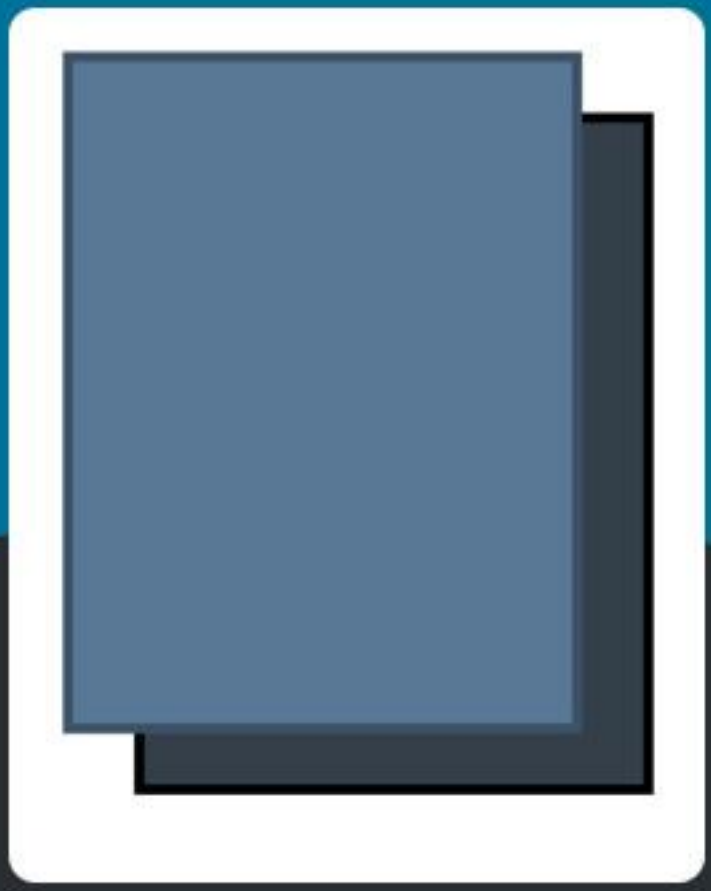

TinyDB

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

TinyDB is a lightweight database to operate various formats of the document. It is an easy and hustles free database to handle data of several applications. TinyDB is based on **python code** and supports clean API.

This database does not need any coding language. It handles small projects without any configurations. Generally, a database can store, retrieve, and modify data in a **JSON file**.

Audience

TinyDB tutorial is helpful to learn from students to professionals in easy steps. This tutorial is designed for beginners to advance level developers for a web application. This tutorial makes you an intermediate or advanced level expert with practice.

Prerequisites

To learn this tutorial, you need to know about the Python version of your computer. You must know the working procedure of the command prompt. You do not need to learn coding language or install the software.

Disclaimer & Copyright

Table of Contents

1. TinyDB – Introduction

What is TinyDB?

TinyDB, written in pure Python programming language, is a small and lightweight document-orineted database with no external dependencies. It provides simple APIs that makes it easy to use. We can use TinyDB database for small project applications without any configuration.

TinyDB module, available as a third-party module for Python programs, can be used to store, retereive, and modify the data in JSON format.

Features of TinyDB

TinyDB is a clean and hustle-free database to operate several formats of documents. It has the following features.

- **Really tiny:** TinyDB database is truly tiny in nature with only 1800 lines of code and 1600 lines tests.
- **Easy to use:** TinyDB is easy to use because of its simple and clean APIs.
- **Document oriented:** In TinyDB, we can store any document. The document will be represented as `dict`.
- **Independent:** The TinyDB database is independent of any external server and external dependencies from PyPI.
- **Compatible with Python 3.6 or latest:** TinyDB is tested and compatible with Python 3.6 and latest. It also works fine with PyPY3.
- **Extensible:** TinDB is easily extensible either by writing new storages or by modifying the behaviour of storages.

Advantages of TinyDB

TinyDB provide various benefits for students, users, and developers.

- TinyDB is open-sourced database aand it does not require any external configirations.
- It is quite easy-to-use, and the user can effortlessly handle documents.
- It automatically stores documents in the database.
- TinyDB is ideal in case of personal projects where we need to install some data.
- It is suitable for small applications that would be blown away by large databases like SQL or an external DB server.

- It uses a simple command line and query to operate data.
- There is 100% test coverage i.e., no explanation needed.

Limitatations of TinyDB

TinyDB will not be the right choice for your project if you need to:

- create indexes for tables,
- manage relationships between tables,
- use an HTTP server, or
- access from multiple processors.

Comparison with Other Databases

The following table highlights how TinyDB is different from MySQL and Oracle databases:

Comparison Basis	MySQL	Oracle	TinyDB
Configurations	Several Configurations	Several Configurations	Less Configurations, lightweight database
Complicated	Yes	Yes	No, easy-to-use and hustle-free
Affordable	No	No	Affordable than other databases
Manageable	Big database, hence difficult to manage	Big database, hence difficult to manage	Small and manageable

2. TinyDB – Environmental Setup

Prerequisite for TinyDB setup

To install TinyDB, you must have Python 3.6 or newer installed in your system. You can go to the link https://www.python.org/downloads/ and select the latest version for your OS, i.e., Windows and Linux/Unix. We have a comprehensive tutorial on Python, which you can refer at https://www.tutorialspoint.com/python3/index.htm

Installing TinyDB

You can install TinyDB in three different ways: using the Pack Manager, from its Source, or from GitHub.

Using the Package Manager

The latest release versions of TinyDB are available over both the package managers, **pip** and **conda**. Let us check how you can use them to install TinyDB:

To install `TinyDB` using **pip**, you can use the following command:

```
pip install tinydb
```

To install TinyDB via **conda-forge**, you can use the following command:

```
conda install -c conda-forge tinydb
```

From Source

You can also install TinyDB from source distribution. Go to link https://pypi.org/project/tinydb/#files to download the files and building it from source.

From GitHub

To install TinyDB using GitHub, grab the latest development version, unpack the files, and use the following command to install it:

```
pip install .
```

Setting up TinyDB

Once installed, use the following steps to set up the TinyDB database.

Step 1: Import TinyDB and its Query

First, we need to import TinyDB and its Query. Use the following command:

```
from tinydb import TinyDB, Query
```

Step 2: Create a file

TinyDB database can store data in many formats like XML, JSON, and others. We will be creating a JSON file by using the following file:

```
db = TinyDB('Leekha.json')
```

The above command will create an instance of *TinyDB* class and pass the file **Leekha.Json** to it. This is the file where our data will be stored. Now, the TinyDB database set up is ready, and you can work on it. We can now insert data and operate the value in the database.

Unistalling TinyDB

If in case you need to uninstall TinyDB, you can use the following command:

```
pip uninstall tinydb
```

3. TinyDB – Insert Data

We have created the instance of the TinyDB and passed a JSON file to it where our data will be stored. It is now time to insert the items in our database. The data should be in the form of a Python dictionary.

Syntax

To insert an item, you can use **`insert()`** method whose syntax is given below:

```
db.insert({'type1': 'value1', 'type2': 'value2', 'typeN': 'valueN'})
```

We can also create a dictionary first and then use the `insert()` method to insert the data into our database.

```
data_item = {'type1': 'value1', 'type2': 'value2', 'typeN': 'valueN' }
db.insert(data_item)
```

After running the above command, the insert() method will return the ID of the newly created object. And, our JSON file will look like the one shown below:

```
{"_default": {"1": {"type1": "value1", "type2": "value2", "typeN": "valueN"}}}
```

Look at the above table entries: **`'default'`** is the name of the table, '**1**' is the ID of the newly created object, and the **values** are the data we have just inserted.

Example: Inserting a Single Item

Let's understand the above concept with the help of examples. Suppose we have a database having student information showing roll numbers, names, marks, subjects, and addresses. Following is the information stored in the database:

```
[
  {
     "roll_number":1,
     "st_name":"elen",
     "mark":250,
     "subject":"TinyDB",
     "address":"delhi"
  },
  {
     "roll_number":2,
     "st_name":"Ram",
     "mark":[
        250,
```

```
            280
         ],
         "subject":[
            "TinyDB",
            "MySQL"
         ],
         "address":"delhi"
      },
      {
         "roll_number":3,
         "st_name":"kevin",
         "mark":[
            180,
            200
         ],
         "subject":[
            "oracle",
            "sql"
         ],
         "address":"keral"
      },
      {
         "roll_number":4,
         "st_name":"lakan",
         "mark":200,
         "subject":"MySQL",
         "address":"mumbai"
      },
      {
         "roll_number":5,
         "st_name":"karan",
         "mark":275,
         "subject":"oracle",
         "address":"benglore"
      }
]
```

In the above database, if you want to insert a new student record (i.e., a single item), use the following command:

```
db.insert({
    'roll_number': 6,
    'st_name':'jim',
    'mark':300,
    'subject':'sql',
    'address':'pune'
})
```

It will return the ID of the newly created object:

```
6
```

Let's enter **one more record**:

```
db.insert({
    'roll_number': 7,
    'st_name':'karan',
    'mark':290,
    'subject':'NoSQL',
    'address':'chennai'
})
```

It will return the ID of the newly created object:

```
7
```

If you want to check the stored items in the database, use the **all()** method as follows:

```
db.all()
```

It will produce the following **output**:

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'},
```

```
    {'roll_number': 6, 'st_name': 'jim', 'mark': 300, 'subject': 'sql',
'address': 'pune'},
    {'roll_number': 7, 'st_name': 'karan', 'mark': 290, 'subject': 'NoSQL',
'address': 'chennai'}
]
```

You can observe that it added two new data items in the JSON file.

Example: Inserting Multiple items at a Time

You can also insert multiple items at a time in a TinyDB database. For this, you need to use the **`insert_multiple()`** method. Let' see an example:

```
items = [
    {'roll_number': 8, 'st_name': 'petter', 'address': 'mumbai'},
    {'roll_number': 9, 'st_name': 'sadhana', 'subject': 'SQL'}
]
db.insert_multiple(items)
```

Now, check the stored items in database, using the `all()` method as follows:

```
db.all()
```

It will produce the following **output**:

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'},
    {'roll_number': 6, 'st_name': 'jim', 'mark': 300, 'subject': 'sql',
'address': 'pune'},
    {'roll_number': 7, 'st_name': 'karan', 'mark': 290, 'subject': 'NoSQL',
'address': 'chennai'},
    {'roll_number': 8, 'st_name': 'petter', 'address': 'mumbai'},
    {'roll_number': 9, 'st_name': 'sadhana', 'subject': 'SQL'}
]
```

You can observe that it added two new data items in the JSON file. You can also skip some key values from the dataitems (as we have done) while adding the last two items. We have skipped `'mark'` and `'address'`.

4. TinyDB – Retrieve Data

There are numerous ways with the help of which you can retrieve data from a TinyDB database. But to use those ways, you first need to create an instance of the **Query** class as follows:

```
from tinydb import Query
Student = Query()
```

Here, **Student** is the name of the database.

Let's check the various ways to retrieve the information from a TinyDB database.

Data Retrieval Using the Search() Method

The search() method, as its name implies, returns the list of items that mathches the query we provided, otherwise it will return an empty list.

For this and other examples, we will be using the following **student** database data:

```
[
    {
        "roll_number": 1,
        "st_name": "elen",
        "mark": 250,
        "subject": "TinyDB",
        "address": "delhi"
    },
    {
        "roll_number": 2,
        "st_name": "Ram",
        "mark": [
            250,
            280
        ],
        "subject": [
            "TinyDB",
            "MySQL"
        ],
        "address": "delhi"
    },
```

```
    {
        "roll_number": 3,
        "st_name": "kevin",
        "mark": [
            180,
            200
        ],
        "subject": [
            "oracle",
            "sql"
        ],
        "address": "keral"
    },
    {
        "roll_number": 4,
        "st_name": "lakan",
        "mark": 200,
        "subject": "MySQL",
        "address": "mumbai"
    },
    {
        "roll_number": 5,
        "st_name": "karan",
        "mark": 275,
        "subject": "TinyDB",
        "address": "benglore"
    }
]
```

Let's take an **example** to understand the search() method:

```
from tinydb import TinyDB, Query
db = TinyDB("leekha.json")
student = Query()
db.search(student.subject == 'TinyDB' )
```

The above query will retrieve the the following **output** from the student database:

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'TinyDB',
'address': 'benglore'}
]
```

Data Retrieval Using the get() Method

As opposed to the search() method, the get() method returns only one matching document. It will return None, otherwise. For example, let's take the following code:

```
from tinydb import TinyDB, Query
student = Query()
db.get(student.subject == 'TinyDB' )
```

The above query will retrieve the following data from the **student** database.

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'}]
```

Data Retrieval using the all() Method

The **all()** method returns all the documents in the database. For example,

```
db.all()
```

It will retrieve the entire data from the student database.

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'TinyDB',
'address': 'benglore'}
]
```

Data Retrieval Using the for Loop

The **for loop** also returns all the documents in the database. For example,

```
for info in db:
    print(info)
```

Just like the all() method, it will retrieve all the rows from the student database.

5. TinyDB – Update Data

TinyDB can store data in many formats and we can easily reterive the stored data using various methods. But sometimes, we need to update the data, for which we can use the **update()** method.

For updating the database, we first need to create an instance of the **Query** class. You can use the following command for this purpose:

```
from tinydb import Query
Student = Query()
```

Here, **Student** is the name of our database.

The update() Method

Here is the syntax for the update() method:

```
db.update({ updated field: updated information… }, stable field:  information)
```

Let's take an example to understand how the update() method works. For this example, we will be using the following **student** database:

```
[
   {
      "roll_number":1,
      "st_name":"elen",
      "mark":250,
      "subject":"TinyDB",
      "address":"delhi"
   },
   {
      "roll_number":2,
      "st_name":"Ram",
      "mark":[
         250,
         280
      ],
      "subject":[
         "TinyDB",
```

```
         "MySQL"
      ],
      "address":"delhi"
   },
   {
      "roll_number":3,
      "st_name":"kevin",
      "mark":[
         180,
         200
      ],
      "subject":[
         "oracle",
         "sql"
      ],
      "address":"keral"
   },
   {
      "roll_number":4,
      "st_name":"lakan",
      "mark":200,
      "subject":"MySQL",
      "address":"mumbai"
   },
   {
      "roll_number":5,
      "st_name":"karan",
      "mark":275,
      "subject":"TinyDB",
      "address":"benglore"
   }
]
```

As per the given data, the name of the student with the roll_number "1" is "elen". The following query will update the student name to "Adam":

```
from tinydb import TinyDB, Query
student = Query()
db.update({'st_name' : 'Adam'}, student.roll_number == 1 )
```

It will return the id of the updated object:

```
[1]
```

Now, you can use the all() method to see the updated database:

```
db.all()
```

It will display the updated data:

```
[{'roll_number': 1, 'st_name': 'Adam', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
{'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
{'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
{'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
{'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'TinyDB',
'address': 'benglore'}]
```

Sometimes, we need to update one or more fields of all the documents in a database. For this, we can use the **update()** mehod directly and don't need to write the query argument. The following query will change the address of all the students to 'College_Hostel':

```
db.update({'address': 'College_Hostel'})
```

It will return the ids of the updated object:

```
[1,2,3,4,5]
```

Again, you can use the all() method to see the updated database.

```
db.all()
```

It will show the updated data:

```
[{'roll_number': 1, 'st_name': 'Adam', 'mark': 250, 'subject': 'TinyDB',
'address': 'College_Hostel'},
```

```
{'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': ' College_Hostel '},

{'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': ' College_Hostel '},

{'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': ' College_Hostel '},

{'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'TinyDB',
'address': ' College_Hostel '}]
```

Observe that the address fields of all the rows have the same data, i.e., 'College_Hostel'.

6. TinyDB – Delete Data

In case you need to delete a particular set of data permanently from a TinyDB database, you can do so by using the **remove()** method. Like for reteriving and updating the data, you first need to create an instance of the **Query** class for deleting the data. You can use the following command for this purpose:

```
from tinydb import Query
Student = Query()
```

Here, **Student** is the name of our database.

The remove() Method

Here is the syntax for the remove() method:

```
db.remove( Query() field regex )
```

The remove() method accepts both an optional condition as well as an optional list of documents IDs.

The student Database

We will use the following **student** database, for the examples in this chapter.

```
[
   {
      "roll_number":1,
      "st_name":"elen",
      "mark":250,
      "subject":"TinyDB",
      "address":"delhi"
   },
   {
      "roll_number":2,
      "st_name":"Ram",
      "mark":[
         250,
         280
      ],
      "subject":[
```

```
         "TinyDB",
         "MySQL"
      ],
      "address":"delhi"
   },
   {
      "roll_number":3,
      "st_name":"kevin",
      "mark":[
         180,
         200
      ],
      "subject":[
         "oracle",
         "sql"
      ],
      "address":"keral"
   },
   {
      "roll_number":4,
      "st_name":"lakan",
      "mark":200,
      "subject":"MySQL",
      "address":"mumbai"
   },
   {
      "roll_number":5,
      "st_name":"karan",
      "mark":275,
      "subject":"TinyDB",
      "address":"benglore"
   }
]
```

Example: Deleting a Single Row of Data

Let's take an example to understand the remove() method.

```
from tinydb import TinyDB, Query
student = Query()
db.remove(student.roll_number == 5)
```

The above query will delete the data where the student's roll number is 5. It will return the ID of the removed object:

```
[5]
```

Now, we can use the all() method to see the updated database.

```
db.all()
```

It will display the data from the updated database:

```
[
    {'roll_number': 1, 'st_name': 'Adam', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'}
]
```

Example: Deleting Multiple Rows of Data

If you want to remove more than one row at a time, you can use the remove() method as follows:

```
from tinydb import TinyDB, Query
student = Query()
db.remove(student.roll_number > 2)
```

It will return the IDs of the removed object:

```
[3,4]
```

Use the all() method to see the updated database.

```
db.all()
```

It will display the data from the updated database:

```
[
    {'roll_number': 1, 'st_name': 'Adam', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'}
]
```

Example: Deleting the Entire Data

If you want to remove all the data from a database, you can use the **truncate()** method as follows:

```
db.truncate()
```

Next, use the all() method to see the updated database.

```
db.all()
```

It will show an empty database as the **output**:

```
[]
```

7. TinyDB – Querying

TinyDB has a rich set of queries. We have ways to construct queries: the first way resembles the syntax of ORM tools and the second is the traditional way of using the 'Where' clause.

In this chapter, let's understand these two ways of constructing a query in a TinyDB database.

The First Method: Importing a Query

The first method resembles the syntax of ORM tools in which first we need to import the query in the command prompt. After importing, we can use the query object to operate the TinyDB database. The **syntax** is given below:

```
from tinydb import Query
student = Query()
```

Here, 'student' is the name of our database. For the examples, we will be using the following **student** database.

```
[
   {
      "roll_number":1,
      "st_name":"elen",
      "mark":250,
      "subject":"TinyDB",
      "address":"delhi"
   },
   {
      "roll_number":2,
      "st_name":"Ram",
      "mark":[
         250,
         280
      ],
      "subject":[
         "TinyDB",
         "MySQL"
      ],
```

```
      "address":"delhi"
   },
   {
      "roll_number":3,
      "st_name":"kevin",
      "mark":[
         180,
         200
      ],
      "subject":[
         "oracle",
         "sql"
      ],
      "address":"keral"
   },
   {
      "roll_number":4,
      "st_name":"lakan",
      "mark":200,
      "subject":"MySQL",
      "address":"mumbai"
   },
   {
      "roll_number":5,
      "st_name":"karan",
      "mark":275,
      "subject":"TinyDB",
      "address":"benglore"
   }
]
```

Example

Following is the query to retereive the data from the **student** database where the **roll_no** of the students are less than 3:

```
>>> db.search(Query().roll_number < 3)
```

Or,

```
>>> student = Query()
>>> db.search(student.roll_number < 3)
```

The above search query will produce the following result:

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'}
]
```

Sometimes the file name is not a valid Python identifier. In that case, we would not be able to access that field. For such cases, we need to switch to **dict access notation** as follows:

```
student = Query();

# Invalid Python syntax
db.search(student.security-code == 'ABCD')

# Use the following dict access notation
db.search(student['security-code'] == 'ABCD')
```

The Second Method: Using the "where" Clause

The second way is the traditional way of constructing queries that uses the "where" clause. The **syntax** is given below:

```
from tinydb import where
db.search(where('field') == 'value')
```

Example

TinyDB "**where**" clause for the **subject** field:

```
db.search(where('subject') == 'MySQL')
```

The above query will produce the following **output**:

```
[{'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'}]
```

8. TinyDB – Searching

TinyDB provides the **search()** method to help you search any data from a document. Along with the **query()** object, the search() method can be used to find the data in a JSON file. We have various ways in which we can use the search() method on a TinyDB database.

Method 1: TinyDB search() with Existence of a Field

We can search the data from a database based on the existence of a field. Let's understand it with an example. For this and other examples, we will be using the following **student** database.

```
[
   {
      "roll_number":1,
      "st_name":"elen",
      "mark":250,
      "subject":"TinyDB",
      "address":"delhi"
   },
   {
      "roll_number":2,
      "st_name":"Ram",
      "mark":[
         250,
         280
      ],
      "subject":[
         "TinyDB",
         "MySQL"
      ],
      "address":"delhi"
   },
   {
      "roll_number":3,
      "st_name":"kevin",
      "mark":[
```

```
         180,
         200
      ],
      "subject":[
         "oracle",
         "sql"
      ],
      "address":"keral"
   },
   {
      "roll_number":4,
      "st_name":"lakan",
      "mark":200,
      "subject":"MySQL",
      "address":"mumbai"
   },
   {
      "roll_number":5,
      "st_name":"karan",
      "mark":275,
      "subject":"TinyDB",
      "address":"benglore"
   }
]
```

Example

The search query based on the existence of a field is as follows:

```
from tinydb import Query
student = Query()
db.search(student.address.exists())
```

The above query will retrieve the following data from the **student** file:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
```

```
{'roll_number': 2,
 'st_name': 'Ram',
 'mark': [250, 280],
 'subject': ['TinyDB', 'MySQL'],
 'address': 'delhi'},
{'roll_number': 3,
 'st_name': 'kevin',
 'mark': [180, 200],
 'subject': ['oracle', 'sql'],
 'address': 'keral'},
{'roll_number': 4,
 'st_name': 'lakan',
 'mark': 200,
 'subject': 'MySQL',
 'address': 'mumbai'},
{'roll_number': 5,
 'st_name': 'karan',
 'mark': 275,
 'subject': 'TinyDB',
 'address': 'benglore'}]
```

Method 2: TinyDB search() with Regular Expression

We can search for a particular data from a database using regular expression (Regex). Let's understand how it works with a couple of examples.

Example 1

Full item search matching the Regular Expression:

```
from tinydb import Query
student = Query()
db.search(student.st_name.matches('[aZ]*'))
```

This query will produce the following **output**:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
```

```
{'roll_number': 2,
 'st_name': 'Ram',
 'mark': [250, 280],
 'subject': ['TinyDB', 'MySQL'],
 'address': 'delhi'},
{'roll_number': 3,
 'st_name': 'kevin',
 'mark': [180, 200],
 'subject': ['oracle', 'sql'],
 'address': 'keral'},
{'roll_number': 4,
 'st_name': 'lakan',
 'mark': 200,
 'subject': 'MySQL',
 'address': 'mumbai'},
{'roll_number': 5,
 'st_name': 'karan',
 'mark': 275,
 'subject': 'TinyDB',
 'address': 'benglore'}]
```

Example-2

Case-sensitive search with Regular Expression:

```
from tinydb import Query
import re
student = Query()
db.search(student.st_name.matches('lakan', flags=re.IGNORECASE))
```

It wil produce the following **output**:

```
[{'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'}]
```

Example-3

Any part of the item matching with Regular Expression:

```
from tinydb import Query
student = Query()
db.search(student.st_name.search('r+'))
```

This query will produce the following **output**:

```
[{'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

Method 3: TinyDB search() using a Substring

We can also use a substring while searching for a particular data from a TinyDB database. Let's understand how it works with a couple of examples:

Example-1

Take a look at this query; it will fetch the all the rows where the "address" field is "delhi".

```
from tinydb import Query
student = Query()
db.search(student['address'] == 'delhi')
```

It will produce the following **output**:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'}]
```

Example-2

In this query, we have used a slightly different syntax for the search() method.

```
from tinydb import Query
student = Query()
db.search(student.address.search('mumbai'))
```

It will fetch all the rows where the "address" field is "mumbai".

```
[{'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'}]
```

9. TinyDB – The where Clause

TinyDB provides the "where" clause that you can use while searching for a particular data. The "where" clause helps by filtering the unwanted data out. With the help of the "where" clause, you can access specific data quickly.

Before using the 'where' clause, we need to first import it. The **syntax** of "where" clause is given below:

```
from tinydb import where
db.search(where('field') == 'value')
```

Let's understand the use of 'where' clause with the help of a couple of examples.

The Student Database

For the examples, we will use the following **student** database.

```
[
   {
      "roll_number":1,
      "st_name":"elen",
      "mark":250,
      "subject":"TinyDB",
      "address":"delhi"
   },
   {
      "roll_number":2,
      "st_name":"Ram",
      "mark":[
         250,
         280
      ],
      "subject":[
         "TinyDB",
         "MySQL"
      ],
      "address":"delhi"
   },
   {
```

```
      "roll_number":3,
      "st_name":"kevin",
      "mark":[
         180,
         200
      ],
      "subject":[
         "oracle",
         "sql"
      ],
      "address":"keral"
   },
   {
      "roll_number":4,
      "st_name":"lakan",
      "mark":200,
      "subject":"MySQL",
      "address":"mumbai"
   },
   {
      "roll_number":5,
      "st_name":"karan",
      "mark":275,
      "subject":"TinyDB",
      "address":"benglore"
   }
]
```

Example 1

Let's use the "**where**" clause for the **subject** field:

```
db.search(where('subject') == 'MySQL')
```

This query will fetch all the rows where the "subject" field is "MySQL".

```
[{
    'roll_number': 4,
    'st_name': 'lakan',
    'mark': 200,
```

```
    'subject': 'MySQL',
    'address': 'mumbai'
}]
```

Example 2

Let's see another use of the "**where**" clause with the "**not equal to**" condition:

```
db.search(where('mark') != 275)
```

This query will fetch all the rows where the "mark" field is not equal to "275":

```
[
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'}
]
```

10. TinyDB – The exists() Query

TinyDB provides an advanced query called **exists()** that checks the existence of data in a JSON file. The **exists()** query actually tests the availability of a subfield data from a JSON file. The exists() query works on the basis of a Boolean condition. If the subfield exists (i.e., BOOLEAN TRUE), it will fetch the data accordingly from the JSON file, otherwise it will return a blank value.

Syntax

The syntax of TinyDB exists() is as follows:

```
db.search(Query().field.exists())
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created from the JSON table **student**.

We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's use the TinyDB **exists()** query for the field named '**subject**':

```
db.search(student.subject.exists())
```

This query will fetch all the rows because all the rows have the "subject" field:

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},

{'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},

 {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},

{'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},

{'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': ' TinyDB ',
'address': 'benglore'}]
```

Example 2

Now let's use the **exists()** query for the '**address**' field:

```
db.search(student.address.exists())
```

It will fetch the following rows:

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},

{'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},

 {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},

{'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},

{'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': ' TinyDB ',
'address': 'benglore'}]
```

Example 3

Now, let's try the **exists()** query for a field that is not available:

```
db.search(student.city.exists())
```

Since none of the rows in the given table has a field called "city", the above exists() query will return a blank value:

```
[]
```

11. TinyDB – The matches() Query

The matches() query matches the data from a JSON file with a given condition (in the form of a regular expression) and returns the results accordingly. It will return a blank value if the condition does not match with the data in the file.

Syntax

The syntax of TinyDB **matches()** is as follows:

```
db.search(Query().field.matches(regular expression))
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**.

Let's understand how it works with the help of a couple of examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can use **matches()** for full item search.

```
from tinydb import Query
student = Query()
db.search(student.st_name.matches('[aZ]*'))
```

This query will fetch all the rows:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
```

```
  'subject': ['oracle', 'sql'],
  'address': 'keral'},
 {'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

Example 2

Let's see how we can use matches() for case-sensitive search.

```
from tinydb import Query
import re
student = Query()
db.search(student.st_name.matches('ram', flags=re.IGNORECASE))
```

This query will fetch the rows where the student name matches the string "ram". Observe that we have used a flag to ignore the case while matching the strings.

```
[{'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'}]
```

Example 3

Let's see how we can use matches() for a particular item.

```
student = Query()
db.search(student.address.matches('keral'))
```

This query will fetch the rows where the address matches the string "keral".

```
[{'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'}]
```

Example 4

Let's see what matches() would return when it does not find a particular item:

```
student = Query()

db.search(student.address.matches('Ratlam'))
```

There are no rows where the "address" field matches the string "Ratlam", hence it will return a blank value:

```
[]
```

12. TinyDB – The test() Query

The **test()** query will test if the given arguments match with the data in a table. If it matches with the data, it will return the matched data, otherwise it will return blank. First of all, we need to define a **test** function and its arguments and then it will search the item in a given database.

Syntax

The syntax of TinyDB **test()** is as follows:

```
db.search(Query().field.test(function or condition, *arguments))
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**.

We can create a custom test function as follows:

```
object = lambda t: t == 'value'
```

Here the **lamba** keyword is important to create the custom test function.

Let's understand how it works with the help of a couple of examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

We will first create a test function and then use it in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
objects = lambda t: t == [250, 280]
db.search(Query().mark.test(objects))
```

It will fetch the rows where the "mark" field has the values [250, 280]:

```
[{'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'}]
```

Example 2

In this example, we will use the "subject" field in the test function:

```
student = Query()
db = TinyDB('student.json')
objects = lambda t: t == 'TinyDB'
db.search(student.subject.test(objects))
```

This query will fetch all the rows where the "subject" field has the value "TinyDB":

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

13. TinyDB – The any() Query

For searching the fields containing a list, TinyDB provides a method called **any()**. This method matches at least one given value from the database. It finds either an entire list or a minimum one value as per the query provided.

Syntax

The syntax of TinyDB **any()** is as follows:

```
db.search(Query().field.any(query|list)
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**.

- If we will provide **query** as the argument of **any()** method, it will match all the documents where at least one document in the list field match the given query.
- On the other hand, if we will provide ***list*** as the argument of **any()** method, it will match all the documents where at least one document in the list field is present in the given list.

Let's understand how it works with the help of a couple of examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can find the fields from our student table where subject is either TinyDB, or MySQL, or SQL or combination of any two or three:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().subject.any(['TinyDB', 'MySQL', 'oracle']))
```

The above query will fetch all the rows where the "subject" field contains **any** of the following values: "TinyDB", "MySQL", or "oracle":

```
[{'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'}]
```

Example 2

Let's see how the **any()** method reacts when it doesn't match anything from the given list:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().subject.any(['Oracle']))
```

This query will return a blank value because there are no rows with its "subject" as "Oracle".

```
[]
```

Example 3

Observe that it is case-sensitive. The "subject" field does not have "**Oracle**", but it does have "**oracle**". Try the following query:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().subject.any(['oracle']))
```

It will fetch the following row:

```
[{'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'}]
```

As it is case-sensitive, it returned a blank value in the previous example because there are no rows with its "subject" as "Oracle".

14. TinyDB – The all() Query

TinyDB provides a method called **all()** that finds an entire list of values as per the query provided. Let's take an example and find out how it works.

Syntax

The syntax of TinyDB **all()** is as follows:

```
db.search(Query().field.all(query|list)
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**.

- If we will provide a **query** as the argument of **all()** method, it will match all the documents where all documents in the list field match the given query.
- On the other hand, if we will provide a **list** as the argument of **all()** method, it will match all the documents where all documents in the list field are present in the given list.

Let's understand how it works with the help of a couple of examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can find the fields from our student table where the subjects are both TinyDB, and MySQL:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().subject.all(['TinyDB', 'MySQL']))
```

This query will fetch the following row:

```
[{'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'}]
```

Example 2

Let's see how we can use **all()** to get the entire data from our database:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.all()
```

It will fetch all the rows from the linked database:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'},
 {'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

15. TinyDB – The one_of() Query

For matching the subfield data, TinyDB provides a method called **one_of()**. This method searches a single category and gets at least one similar value. It will match if the field is contained in the provided list.

Syntax

The syntax of TinyDB **one_of()** is as follows:

```
db.search(Query().field.one_of(list)
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**. It will fetch either single or multiple values of one category.

Let's understand how it works with the help of a couple examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can find the fields from our **student** table where **address** is either "delhi" or "pune":

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().address.one_of(['delhi', 'pune']))
```

It will fetch all the rows where the "address" field contains either "delhi" or "pune".

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'}]
```

Example 2

Let's see another example with **'subject'** field:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(Query().subject.one_of(['TinyDB', 'MySQL']))
```

It will fetch all the rows where the "subject" field contains either "TinyDB" or "MySQL".

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 4,
  'st_name': 'lakhan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

16. TinyDB – Logical Negate

Logical Negate works as an inverse logical gate. It will match the documents that don't match the given query. In simple words, it will display the opposite meaning of the given command.

Syntax

The syntax of TinyDB **Logical Negate** is as follows:

```
db.search(~(Query().field)
```

Here**, field** represents the part of data that we want to access. **Query()** is the object created of our JSON table named **student**. It will fetch the data that represents the opposite meaning of the given command.

Let's take a couple of examples and see how it works. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can find the fields from our **student** table where the student name is not **'elen'**:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(~(Query().st_name == 'elen'))
```

The above query will fetch all the rows where the student name is not "elen".

```
[{'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'},
 {'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
```

```
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

Example 2

Let's see how we can avoid a particular address using logical negate:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search(~(student.address.one_of(['keral', 'delhi'])))
```

This query will fetch all the rows where the "address" field does not have either "keral" or "delhi".

```
[{'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

17. TinyDB – Logical AND

The "Logical AND" operator combines multiple conditions and evaluates to True if all the conditions are met. TinyDB Logical AND operates on two queries of a database. If both the queries are True, TinyDB will fetch the required data. On the other hand, if any one of the queries is False, it will return a blank.

Syntax

The syntax of TinyDB **`Logical AND`** is as follows:

```
db.search((Query().(query1) & (Query().(query2)
```

Here**, field** represents the part of data that we want to access. **`Query()`** is the object created of our JSON table named **student**. It will fetch the data if both the conditions met, otherwise it will return a blank.

Let's take a couple examples and see how Logial AND works. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see what our TinyDB Student database returns when we apply Logical AND on "st_name=lakhan" and "subject=MYSQL" field:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search ((Query().st_name == 'lakhan') & (Query().subject == 'MySQL'))
```

This query will fetch only those rows where the student name is "lakhan" and the "subject" is "MySQL".

```
[{'roll_number': 4,
  'st_name': 'lakhan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'}]
```

Example 2

In this example, let's apply Logical AND on the "subject" and "roll_number" fields:

```
from tinydb import TinyDB, Query
student = Query()
db = TinyDB('student.json')
db.search((student.subject.search('M')) & (student.roll_number < 5))
```

This query will fetch all the rows where the roll_number is less than "4" and "subject" starts with the letter "M".

```
[{'roll_number': 4,
  'st_name': 'lakhan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'}]
```

18. TinyDB – Logical OR

The "Logical OR" operator combines multiple conditions and evaluates to True only if either of the condition is met. TinyDB Logical OR operates on two queries of a database. If any one of the queries is True, TinyDB will fetch the required data. On the other hand, if both the queries are False, it will return a blank.

Syntax

The syntax of TinyDB **`Logical OR`** is as follows:

```
db.search((Query().(query1) | (Query().(query2)
```

Here**, field** represents the part of data that we want to access. **`Query()`** is the object created of our JSON table named **student**. It will fetch the data if any one of the conditions is met, otherwise it will return a blank.

Let's take a couple of examples and see how it works. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see what our TinyDB Student database returns when we apply Logical OR on the "st_name" and "subject" fields and check the following conditions: "st_name=lakhan" and "subject=TinyDB":

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search ((Query().st_name == 'lakan') | (Query().subject == 'TinyDB'))
```

This query will fetch the following rows:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 4,
  'st_name': 'lakhan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
```

```
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'}]
```

Example 2

In this example, let's apply Logical OR on the "subject" and "roll_number" fields:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.search((student.subject.search('M')) | (student.roll_number < 5))
```

This query will fetch all the rows where the "subject" field starts with the letter "M" or the "roll_number" is less than "5".

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'},
 {'roll_number': 4,
  'st_name': 'lakhan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'}]
```

19. TinyDB – Handling Data Query

TinyDB – Storing Multiple Data

We have already discussed how you can use the **'insert'** query to store data in a database. On a similar note, you can use the **'insert_multiple'** query to store multiple data items simultaneously. Here is the syntax of **'insert_multiple'** query in TinyDB:

```
db.insert_multiple ([{ key1 : value1, key2 : value2, ..., keyN : valueN}, {
key1 : value1, key2 : value2, ..., keyN : valueN }])
```

Let's take a couple of examples to demonstrate how the "insert_multiple" query works. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can insert two records of students in our **'student'** table using the **insert_multiple** query:

```
from tinydb import TinyDB, Query


db = TinyDB('student.json')


db.insert_multiple([
    {'roll_number': 6, 'st_name': 'Siya', 'mark': 240, 'subject': 'NoSQL',
'address': 'pune'},
    {'roll_number': 7, 'st_name': 'Adam', 'mark': 210, 'subject': 'oracle',
'address': 'Keral'}
])
```

It will display the document IDs of the newly saved records:

```
[6, 7]
```

Let's check whether the new records are saved in the database or not? Use the **all()** method, as shown below:

```
db.all()
```

It will show all the records stored in the given table:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
```

```
 'subject': 'TinyDB',
 'address': 'delhi'},
{'roll_number': 2,
 'st_name': 'Ram',
 'mark': [250, 280],
 'subject': ['TinyDB', 'MySQL'],
 'address': 'delhi'},
{'roll_number': 3,
 'st_name': 'kevin',
 'mark': [180, 200],
 'subject': ['oracle', 'sql'],
 'address': 'keral'},
{'roll_number': 4,
 'st_name': 'lakan',
 'mark': 200,
 'subject': 'MySQL',
 'address': 'mumbai'},
{'roll_number': 5,
 'st_name': 'karan',
 'mark': 275,
 'subject': 'TinyDB',
 'address': 'benglore'},
{'roll_number': 6,
 'st_name': 'Siya',
 'mark': 240,
 'subject': 'NoSQL',
 'address': 'pune'},
{'roll_number': 7,
 'st_name': 'Adam',
 'mark': 210,
 'subject': 'oracle',
 'address': 'Keral'}]
```

You can see the two new records of **students** have been saved in the database.

Example 2

Let's see how we can use **insert_multiple** with a **for** loop to insert multiple values simultaneously in a table. Use the following code:

```
db.insert_multiple({'roll_number': 10, 'numbers': r} for r in range(3))
```

It will return the document IDs of the newly saved records:

```
[8, 9, 10]
```

Again, use the **all()** method to verify whether the new records have been saved in the database or not?

```
db.all()
```

It will fetch all the records stored in the given **student** table:

```
[{'roll_number': 1,
  'st_name': 'elen',
  'mark': 250,
  'subject': 'TinyDB',
  'address': 'delhi'},
 {'roll_number': 2,
  'st_name': 'Ram',
  'mark': [250, 280],
  'subject': ['TinyDB', 'MySQL'],
  'address': 'delhi'},
 {'roll_number': 3,
  'st_name': 'kevin',
  'mark': [180, 200],
  'subject': ['oracle', 'sql'],
  'address': 'keral'},
 {'roll_number': 4,
  'st_name': 'lakan',
  'mark': 200,
  'subject': 'MySQL',
  'address': 'mumbai'},
 {'roll_number': 5,
  'st_name': 'karan',
  'mark': 275,
  'subject': 'TinyDB',
  'address': 'benglore'},
 {'roll_number': 6,
  'st_name': 'Siya',
```

```
 'mark': 240,
 'subject': 'NoSQL',
 'address': 'pune'},
{'roll_number': 7,
 'st_name': 'Adam',
 'mark': 210,
 'subject': 'oracle',
 'address': 'Keral'},
{'roll_number': 10, 'numbers': 0},
{'roll_number': 10, 'numbers': 1},
{'roll_number': 10, 'numbers': 2}]
```

Notice the last three rows. We have used the **insert_multiple** method with a **for** loop to insert three new entries.

20. TinyDB – Modifying the Data

We have already discussed the **update** query with the help of which we can modify the values as well as handle the data in our database. But the **update** query such as **db.update(fields, query)** allows us to update a document by adding or overwriting its values.

But sometimes, we would like to remove one field or need to increment its value. In such cases, we can pass a function instead of fields. We can use the following operations with the update query:

The Increment Query

The increment query, as its name implies, is used to increment the value of a key in the database. The **syntax** of increment query is as follows:

```
from tinydb.operations import increment
db.update(increment('key'))
```

The Add Query

The add query is used to add value to the value of a key. It works for the strings as well. The **syntax** of add query is as follows:

```
from tinydb.operations import add
db.update(add(key, value))
```

The Set Query

This query is used to set the key to the value of the data. The **syntax** of set query is as follows:

```
from tinydb.operations import set
db.update(set(key, value))
```

The Decrement Query

The decrement query is used to decrement the value of a key. The **syntax** of decrement query is as follows:

```
from tinydb.operations import decrement
db.update(decrement(key))
```

The Subtract Query

The subtarct query is used to subtract value from the value of a key. The **syntax** of subtract query is as follows:

```
from tinydb.operations import subtract
db.update(subtract(key, value))
```

The Delete Query

The delete query is used to delete a key from a document. The **syntax** of delete query is as follows:

```
from tinydb.operations import delete
db.update(delete(key))
```

Let's take a couple of examples to demonstrate how you can use these operations along with the **update** query. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can **increment** the marks of a student in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
from tinydb.operations import increment
db.update(increment('mark'), Query().mark == 275)
```

It will produce the following **output**:

```
[5]
```

The above output shows that it has updated the record whose document ID is 5.

Example 2

Let's see how we can **add** 5 marks to the marks of a student in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
from tinydb.operations import add
db.update(add('mark', 5), Query().mark == 200)
```

It will produce the following **output**:

```
[4]
```

The above output shows that it has updated the record whose document ID is 4.

Example 3

Let's see how we can **set** the marks to 259 where the marks of a student are 250 in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
from tinydb.operations import add
db.update(add('mark', 259), Query().mark == 250)
```

It will display the following **output**:

```
[1]
```

The above output shows that it has updated the record whose document ID is 1.

Example 4

Let's see how we can **decrement** the marks of a student in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
from tinydb.operations import decrement
db.update(decrement('mark'), Query().mark == 205)
```

It will produce the following **output**:

```
[4]
```

The above output shows that it has updated the record whose document ID is 4.

Example 5

Let's see how we can **subtract** 5 marks to the marks of a student in our **student** table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
from tinydb.operations import add
db.update(add('mark', 5), Query().mark == 204)
```

It will produce the following **output**:

```
[4]
```

The above output shows that it has updated the record whose document ID is 4.

Example 6

Let's see how we can **subtract** 5 marks to the marks of a student in our student table:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
```

```
from tinydb.operations import delete

db.update(delete('mark'), Query().mark == 209)
```

It will produce the following **output**:

```
[4]
```

The above output shows that it has updated the record whose document ID is 4. It will delete the mark field from the database where the value is 209.

Example 7

Let's see how we can update multiple values in a table with a single query:

```
from tinydb import TinyDB, Query

db = TinyDB('student.json')

from tinydb import where

db.update_multiple([ ({'st_name':'Eliana'}, where ('roll_number')==1),
({'mark':20}, where ('roll_number') ==2) ])
```

It will produce the following **output**:

```
[1, 2]
```

The above output shows that it has updated two records whose document IDs are 1 and 2.

21. TinyDB – Upserting Data

We discussed the **update** and **insert** queries, but sometimes, we a need a mix of both update and insert. In such cases, TinyDB provides a unique **upsert** query. This query helps us to insert and update data at a time as per the condition. It works in two ways:

- If data is available, then it chooses the **update** operation.
- If data is not available, then it chooses the **insert** operation.

Syntax

The syntax of **upsert** query is as follows:

```
db.upsert({ 'key' : 'value',  'logged – in' : True}, regular expression)
```

Let's take a couple of examples to demonstrate how you can use this **upsert** query in TinyDB. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can use the **upsert** query to change the address of a student to "surat", whose roll numbe is 2. In this case, we we have a matching user, hence it will update with the address to have logged-in set to True:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.upsert({'address':'Surat'}, Query().roll_number==2)
```

It will produce the following **output**, which means record number "2" got updated.

```
[2]
```

Use the following code to verify whether record number "2" got updated or not:

```
db.get(doc_id=2)
```

It will show the updated data:

```
{
    'roll_number': 2,
    'st_name': 'Ram',
    'mark': [250, 280],
    'subject': ['TinyDB', 'MySQL'],
    'address': 'Surat'
```

```
}
```

Example 2

Let's see how we can use the **upsert** query for unavailable data in our table:

```
from tinydb import TinyDB, Query

db = TinyDB('student.json')

db.upsert({'E-mail':'ram@gmail.com','logged-in': True},
Query().roll_number==2)
```

It will show the following output, which means the document with the ID "2" got updated.

```
[2]
```

Use the following code to verify the updated values:

```
db.get(doc_id=2)
```

It will produce the following output:

```
{
    'roll_number': 2,
    'st_name': 'Ram',
    'mark': [250, 280],
    'subject': ['TinyDB', 'MySQL'],
    'address': 'Surat',
    'logged-in': True,
    'E-mail': 'ram@gmail.com'
}
```

Notice that we used the **upsert** query to create a new key (**E-mail**) which was non-existent and supplied it with the value "**ram@gmail.com**".

22. TinyDB – Retrieving Data

After creating a database, we need to frequently retrieve the data from that database. Following are the ways by which we can reterive the data from a database:

The len() Query

With the help of this query, we can get the number of documents in a database. Its **syntax** is as follows:

```
len(db)
```

The get Query

The **get** query is used to reterive specific documents matching a query. Its **syntax** is as follows:

```
db.get(query)
```

The contains Query

The **contains** query is used to check whether the database contains a matching value or not. Its **syntax** is as follows:

```
db.contains(query)
```

The count Query

The **count** query is used to retrieve the number of matching documents from a database. Its **syntax** is as follows:

```
db.count(query)
```

Let's take a few examples to understand how these queries work in TinyDB. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can use the **len()** query to get the number of documents in our database:

```
from tinydb import TinyDB
db = TinyDB('student.json')
print ("Number of documents in student db: ", len(db))
```

It will show the number of documents present in the specified database:

```
Number of documents in student db: 5
```

Example 2

Let's see how we can use the **get()** query to get a specific document from our database:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.get(Query().address == 'delhi')
```

This query will fetch the row where the "address" field has the value "delhi".

```
{
    'roll_number': 1,
    'st_name': 'elen',
    'mark': 250,
    'subject': 'TinyDB',
    'address': 'delhi'
}
```

Example 3

Let's see how we can use the **contains()** query to verify if our database contains a specific value:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
db.contains(Query().address == 'delhi')
```

The contains() query returns a Boolean value, based on the existence of the specified value in the given database. In this case, it will return "True" because our database does have a "address" key with the value "delhi".

```
True
```

Example 4

Let's see how we can use the **count()** query to get the number of documents for which a given condition is True:

```
from tinydb import TinyDB, Query
db = TinyDB('student.json')
print (db.count(Query().subject == 'NoSQL'))
```

It will return the following **output**:

```
3
```

It means there are 3 documents in the database where the "subject" key has the value "NoSQL".

23. TinyDB – Document ID

TinyDB uses document ID, represented by **doc_id**, to access as well as modify the value of documents in a database. Here we will see how we can use this document_id for various operations.

Display Data using Document ID

We can use **doc_id** in **get()** method to display the data from a database. Its **syntax** is as follows:

```
db.get(doc_id = value)
```

Check for a Document in a Database

We can use **doc_id** in **contains()** method to check if a document is present in a database or not. Its **syntax** is given below:

```
db.contains(doc_id = value)
```

Update All Documents

We can use **doc_id** in **update()** method to update all the documents in a database with the given document IDs. Here is its **syntax**:

```
db.update({key : value}, doc_ids = […])
```

Remove a Document

We can use **doc_id** in **remove()** method to remove a specific document or all the documents in a database with the given document IDs. Its **syntax** is given below:

```
db.remove(doc_ids = […])
```

Let's take a few examples to demonstrate how you can use **doc_id** in TinyDB with these methods. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Let's see how we can use **doc_id** to get the data of a specific document from a database:

```
from tinydb import TinyDB
db = TinyDB('student.json')
db.get(doc_id = 5)
```

It will fetch the data from the document with the doc_id "5".

```
{
    'roll_number': 5,
    'st_name': 'karan',
    'mark': 275,
    'subject': 'oracle',
    'address': 'benglore'
}
```

Example 2

Let's see how we can use **doc_id** to check if the database contains a document with a specific ID:

```
from tinydb import TinyDB
db = TinyDB('student.json')
db.contains(doc_id = 15)
```

Based on the availability of the document, it will return either True or False. In this case, our database does not have a document with the doc_id "15". Hence, it returns False.

```
False
```

Example 3

Let's see how we can use **doc_id** to update the documents of our database:

```
from tinydb import TinyDB
db = TinyDB('student.json')
db.update({'mark':'280'}, doc_ids = [4])
```

Here, we updated the "marks" field of the document with the doc_id "4". To check the updated data, use the following query:

```
print(db.get(doc_id=4))
```

It will display the updated data of the document with the doc_id "4":

```
{
    'roll_number': 4,
    'st_name': 'lakan',
    'mark': '280',
    'subject': 'MySQL',
    'address': 'mumbai'
}
```

Example 4

Let's see how we can use **doc_id** to remove specific documents from our database:

```
from tinydb import TinyDB
db = TinyDB('student.json')
db.remove(doc_ids = [3,4])
```

Here, we removed two documents with doc_ids "3" and "4". To verify, use the following **get()** queries:

```
db.get(doc_id=3)
db.get(doc_id=4)
```

It will show the following output:

```
None
None
```

It means that we have successfully removed the documents with doc_ids "3" and "4".

24. TinyDB – Tables

In TinyDB, we can work with multiple tables. These tables have the same properties as the TinyDB class. Let's see how we can create tables in TinyDB and apply various operations on them:

Creating Tables

It's very easy to create a table in TinyDB. Here's its syntax:

```
table_object = db.table('table name')
```

Inserting Values in a Table

To insert data in a specific table, use the following syntax:

```
table_object.insert({ 'key' : value })
```

Retreiving Values from a Table

To retrieve values from a table, use the following query:

```
table_object.all()
```

Deleting a Table from a Database

To delete a table from a database, use the drop_table() query. Here is its **syntax**:

```
db.drop_table('table name')
```

Delete Multiple Tables from a Database

To delete multiple tables from a database, use the following query:

```
db.drop_tables()
```

Let's understand how to use these queries with the help of a few examples. We will use the same **student** database that we have used in all the previous chapters.

Example 1

Use the following code to create a new table called **Student_Detail**:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
table_object = db.table('Student_Detail')
```

Example 2

Next, let's insert values in this new table **Student_Detail**:

```
from tinydb import TinyDB, Query

db = TinyDB("student.json")

table_object = db.table('Student_Detail')

table_object.insert({
    'roll_number': 1,
    'st_name': 'elen',
    'mark': 250,
    'subject': 'TinyDB',
    'address': 'delhi'
})
```

It will return the doc_id of the record inserted in the table.

```
[1]
```

To verify, use the following code:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
table_object = db.table('Student_Detail')
table_object.all()
```

It will show data contained in the Student_Detail table:

```
{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'}
```

Example 3

To see all the tables present in the database, use the following query:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
print(db.tables())
```

There are two tables inside "student.json". It will show the names of these two tables:

```
{'Student_Detail', '_default'}
```

Example 4

Let's see how we can reterive all the values from a table:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
table_object = db.table("Student_Detail")
print(table_object.all())
```

It will show the following **output**:

```
[{
    'roll_number': 1,
    'st_name': 'elen',
    'mark': 250,
    'subject': 'TinyDB',
    'address': 'delhi'
}]
```

Example 5

Let's see how we can remove a table from a database:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
db.drop_table('Student_Detail')
```

It will remove the "Student_Detail" table from the database. To remove all the tables from a database, use the "drop_tables()" query:

```
db.drop_tables()
```

It will remove all the tables from the database.

25. TinyDB – Default Table

TinyDB provides a default table in which it automatically saves and modifies the data. We can also set a table as the default table. The basic queries, methods, and operations will work on that default table. In this chapter, let's see how we can see the tables in a database and how we can set a table of our choice as the default table:

Showing the Tables in a Database

To get the list of all the tables in a database, use the following code:

```
from tinydb import TinyDB, Query
db = TinyDB("student.json")
db.tables()
```

It will produce the following **output**: We have two tables inside "student.json", hence it will show the names of these two tables:

```
{'Student_Detail', '_default'}
```

The output shows that we have two tables in our database, one is "Student_Detail" and the other "_default".

Displaying the Values of the Default Table

If you use the **all()** query, it will show the contents of the default table:

```
from tinydb import TinyDB
db = TinyDB("student.json")
db.all()
```

To show the contents of the "Student_Detail" table, use the following query:

```
from tinydb import TinyDB
db = TinyDB("student.json")
print(db.table("Student_Detail").all())
```

It will show the contents of the "Student_Detail" table:

```
[{
    'roll_number': 1,
    'st_name': 'elen',
    'mark': 250,
    'subject': 'TinyDB',
```

```
    'address': 'delhi'
}]
```

Setting a Default Table

You can set a table of your choice as the default table. For that, you need to use the following code:

```
from tinydb import TinyDB
db = TinyDB("student.json")
db.default_table_name = "Student_Detail"
```

It will set the "Student_Detail" table as the default table for our database.

26. TinyDB – Caching Query

Catching query is an advanced feature of TinyDB with the help of which it caches the query result for performance optimization. In this way, when we run the same query again, TinyDB doesn't need to read the data from the storage. We can pass the **cache_size** to the table function to optimize the query cache size.

Syntax

The syntax of TinyDB query caching is shown below:

```
table = db.table('table_name', cache_size=value)
```

Example

TinyDB creates cache size memory in given table.

```
from tinydb import TinyDB
db = TinyDB('student.json')
objects = db.table('Student_Detail', cache_size = 50)
objects.all()
```

It will produce the following **output**. Observe that cache size does not affect the table values.

```
[{
    'roll_number': 1,
    'st_name': 'elen',
    'mark': 250,
    'subject': 'TinyDB',
    'address': 'delhi'
}]
```

We can set unlimited cache size by putting "cache_size = None".

```
objects = db.table('Student_Detail', cache_size = None)
```

We can also disable the cache size by putting "cache_size = 0".

```
objects = db.table('Student_Detail', cache_size = 0)
```

To clear the cache size, use the following query:

```
db.clear_cache()
```

27. TinyDB – Storage Types

TinyDB has two types of storage: JSON and in-memory. TinyDB, by default, stores the data in JSON files. While creating a database, you need to specify the path where to store the JSON file on your computer.

Storing Data in a JSON File

First, let's see how we can use a **JSON file** to store the data:

```
from tinydb import TinyDB, where

db = TinyDB('path/to/file_name.json')
```

Example 1

In this example, we are showing how you can insert multiple documents into a JSON file:

```
from tinydb import TinyDB


db = TinyDB('storage.json')


db.insert_multiple([
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakhan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'}
])
```

Here, we inserted 5 documents inside "storage.json". To verify the records, use the following query:

```
db.all()
```

It will show the contents of "storage.json" file:

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
```

```
 {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
 {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
 {'roll_number': 4, 'st_name': 'lakhan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
 {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'}]
```

Using in-memory to Store Data

Now, let's see how we can use "**in-memory**" to store the data:

```
from tinydb import TinyDB
from tinydb.storages import MemoryStorage
object = TinyDB(storage = MemoryStorage)
TinyDB.DEFAULT_STORAGE = MemoryStorage
```

Example 2

The following example shows how you can insert multiple documents in default storage memory:

```
from tinydb import TinyDB
from tinydb.storages import MemoryStorage
object = TinyDB(storage = MemoryStorage)
TinyDB.DEFAULT_STORAGE = MemoryStorage


object.insert_multiple([
    {'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
    {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
    {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
    {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
    {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'}
])
```

To verify whether the documents have been inserted or not, use the following query:

```
object.all()
```

The following **output** shows the inserted data:

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},

 {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},

 {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},

 {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},

 {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'}]
```

28. TinyDB – Middleware

TinyDB middleware helps us to customize database storage behavior by wrapping around the existing storage. This middleware improves the performance of the database.

Caching Middleware

This middleware, as its name implies, improves the speed of a database by reducing the disk I/O. The working of CachingMiddleware is as follows:

- First, it catches all the read operations.
- Then it writes the data to the disk after a configured number of write operations.

Syntax

The syntax to use CachingMiddleware is as follows:

```
from tinydb.storages import JSONStorage
from tinydb.middlewares import CachingMiddleware
db = TinyDB('middleware.json', storage = CachingMiddleware(JSONStorage))
db.close()
```

Example

The following example shows how you can perform a basic middleware procedure in a database.

```
from tinydb import TinyDB
from tinydb.storages import JSONStorage
from tinydb.middlewares import CachingMiddleware
object = TinyDB('storage.json', storage=CachingMiddleware(JSONStorage))
object.all()
```

Output

It will produce the following output:

```
[{'roll_number': 1, 'st_name': 'elen', 'mark': 250, 'subject': 'TinyDB',
'address': 'delhi'},
 {'roll_number': 2, 'st_name': 'Ram', 'mark': [250, 280], 'subject':
['TinyDB', 'MySQL'], 'address': 'delhi'},
 {'roll_number': 3, 'st_name': 'kevin', 'mark': [180, 200], 'subject':
['oracle', 'sql'], 'address': 'keral'},
```

```
 {'roll_number': 4, 'st_name': 'lakan', 'mark': 200, 'subject': 'MySQL',
'address': 'mumbai'},
 {'roll_number': 5, 'st_name': 'karan', 'mark': 275, 'subject': 'oracle',
'address': 'benglore'}]
```

Close the database to make sure that all the data is safely written.

```
db.close()
```

29. TinyDB – Extend TinyDB

It is possible to extend TinyDB and modify its behaviour. There are four ways to do so:

- Custom middleware
- Custom storages
- Hooks and overrides
- Subclassing TinyDB and table

In this chapter, let's understand each of these methods in detail.

Custom Middleware

Sometimes the user does not want to write a new storage module. In such cases, the user can modify the behaviour of an existing storage module. Let's see an example in which we will build a custom middleware filtering out the empty items:

First, let's see the data that will go through the custom middleware:

```
{
'_default': {
    1: {'key1': 'value1'},
    2: {'key2': 'value2'},
    ……,
    N: {'keyN': 'valueN'}
},
```

Now, let's see how we can implement the custom middleware:

```
class RemoveEmptyItemsMiddleware(Middleware):

def __init__(self, storage_cls):
    super(self).__init__(storage_cls)

    def read(self):
        data = self.storage.read()
        for _default in data:
            st_name = data
            for doc_id in table:
                item = st_name
                if item == {}:
```

```
                    del st_name
                return data

    def close(self):
        self.storage.close()
```

Custom Storage

As discussed earlier, TinyDB comes with two types of storages: in-memory and JSON file storage. Along with that, TinyDB also provides an option to add our own custom storage. In the following example, let's see how we can add a YAML storage using PyYAML:

```
import yaml
class YAMLStorage(Storage):
def __init__(self, db.json):
         self. db.json = db.json
```

To read the file:

```
def read(self):
    with open(self.db.json) as handle:
        try:
            info = yaml.safe_load(handle.read())
            return info

        except yaml.YAMLError:
            return None
```

To write the file:

```
def write(self, info):
    with open(self.db.json, 'w+') as handle:
        yaml.dump(info, handle)
```

To close the file:

```
def close(self):
    pass
```

Hooks and Overrides

Sometimes, both custom storage and custom middleware cannot work in the way you want. In such cases, user can use predefined hooks and overrides to modify the behaviour of TinyDB. As an example, we will be configuring the name of the default table as follows:

```
TinyDB.default_table_name = 'student_detail'
```

We can also assign cache capacity as follows:

```
TinyDB.table_class.default_query_cache_capacity = 50
```

Subclassing TinyDB and Table

This is the last way we can use to modify the behaviour of TinyDB. As an example, we will be creating a subclass that can be used with hooks and overrides to override the default classes.

```
Class ExtendTable(Table):
    TinyDB.table_class = student_detail
```

30. TinyDB – Extensions

Given below is a set of TinyDB extensions that can be used for various operations:

- **TinyDBTimestamps:** It is an experimental extension that can be used to automatically add and update at timestamps to TinyDB documents.
- **aiotinydb:** It is a stable extension that enables us to use TinyDB in asyncio-aware contexts without slowing done the synchromous Input/Output(I/O).
- **tinydb – smartcache:** It is also a stable extension. It provides users with a smart query cache. It automatically updates the query cache so that the cache does not get invalidated after inserting, updating, or removing any documents.
- **tinydb – serialization:** It is a stable extension that provides serialization of the database object.
- **tinydb – appengine:** It provides storage for App Engine. It is also a stable extension.
- **Tinyrecord:** A stable extension that implements experimental atomic transaction support for the TinyDB NoSQL DB.
- **TinyMP:** It is also stable extension used for storage based on the MessagePack.
- **Tinymongo:** It is an experimental extension which is used for the replacement of MongoDB.
- **Tinyindex:** It is also an experimental extension of the database. As name implies, it is used for indexing data of the database.