json-as-db

Release 0.2.0

Joonas

Jan 25, 2023

CONTENTS

Using JSON file as a very simple and lightweight database.

Save all your records into file as JSON format to be easy to read for human. also fetch from the file just as what you see.

CHAPTER

ONE

CONTENTS

1.1 Installation

Installing JSON-as-DB is pretty simple.

Note: Using a virtual environment will make the installation easier, and will help to avoid clutter in your system-wide libraries.

JSON-as-DB is available on Pypi as `json-as-db`, so you can install using pip as like following command.

```
$ pip install json-as-db
```

Also you can install directly from source. You will need git in order to clone the repository.

```
$ git clone https://github.com/joonas-yoon/json-as-db.git
$ cd json-as-db
$ pip install -e .
```

If you want to test methods working well, here is the unit tests in `tests/` directory. These unittest-based tests are written using PyTest.

Now that you have installed all dependencies, you can run tests to check functions like,

```
$ PYTHONPATH=src python -m pytest
```

1.2 Examples

1.2.1 Database

The *Database* is compatible with dictionary where is Python built-in class, except only for some of setters.

So you can use them as dictionary-like, including useful CRUD methods and more.

```
>>> from json_as_db import Database
>>> db = Database()
>>> db.add([
...   { "id": 1001, "type": "Regular" },
...   { "id": 1002, "type": "Chocolate" }
... ])
["aT7kM2pW8L7JisSkNjpAhr", "RUJGcVBFANvNRReXa8U3En"]
>>> db.get("aT7kM2pW8L7JisSkNjpAhr")
```

(continues on next page)

(continued from previous page)

```
{"id": 1001, "type": "Regular"}
>>> db.count()
2
>>> db.all()
[{"id": 1001, "type": "Regular"}, {"id": 1002, "type": "Chocolate"}]
>>> db.save('path/dir/file.json')
```

Moreover, *Database* provides and supports many operations. Please see the following examples.

Basic operations

Add

You can add single item into *Database*. It returns automatically generated ID of added item. This new ID is important to use to get and to manipulate data from *Database*.

```
>>> db.add({
...     "id": 1001,
...     "type": "Regular"
... })
"aT7kM2pW8L7JisSkNjpAhr"
```

It supports to add multiple items with list.

```
>>> db.add([
...   {
...     "id": "1001",
...     "type": "Regular"
...   },
...   {
...     "id": "1002",
...     "type": "Chocolate"
...   },
...   {
...     "id": "1003",
...     "type": "Blueberry"
...   },
... ])
['FqkmbYFSCRCAHQWydhM69v', 'RUJGcVBFANvNRReXa8U3En', 'F3c3rWpzb3Wh2XYQpoYu9v']
```

Remove

You can remove object(s) in *Database* using remove method by given ID(s). It returns removed object from *Database*.

```
>>> db.remove("aT7kM2pW8L7JisSkNjpAhr")
{'id': '1001', 'type': 'Regular'}
>>> db.remove(["FqkmbYFSCRCAHQWydhM69v", "RUJGcVBFANvNRReXa8U3En"])
[{'id': '1001', 'type': 'Regular'}, {'id': '1002', 'type': 'Chocolate'}]
```

Get

You can get object(s) by given ID(s) in two ways, the first one is:

```
>>> db.get("aT7kM2pW8L7JisSkNjpAhr")
{'id': '1001', 'type': 'Regular'}
>>> db.get(["FqkmbYFSCRCAHQWydhM69v", "RUJGcVBFANvNRReXa8U3En"])
[{'id': '1001', 'type': 'Regular'}, {'id': '1002', 'type': 'Chocolate'}]
```

The second way, to get a single object with ID, is using getter operation.

```
>>> db["aT7kM2pW8L7JisSkNjpAhr"]
{'id': '1001', 'type': 'Regular'}
```

If there is no key in *Database* with given ID(s), it returns simply `None`.

```
>>> db.get("NotExistKeyString")
None
>>> db.get(['FqkmbYFSCRCAHQWydhM69v', 'NotExistKeyString'])
[{'id': '1001', 'type': 'Regular'}, None]
```

Modify

Single item

```
>>> db.modify(
...     id="FqkmbYFSCRCAHQWydhM69v",
...     value={
...         "type": "Irregular"
...     })
{'type': 'Irregular'}
```

Multiple items

```
>>> db.modify(
...     id=["FqkmbYFSCRCAHQWydhM69v", "RUJGcVBFANvNRReXa8U3En"],
...     value=[
...         {'type': 'Apple'}, {'type': 'Orange'}
...     ])
[{'type': 'Apple'}, {'type': 'Orange'}]
```

Find

You can find objects with passing lambda function to handle each items. For example:

```
>>> db.find(lambda x: x['type'].endswith('e'))
['2g4kaFAiDBPchz66HNPsZa', 'dpKsCc7evmV7Mxq8ikgY89', 'fewugXnJHosmaXeqbXrLtD']
>>> db.get(['2g4kaFAiDBPchz66HNPsZa', 'dpKsCc7evmV7Mxq8ikgY89', 'fewugXnJHosmaXeqbXrLtD
↪'])
[{'id': 1001, 'type': 'Chocolate'}, {'id': 1002, 'type': 'Orange'}, {'id': 1003, 'type':
↪'Apple'}]
```

Also we provide flexible wrappers for ==, *!*=, <, <=, >, >= to comparison operators.

```
>>> from json_as_db import Key
>>> db = Database()
>>> db.add([{'product': 'Chocolate', 'amount': 10}, {'product': 'Orange', 'amount': 1}, {
↪'product': 'Apple', 'amount': 3}])
['oTTno3xwizirjM6skVrZsi', 'Gw9pwyeV6cXJbHkT3sSUaW', 'LVw4smsL9WRgtRo5bTSnuX']
>>> db.find(Key('type') == 'Chocolate')
['oTTno3xwizirjM6skVrZsi']
>>> db.find(Key('product') > 'Apple')
['oTTno3xwizirjM6skVrZsi', 'Gw9pwyeV6cXJbHkT3sSUaW']
>>> db.find(Key('amount') <= 3)
['Gw9pwyeV6cXJbHkT3sSUaW', 'LVw4smsL9WRgtRo5bTSnuX']
```

Commit & Rollback

When `commit()`, it saves its states and all items at that time. Using `rollback()` restores all states and items from latest commit. Note that *Database* supports to store only for a single commit.

```
>>> db.all()  # Show all items before commit
[{'type': 'Orange'}]
>>> db.commit()
>>> db.add([{'type': 'Apple'}, {'type': 'Banana'}])  # Add some items after commit
>>> db.all()
[{'type': 'Orange'}, {'type': 'Apple'}, {'type': 'Banana'}]
>>> db.rollback()
>>> db.all()
[{'type': 'Orange'}]
```

Load

You can get the *Database* object from local JSON formatted file.

This method reads JSON file from directory where given path. In following example, it reads the file from `path/dir/sample.json`.

```
>>> db.load('path/dir/sample.json')
{'data': {'2g4kaFAiDBPchz66HNPsZa': {'type': 'Orange'}}, 'creator': 'json_as_db',
↪'created_at': '2022-12-25T14:23:28.906103', 'version': '1.0.0', 'updated_at': '2022-12-
↪25T14:23:28.906103'}
```

Save

Save *Database* into file as JSON format. You can read from this saved file.

```
>>> db.save()
```

It supports keyword parameters for JSON formatter and options to file saving. Please refer to the document page of modules in details.

```
>>> db.save(file_kwds={'encoding': 'utf-8'}, json_kwds={'indent': 4})
```

then you can see the file content as like the following,

```
{
    "created_at": "2022-12-25T16:50:02.459068",
    "creator": "json_as_db",
    "data": {
        "AwMJDzrjkpWJCee5iSozXW": {
            "type": "Orange"
        }
    },
    "updated_at": "2022-12-25T17:11:56.790276",
    "version": "1.0.0"
}
```

Other operations

All

We can get all values of items in *Database* without their keys.

```
>>> db.all()
[{'type': 'Yogurt'}, {'type': 'Apple'}, {'type': 'Banana'}]
```

Clear

The clear method removes all of the objects.

```
>>> db.clear()
```

Has

If we want to know whether *Database* has key, here is easy way to know. Please see the following examples.

```
>>> db.keys()
dict_keys(['AwMJDzrjkpWJCee5iSozXW', '5C8SJM54ogkCmsNJA2Cdja', '8LEJS5uGuopxcPQ3uKN8ty'])
>>> db.has('AwMJDzrjkpWJCee5iSozXW')
True
>>> db.has('NotExistsKeyString')
False
```

And it supports the parameter of list type as it will return list of each result as boolean.

```
>>> db.has(['AwMJDzrjkpWJCee5iSozXW', 'NotExistsKeyString'])
[True, False]
```

Count

```
>>> db.count()
3
```

```
>>> len(db)
3
```

Drop

This works as same as `clear()` method, but this returns the count of dropped items. As you know, the count of dropped one is exactly equal to the count using `count()` before dropping.

```
>>> db.all()
[{'type': 'Yogurt'}, {'type': 'Apple'}, {'type': 'Banana'}]
>>> db.drop()
3
>>> db.all()
[]
>>> db.drop()
0
```

Specials

Fields

Metadata

Retrieves metadata from *Database*. This doesn't contain hidden fields in instance of its.

```
>>> db.metadata
{'version': '1.0.0', 'creator': 'json_as_db', 'created_at': '2022-12-25T16:50:02.459068',
↪ 'updated_at': '2022-12-25T17:11:56.790276'}
```

Data

The `data` field is shortcut to get all items directly in code. It returns all key and values as the following,

```
>>> db.data
{'AwMJDzrjkpWJCee5iSozXW': {'type': 'Orange'}, '5C8SJM54ogkCmsNJA2Cdja': {'type': 'Apple
↪'}, '8LEJS5uGuopxcPQ3uKN8ty': {'type': 'Banana'}}
```

To keep data in safe, actions to set directly is prohibited. So, when you try to set it using syntax like `db.data = {}`, it will fail with *AttributeError* exception.

```
>>> db.data = {'NewInjectedIdWhatIWant': {'type': 'Bug'}}
AttributeError: can't set attribute
```

Version

Note: This does NOT mean the version of package. This is *Database* version for specification to read and parse.

```
>>> db.version
'1.0.0'
```

Accessor

Dictionary-like

Database object is compatible with *dictionary* which is Python built-in class.

All accessors are wrapped and it provides internal data, not metadata. Please check the following example runnings and results.

```
>>> db.data
{'AwMJDzrjkpWJCee5iSozXW': {'type': 'Orange'}, '5C8SJM54ogkCmsNJA2Cdja': {'type': 'Apple
↪'}}
>>> ID = 'AwMJDzrjkpWJCee5iSozXW'
>>> db.data[ID]
{'type': 'Orange'}
>>> db[ID]
{'type': 'Orange'}
>>> db.get(ID)
{'type': 'Orange'}
>>> db.get(ID).update({'type': 'Yogurt'})
>>> db.data
{'AwMJDzrjkpWJCee5iSozXW': {'type': 'Yogurt'}, '5C8SJM54ogkCmsNJA2Cdja': {'type': 'Apple
↪'}}
```

1.3 Modules

1.3.1 Database

class json_as_db.core.database.**Database**(**arg*, ***kwargs*)

Bases: dict

add(*item: Union[Any, List[Any]]*) → Union[str, List[str]]

Parameters
item (*Union[Any, List[Any]]*) – Object(s) to add to database

Returns
Automatically generated ID of added item

Return type
Union[str, List[str]]

all() → List[Any]

Provide all items in database.

Returns
All items as list

Return type
List[Any]

clear() → None

Clear all items. This method updates timestamp in metadata.

commit() → None

Save its states and all items at that time.

count() → int

Returns
indicates the count of all data

Return type
int

property data: dict

drop() → int

Returns
indicates the count of dropped items

Return type
int

find(*func: Callable[[...], bool]*) → List[str]

Returns array of IDs that satisfies the provided testing function.

Parameters
func (*Callable[..., bool]*) – A function to execute for each items in database. It will call *func(value)* to determine boolean.

Returns
array with id of found items

Return type
List[str]

get(*key: Union[str, List[str]]*, *default=None*) → Union[Any, List[Any]]

Get objects by given IDs when list is given. When single string is given, returns single object by given key

Parameters

- **key** (*str | List[str]*) – single key or list-like
- **default** (*Any, optional*) – default value if not exists. Defaults to None.

Returns
single object or list-like

Return type
Any | List[Any]

Examples

```
>>> db.get('kcbPuqpfV3YSHT8YbECjvh')
{...}
>>> db.get(['kcbPuqpfV3YSHT8YbECjvh'])
[{...}]
>>> db.get(['kcbPuqpfV3YSHT8YbECjvh', 'jmJKBJBAmGESC3rGbSb62T'])
[{...}, {...}]
```

has(*key: Union[str, List[str]]*) → Union[bool, List[bool]]

performs to determine whether has key

Parameters
key (*Union[str, List[str]]*) – to find with string(s) as key

Returns
boolean or array of boolean.

Return type
Union[bool, List[bool]]

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

load(*path: str*, *file_args: dict = {'encoding': 'utf-8', 'mode': 'r'}*, *json_args: dict = {}*) → *Database*

Load database object from a file

Parameters

- **path** (*str*) – a string containing a file name to load.
- **file_args** (*dict, optional*) – keyword arguments for file *open(**kwagrs)*. Defaults to dict(mode="r", encoding="utf-8",).
- **json_args** (*dict, optional*) – keyword arguments for *json.loads(**kwargs)*. Defaults to dict().

Raises
AttributeError – when JSON file does not contain keys and values to read into valid Database object.

Returns
itself

Return type
Database

property metadata: dict

modify(*id: Union[str, List[str]]*, *value: Union[Any, List[Any]]*) → Union[Any, List[Any]]

Parameters

- **id** (*str | List[str]*) – id(s) to modify
- **value** (*Any | List[Any]*) – value(s) to modify

Raises
ValueError – type or length is not matched

Returns
Modified value(s)

Return type
Any | List[Any]

remove(*key: Union[str, List[str]]*) → Union[Any, List[Any]]

Parameters
key (*Union[str, List[str]]*) – ID(s) to remove from database

Returns
removed items

Return type
Union[Any, List[Any]]

rollback() → None

Restore all states and items from latest commit.

save(*path: str*, *file_args: dict = {'encoding': 'utf-8', 'mode': 'w'}*, *json_args: dict = {'sort_keys': True}*, *make_dirs: bool = False*) → None

Save database object into a file as JSON format

Parameters

- **path** (*str*) – a string containing a file name to save.
- **file_args** (*dict, optional*) – keyword arguments for file *open(**kwagrs)*. Defaults to dict(mode="w+", encoding="utf-8",).
- **json_args** (*dict, optional*) – keyword arguments for *json.dumps(**kwargs)*. Defaults to dict(sort_keys=True,).
- **make_dirs** (*bool, optional*) – create non-exists directories in given path. Defaults to False.

update(*mapping: Union[dict, tuple] = ()*, ***kwargs*) → None

Note that this method overrides database itself.

values() → an object providing a view on D's values

property version: str

1.3.2 Matcher

class json_as_db.core.matcher.**Comparator**

Bases: object

evaluate(*other: dict*) → bool

class json_as_db.core.matcher.**Condition**(*key: str*, *value: Any*, *operator: Operator*)

Bases: *Comparator*

copy() → *Condition*

evaluate(*item: dict*) → bool

key: str

operator: Operator

value: Any

class json_as_db.core.matcher.**Conditions**(*lvalue: Union[Condition, Conditions]*, *rvalue: Union[Condition, Conditions]*, *operator: Operator*)

Bases: *Comparator*

copy() → *Conditions*

evaluate(*other: dict*) → bool

invert() → *Conditions*

According boolean algebra that satisfies De Morgan's laws In case of *(a & b).invert()*, it returns having the meaning of *(not a) or (not b)* that is equivalent to *(a and b)* in logical.

class json_as_db.core.matcher.**Key**

Bases: str

CHAPTER

TWO

LICENSE

Under the MIT license.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

j

INDEX

S

U

V