means by which to get the password. For this project, I used the password `WingsofRedemption` for the new `aacuser` account.

The following are important things to keep in mind when proceeding to the current problem set, and later the first project:

- The database name that I will work with is set to `aac`.
- The collection where the document data are stored in is called `animals`.
- The user, which embodies the principle of least privilege by means of a role-based access control, is called `aacuser`.
- The `aacuser` account has a password of `WingsofRedemption`.
- The `aacuser` account has access to the `aac` database, with `readWrite` permissions.

[5]Note that when creating the `aacuser` account, the credential information is stored in the `admin` database. This will be relevant when specifying which database contains the `aacuser` login data with the `--authenticationDatabase` flag of the `mongosh` command. See Giamas (2022, chapter 9) to learn more about MongoDB user management.

- Two indexes, `breed_1` and `outcome_type_1`, have been created to optimize `find*()` queries.[6] This will come in useful not so much in this writeup, but most likely in the first project and future work to be done.

2 Problem Set & Their Solutions

The main task of this writeup is to develop a Python module that interfaces with a MongoDB instance, as expressed in tasks 2.1 and 2.2. Testing the library to ensure its reliability comes second. The following preface for tasks 2.1 and 2.2 describe what is to be done regarding the development of the Python library:

> "Next, you must develop a Python module in a PY file using object-oriented programming methodology to enable *create* and read functionality for the database. Other Python scripts must be able to import your Python code as a module to support code reusability. A `CRUD_Python_Module.py` file has been created in the `code_files` directory in Codio with the example starter code." — CS-340 (n.d., §2), regarding tasks 2a and 2b.

Tasks 2a and 2b involve the refinement of a given "CRUD class," in a given Python module, by which to interface with the MongoDB instance: logging in through a user, the `aacuser` in this case, and performing *create, read, update, and delete* operations on said instance. I will use industry standard best practices when developing the class, and document the functionality to the best of my ability.[7]

Before attacking the problems outlined in tasks 2a and 2b, I wanted to give an overview of the Python module `CRUD_Python_Module.py` — which is provided as a starter by which to develop the functionality for the MongoDB client-side interface. Plate 3 depicts a "basic template" outlining what the codebase will look like — specifically the "CRUD class" that I will use to interface with MongoDB instances.

To give a brief description, an `AnimalShelter` class is declared in line 4, which inherits functionality from `PyMongo`'s `MongoClient` "super-class." Lines 6-11 define variables that will be used as parameters to connect to the MongoDB instance. Lines 13-15 initiate the connection, and define objects `database` and `collection` that specify which databases and collections to work with, respectively.

[6] See Ahmann (2025, §1.2–1.3) — which elaborate more on indexes.

[7] As noted in CS-340 (n.d., §2), best practices include: proper naming conventions, exception handling, and inline comments.

Plate 3: Given Python Module's "CRUD Class"

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

class AnimalShelter(object):
    def __init__(self):
        USER = 'aacuser'
        PASS = 'WingsofRedemption'
        HOST = 'localhost'
        PORT = 27017
        DB = 'aac'
        COL = 'animals'

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

    def create(self):
        pass

    def read(self):
        pass
```

Two functions, `create()` and `read()`, exist to create new documents, and to return all documents from the given collection. Tasks 2a and 2b are to further develop them into a working solution that can pass basic testing.

2.1 Task 2a: Python method by which to insert a document into a MongoDB instance

> "Develop a CRUD class that, when instantiated, provides the following functionality (a): A method that inserts a document into a specified MongoDB database and collection:
>
> - Input argument to function will be a set of key/value pairs in the data type acceptable to the MongoDB driver insert API call

- Return `True` if successful insert, else `False`." — CS-340 (n.d., §2a)

For task 2a, an `insert` function is already provided, and my job was to improve upon it. Plate 4.1 depicts the result of a working function that I have developed.

Plate 4.1: Worked solution to `create` function

```
1. def create(self, data: dict) -> bool:
2.     try:
3.         if data is None:
4.             raise Exception("The data type
    should not be 'None'")
5.         elif not isinstance(data, dict):
6.             raise Exception("The data type
    should be a dictionary")
7.         else:
8.             self.database.animals.insert_one(data)
9.     except Exception as e:
10.         print("Exception raised: {0}".format(e))
11.         return False
12.    return True
```

The function takes in a dictionary, stored in `data`, and then returns a `bool` object that is either `True` or `False` depending on whether-or-not it was able to insert a new document into the `animals` collection.[8] Exception handling is used to keep the service working in the case of some runtime error, as indicated by the `try` and `except` blocks defined on lines 2 and 9, respectively. An exception is raised if the given `data` input is not a dictionary, or is a `None` type.

2.2 Task 2b: Python method by which to query documents into a MongoDB instance

> "[Following from the work completed in task 2a, implement further code that] provides the following functionality (b):"

[8] I used type hints to guide other developers using this module into giving in correct inputs, and letting them know what to expect as an output.

- Input arguments to function should be the key/value lookup pair to use with the MongoDB driver find API call
- Return result in a list if the command is successful, else an empty list." — CS-340 (n.d., §2.2)

Unlike task 2a, a `read()` function had not be provided, and the student had to use their understanding of Python and MongoDB to create one *ex nihilo*. Plate 4.2 depicts the worked solution that I came up with for a `read()` function.

Plate 4.2: Worked solution to `read` function

```
1. def read(self, query: dict) -> list:
2.     try:
3.         results = None
4.         if not isinstance(query, dict):
5.             raise Exception("’query’
    should be a dictionary")
6.         else:
7.             if query == {}:
8.                 results = self.collection.find()
9.             else:
10.                results = self.collection.find(query)
11.            return [doc for doc in results]
12.    except Exception as e:
13.        print("Exception raised: {0}".format(e))
14.    return []
```

Specifically, a `read` function is defined in line 1. Type hints are used to specify that the `query` takes in a Python dictionary, and that the function returns a list. The list may be empty in the case that the `find` transaction failed, or in the case that the collection does not have any documents. Exception handling is used, as shown with the `try` and `except` code blocks on lines 2 and 12, respectively.

In the `try` block, assuming that the given `query` is a Python dictionary, it will execute a generic find with no filters if the dictionary is empty (lines 7 and 8). Otherwise, it will use the given filters in `query` to narrow out the results (lines 9 and 10). The results are stored in a variable called `results`, and list comprehension is used to build up a Python list of documents which is then to be returned (line 11).

2.3 Task 3: Testing Script for the newly developed Python MongoDB interface

Task 3 involves the creation of test script in the form of a *Jupyter Notebook* that will ensure that the newly created "CRUD class" functions as expected, and works well under a semi-hostile environment.[9] I opened up "starter notebook" called `ModuleFourTestScript.ipynb` and started coding the test-harness by importing needed modules and libraries, including the newly created Python CRUD module, and then developed a few test cases by which to pass to the developed `create` and `read` functions. Plate 5.1 depicts the imported modules.

Plate 5.1: Jupyter Notebook:
Importing Needed Modules

```
import pprint
from IPython.display import Markdown, display

from CRUD_Python_Module import *
mongo_instance = AnimalShelter()
```

The `pprint` module is used to print out "easier to read" JSON and the `Markdown` and `display` functions are used to render Markdown into the Jupyter Notebook. These just make it easier to read the results. I then proceeded to test the `create()` function. Plate 5.2 depicts the test cases, and plate 5.3 depicts the procedure by which to test each case.

[9] I say "semi-hostile" because the test cases that I wrote are fairly small, and a more comprehensive list of test cases, and perhaps even a software fuzzer or fault-injection tool, is needed to really assess how well the library performs under an "in the wild" environment.

Plate 5.2: Arbitrary Documents as Test Cases

```
insert_test_cases = {
    "test_case_1": {
        "_id":"test_case_20032-2020-DSP",
        "certificate_number": 666,
        "business_name":"DarkSydePhil",
        "date":"April 20, 1984",
        "result":"Shoutout to HeyArVy",
        "sector":"lolcow gamers"
    },
    "test_case_2": {
        "_id":"test_case_20032-2020-WINGS",
        "certificate_number": 777,
        "business_name":"WingsOfRedemption",
        "date":"April 20, 1985",
        "result":"Shoutout to HeyArVy (again :p)",
        "sector":"lolcow gamers",
        "address": {
            "number": 7777,
            "street":"@WingsTings",
            "city":"Richards",
            "zip":90210
        }
    }
}
```

`insert_test_cases` is something of a "meta-dictionary" in which each entry is another dictionary that will be used as a parameter to be passed to `create()`'s data parameter. A for loop (plate 5.3) is used to iterate over the `insert_test_cases` dictionary, and pass each test case into the newly declared `mongo_instance.create()` function.

Plate 5.3: Procedure by which to test the `create()` function

```
1. for n, case in insert_test_cases.items():
2.    print("Insert Document Case:
   \"{0}\": {1}\n\n".format(n, case))
3.    successful = mongo_instance.create(case)
4.    if successful:
5.        print("\tSuccessfully
   inserted new document.")
6.    else:
7.        print("\tFailed to insert new
   document.\n")
```

Recall that `mongo_instance.create()` returns a boolean variable that results in `True` if the transaction completed successfully, and a `False` if it failed. To measure this behaviour, I stored what was returned into a variable called `successful` (plate 5.3, line 3), and then printed out a "success" message if `successful` is `True`, otherwise a "failed" message. Figures 1 and 2 depict the results of this part of the test harness.

I then proceeded to test the `read()` function with code depicted in plate 5.4. A `MAX_PRINTOUT` variable is declared (line 1), which limits the number of documents printed out to the Jupyter Notebook.[10] A `read_test_cases` is declared (lines 2-7), which is a "meta list" of lists that contain a "test case" label as its first entry, and a dictionary denoting filter parameters as a second entry. These are inputs to be used for testing the `mongo_instance.read()` function.

[10] This was added as previous attempts to print out all documents in the `animals` collection resulted in failure because of limits in short-term random access memory.

Plate 5.4: Procedure by which to test the `read()` function

```
1. MAX_PRINTOUT = 5

2. read_test_cases = [
3.    ["All documents", {}],
4.    ["Documents for Animal Type = Dog",
    {"animal_type":"Dog"}],
5.    ["Document = Test Case 1",
    {"_id":"test_case_20032-2020-DSP"}],
6.    ["Document = Test Case 2",
    {"_id":"test_case_20032-2020-WINGS"}]
7. ]

8. for test_case in read_test_cases:
9.     results = mongo_instance.read(test_case[1])
10.
11.    display(
12.        Markdown(
13.            "## Results for query test case:
    \"{0}\": ".format(test_case[0])
14.        )
15.    )

16.    if len(results) > MAX_PRINTOUT:
17.        pprint.pp(results[0:MAX_PRINTOUT])
18.    else:
19.        pprint.pp(results)
```

A for-loop is used to iterate through the test cases defined in `read_test_cases` (lines 8–19). `mongo_instance.read()` is executed, and its results are stored in the `results` variable (line 9). Results from the `read()` function can be hard to read, and I got around this problem by using the `display` and `Markdown` functions to render a header that makes it easier for human analysts to differenciate between test cases (lines 11–15). I also instructed the `pprint.pp()` "pretty formatting" function to print out a maximum number of documents as defined in the `MAX_PRINTOUT` variable (lines 16–19).

3 Discussion

Due to time constraints, the quality of the "CRUD class" codebase and other work for this module has somewhat been "cheapened." When reviewing the worked solutions and code for the test harness, I noticed some inconsistencies and inefficiencies. The following is a short list of "quirks" that I was able to identify while proof-reading the worked solutions:

- When constructing test cases, I made the `insert_test_cases` into a dictionary, where the "name" of the test case was the key of the dictionary, and the document to be inserted was the key's respective value (plate 5.2).

  But in the `read_test_cases`, I presented a multidimensional list where the test case's label is the first item in a particular "sublist," and the actual test case is the second item in the list (plate 5.4). This is clearly an inconsistency in the implementation of a bank of test cases.[11]

- When handling exceptions, I print out error messages through the built-in `print()` function. However, I should consider implementing Python's `logging` module when recording errors.[12]

- Regarding the `read()` function, it would probably be a better idea to define the `results` variable outside of the `try/catch` block, all the way at the very start of the function — just after the function's `def` declaration, and initialise it as an empty list — i.e. `results = []`. This would be good for eliminating redundancies.

I do intend to address identified quirks of the "CRUD class" Python module, and eliminate bad code and introduce better coding practices. Regarding the latter, I used proper naming conventions, exception handling, comments for documentation, and type hints as to hopefully make its readabiltiy and reliability decent. I also intend to apply techniques of refactoring, linting, and unit testing with some level of input fuzzing to ensure a higher quality codebase.

3.1 Summary

The following are takeaways regarding the progress of the client-side Python module, its "CRUD class," and its respective Jupyter Notebook test harness:

[11] I was originally going to write this sentence as "a *list* of test cases," but decided to not *list* because that is a specific data structure in Python.

[12] See Python 3.13.7 (n.d.). *logging — Logging facility for Python.* Retrieved on Sept. 29, 2025 from: https://docs.python.org/3/library/logging.html

- Functionality that allows for the “create” and “read” functions of a CRUD database has been implemented.
- A rudimentary test harness in the form of a *Jupyter Notebook* was created to perform a cursory assessment the reliability of the “CRUD class.”
- I have used proper function and variable naming, exception handling, comments as documentation, and type hints to improve code readability and reliability.
- I have identified some “quirks” in the current implementation of the CRUD class. I intend to address these in the future.

Of course, further work is called for. Specifically, more functionality to meet *Project 1*’s requirements, the refactoring and further testing of the “CRUD class” Python module, the creation of refined and more exhaustive test cases, and more rigour in the software testing process. I will elaborate more on this in the future.

References

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

CS-340 (n.d.). *Module Four Milestone 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 Appendix: List of figures depicting screen-shots that demonstrate task completion

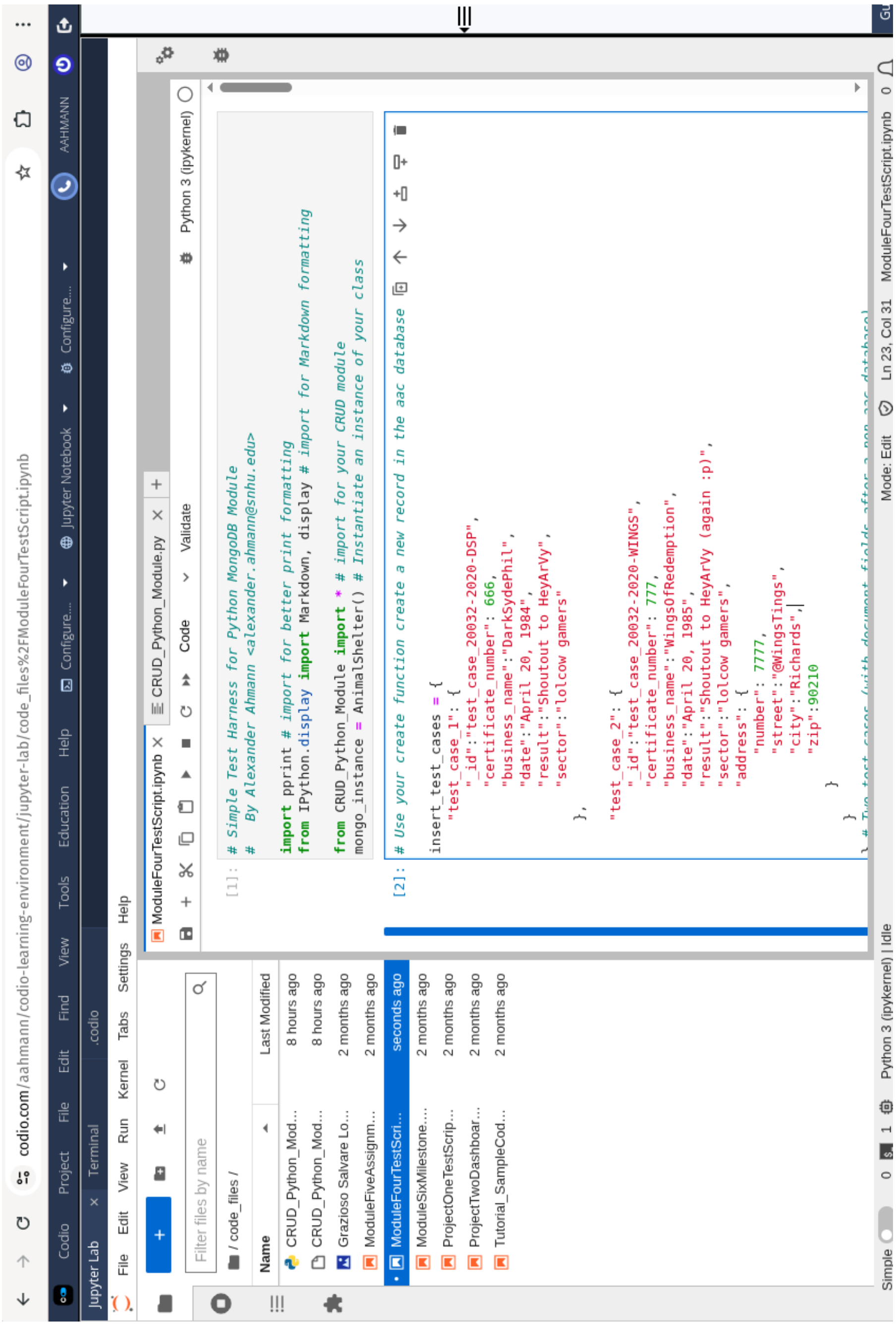

Figure 1: Results of running the simple test-harness Jupyter Notebook.

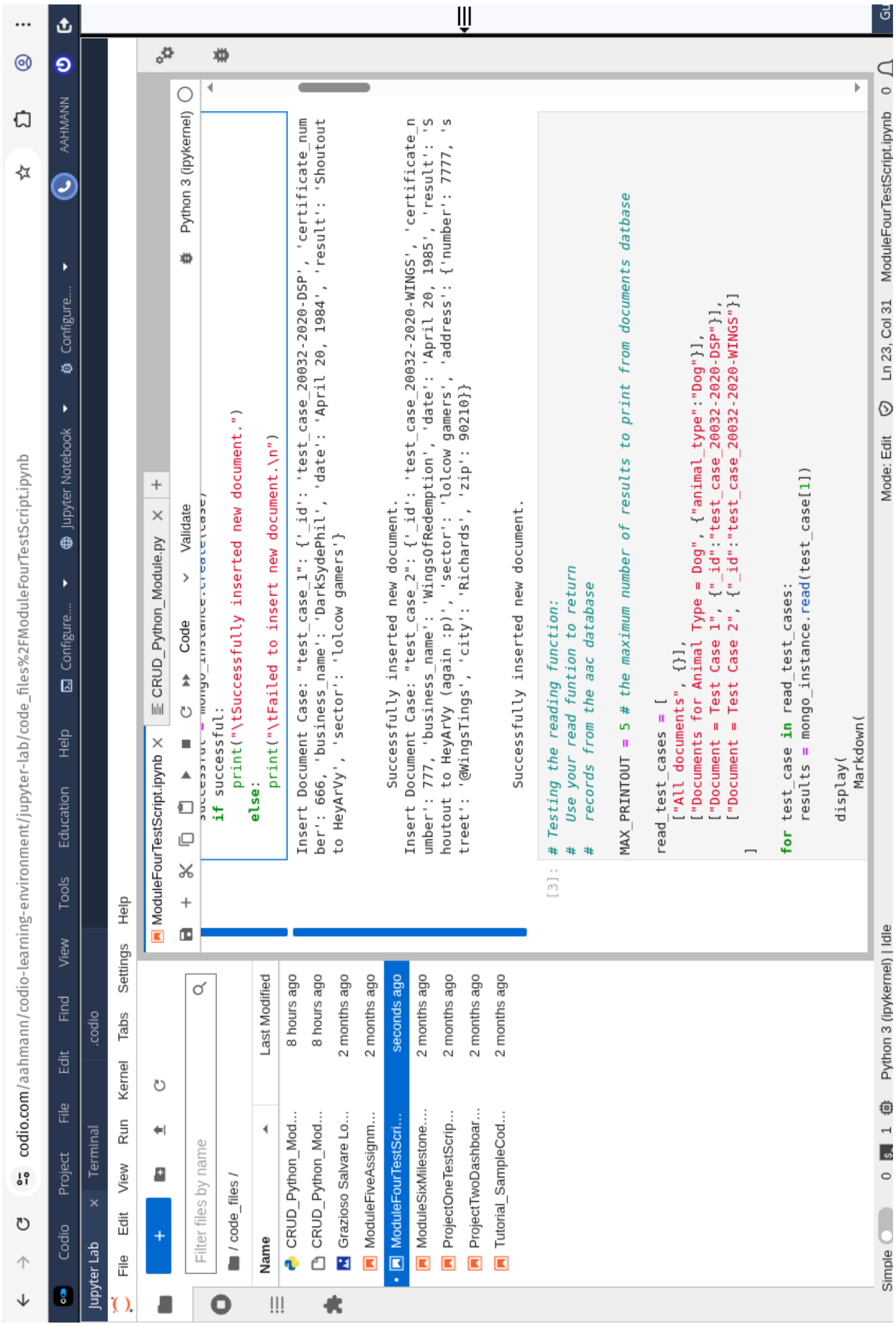

Figure 2: Results of running the simple test-harness Jupyter Notebook (cont.).

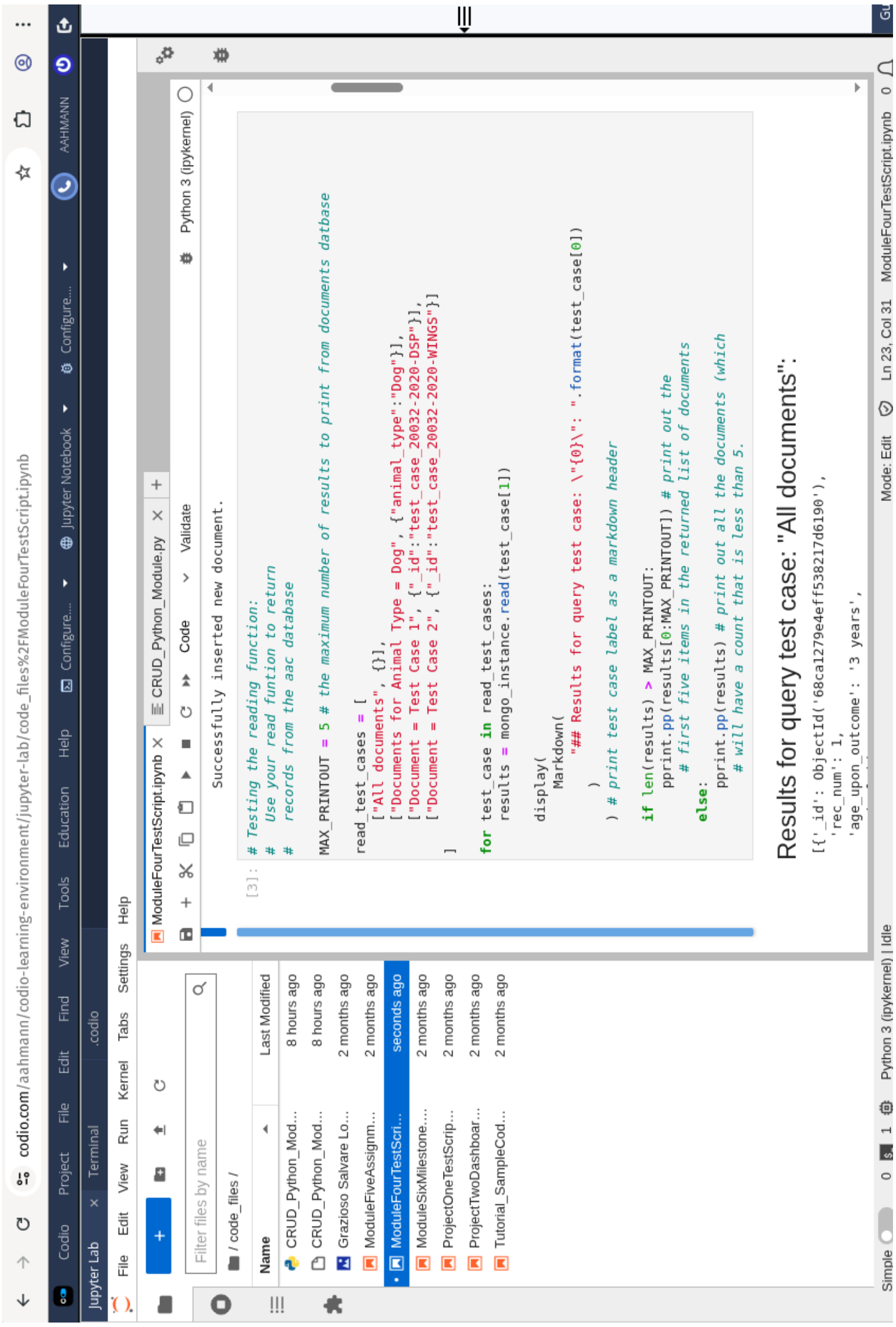

Figure 3: Results of running the simple test-harness Jupyter Notebook (cont.).

Figure 4: Results of running the simple test-harness Jupyter Notebook (cont.).

Figure 5: Results of running the simple test-harness Jupyter Notebook (cont.).

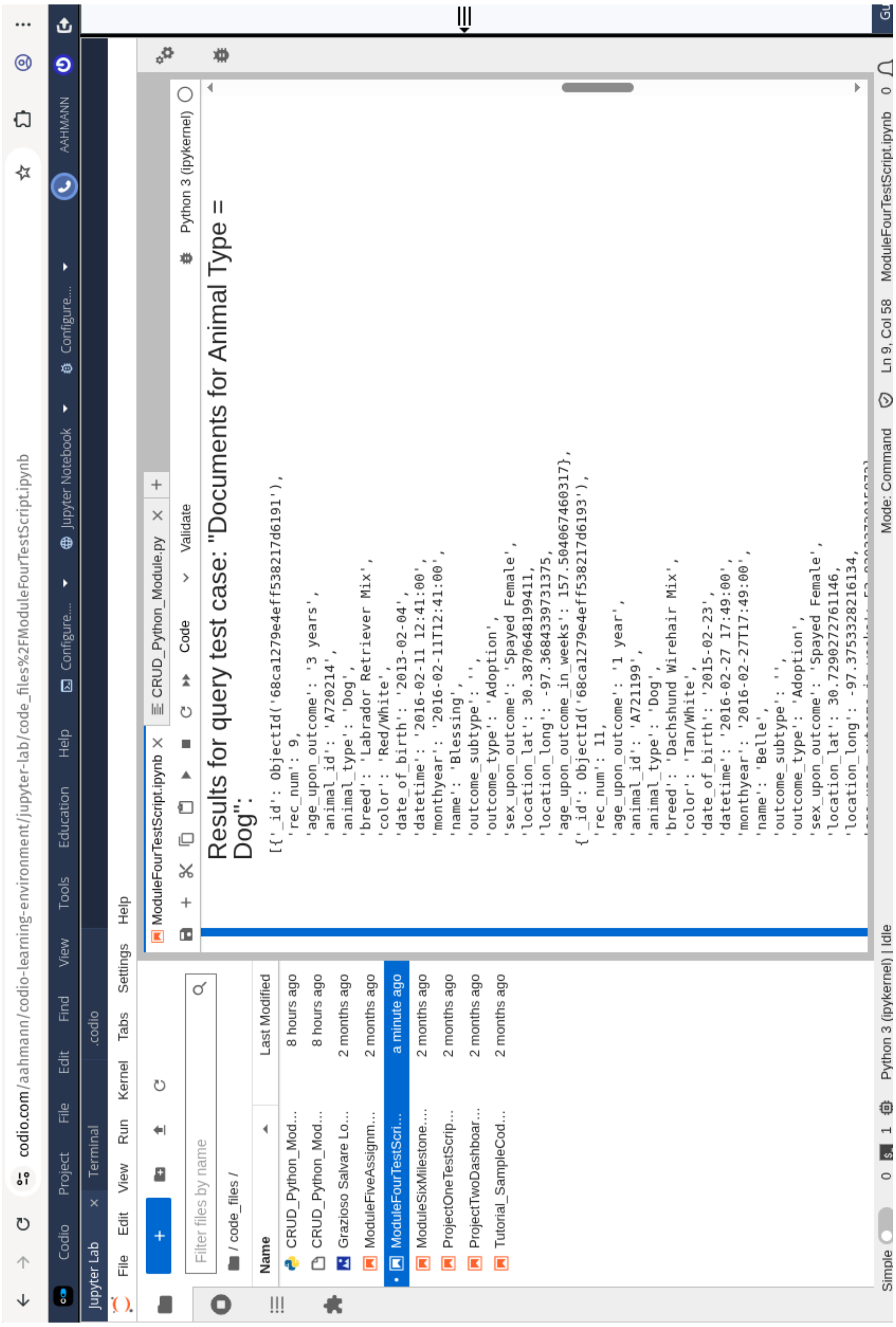

Figure 6: Results of running the simple test-harness Jupyter Notebook (cont.).

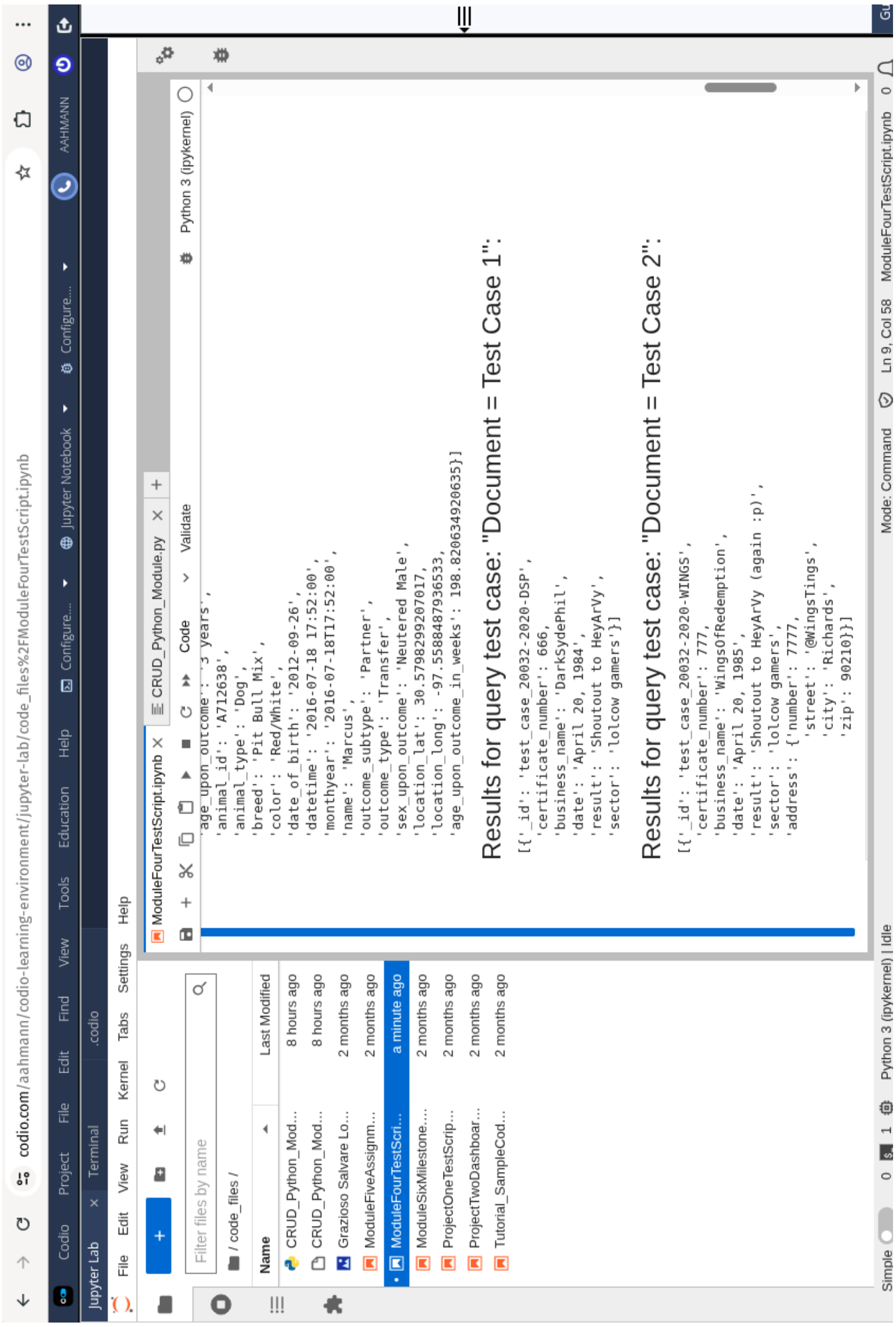

Figure 7: Results of running the simple test-harness Jupyter Notebook (cont.).