Python REST APIs With Flask, Connexion, and SQLAlchemy – Part 3

by Philipp Acsany Nov 21, 2022 10 Comments api databases flask intermediate web-dev

Mark as Completed

Tweet Share Email

Table of Contents

Most modern web applications are powered by a **REST API** under the hood. That way, developers can separate the front-end code from the back-end logic, and users can interact with the interface dynamically. In this three-part tutorial series, you're building a REST API with the **Flask web framework**.

You've created a foundation with a basic Flask project and added endpoints, which you connected to a **SQLite database**. You're also testing your API with **Swagger UI API documentation** that you're building along the way.

In the third part of this tutorial series, you'll learn how to:

- Work with **multiple tables** in a database
- Create **one-to-many** fields in your database
- Manage **relationships** with SQLAlchemy
- Leverage **nested schemas** with Marshmallow
- Display **related objects** in the front end

You can download the code for the third part of this project by clicking the link below:

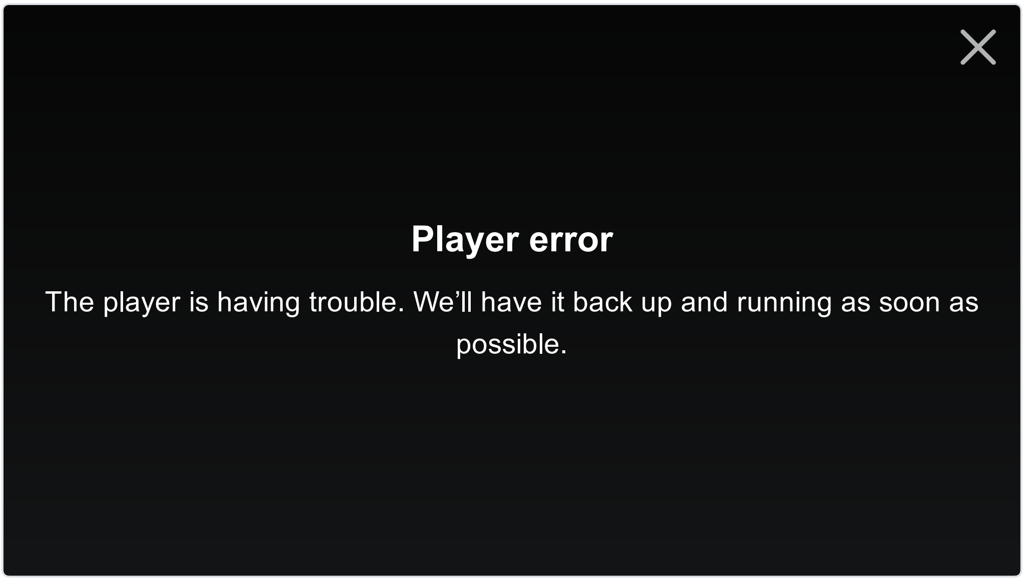
Source Code: Click here to download the free source code that you'll use to finish building a REST API with the Flask web framework.

Demo

In this three-part tutorial series, you're building a REST API to keep track of notes for people who may visit you throughout the year. You'll create people like the Tooth Fairy, the Easter Bunny, and Knecht Ruprecht.

Ideally, you want to be on good terms with all three of them. That's why you'll send them notes, to increase the chance of getting valuable gifts from them.

In this tutorial, you'll expand your programming tool belt further. You'll learn how to create hierarchical data structures represented as one-to-many relationships by SQLAlchemy. In addition, you'll also extend the REST API that you've already built to create, read, update, and delete notes for a person:



Player error

The player is having trouble. We'll have it back up and running as soon as possible.

It's time to finish this three-part tutorial series by creating relationships between people and notes!

Planning Part Three

In part one of this series, you built your REST API. Working through part two, you connected your REST API to a database. That way, your Flask application can make changes to existing data and create new data that persists even when you restart your app server.

So far, you've added the ability to save changes made through the REST API to a database using SQLAlchemy and learned how to serialize that data for the REST API using Marshmallow.

Currently, the `people.db` database only contains people data. In this part of the series, you'll add a new table to store notes. To connect notes to a person, you'll create **relationships** between the entries of the `person` table and the `note` table in your database.

You'll bootstrap `people.db` with a `build_database.py` script that contains the necessary people and notes data for you. Here's an excerpt of the dataset that you'll work with:

Python

```python
PEOPLE_NOTES = [
    {
        "lname": "Fairy",
        "fname": "Tooth",
        "notes": [
            ("I brush my teeth after each meal.", "2022-01-06 1
            ("The other day a friend said I have big teeth.", "
            ("Do you pay per gram?", "2022-03-05 22:18:10"),
        ],
    },
   # ...
]
```

You'll learn how to adjust your SQLite database to implement relationships. After that, you'll be able to translate the `PEOPLE_NOTES` dictionary into data that conforms with your database structure.

Finally, you'll show the content of your database on the home page of your app and use your Flask REST API to add, update, and delete notes that you're writing for people.

Getting Started

Ideally, you followed the first part and the second part of this tutorial series before continuing with the third part, which you're reading right now. Alternatively, you can also download the source code from part two by clicking the link below:

> **Source Code: Click here to download the free source code** that you'll use to continue building a REST API with the Flask web framework.

If you downloaded the source code from the link above, then make sure to follow the installation instructions within the provided `README.md` file.

Before you continue with the tutorial, verify that your folder structure looks like this:

```
rp_flask_api/
│
├── templates/
│   └── home.html
│
├── app.py
├── config.py
├── models.py
├── people.py
└── swagger.yml
```

Once you've got the Flask REST API folder structure in place, you can read on to check if your Flask project works as expected.

Check Your Flask Project

Before you continue working on your Flask project, it's a good idea to create and activate a virtual environment. That way, you're installing any project dependencies not system-wide but only in your project's virtual environment.

Select your **operating system** below and use your platform-specific command to set up a virtual environment:

Windows | Linux + macOS

Windows PowerShell

```
PS> python -m venv venv
PS> .\venv\Scripts\activate
(venv) PS>
```

With the commands shown above, you create and activate a virtual environment named `venv` by using Python's built-in `venv` module. The parenthesized `(venv)` in front of the prompt indicate that you've successfully activated the virtual environment.

Note: If you haven't worked through part two of this tutorial series, then make sure to download the source code by clicking the link below:

Source Code: Click here to download the free source code that you'll use to continue building a REST API with the Flask web framework.

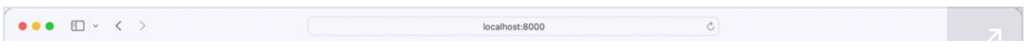
Before continuing, install the dependencies by following the instructions listed in the provided `README.md` file.

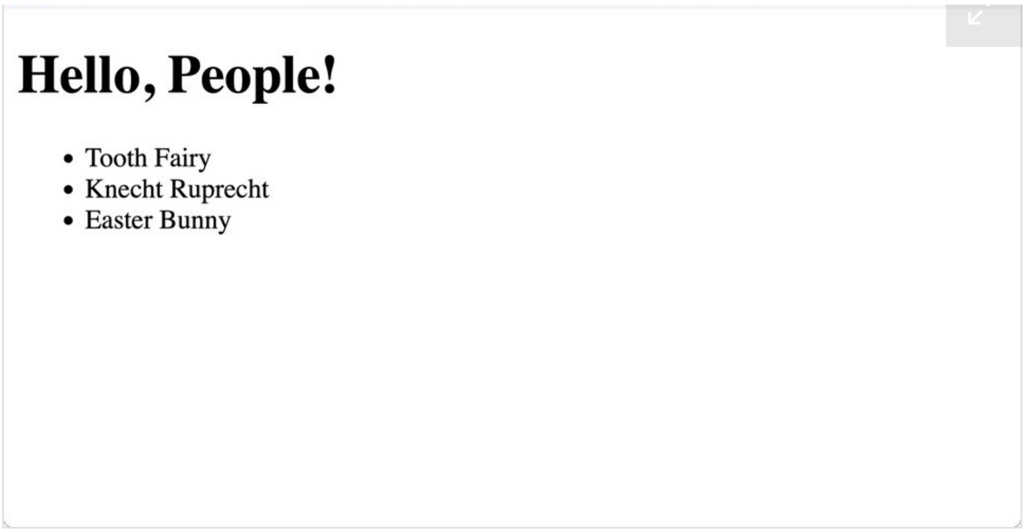
Now you can verify that your Flask application is running without errors. Execute the following command in the directory containing the `app.py` file:

Shell

```
(venv) $ python app.py
```

When you run this application, a web server will start on port 8000. If you open a browser and navigate to `http://localhost:8000`, you should see a page with the heading *Hello, People!* displayed:





Hello, People!

- Tooth Fairy
- Knecht Ruprecht
- Easter Bunny

Perfect, your app is running flawlessly! Now it's time to think about the new database structure.

Inspect the Dataset

Before starting to plan how you want to adjust your database, it's a good idea to have a look at the data that your database currently contains and the dataset that you'll work with.

The `person` table of your `people.db` database currently looks like this:

id	lname	fname	timestamp
1	Fairy	Tooth	2022-10-08 09:15:10
2	Ruprecht	Knecht	2022-10-08 09:15:13
3	Bunny	Easter	2022-10-08 09:15:27

You'll start to extend your database with a `PEOPLE_NOTES` list:

Python

```
PEOPLE_NOTES = [
    {
        "lname": "Fairy",
        "fname": "Tooth",
        "notes": [
            ("I brush my teeth after each meal.", "2022-01-06 1
            ("The other day a friend said, I have big teeth.",
            ("Do you pay per gram?", "2022-03-05 22:18:10"),
        ],
    },
    {
        "lname": "Ruprecht",
        "fname": "Knecht",
        "notes": [
            ("I swear, I'll do better this year.", "2022-01-01
            ("Really! Only good deeds from now on!", "2022-02-0
        ],
    },
    {
        "lname": "Bunny",
        "fname": "Easter",
        "notes": [
            ("Please keep the current inflation rate in mind!",
            ("No need to hide the eggs this time.", "2022-04-06
        ],
    },
]
```

Note that the `lname` values in `PEOPLE_NOTES` correspond to the contents of your `lname` column in the `person` table of your `people.db` database.

In the dataset above, each person includes a key called `notes`, which is associated with a list containing tuples of data. Each tuple in the `notes` list represents a single note containing the content and a timestamp.

Each single person is associated with multiple notes, and each single note is associated with only one person. This hierarchy of data is known as a **one-to-many** relationship, where a single parent object is related to many child objects. You'll see how this one-to-many relationship is managed in the database with SQLAlchemy later in this tutorial.

Build Relationships With People

Instead of extending the `person` table and trying to represent hierarchical data in a single table, you'll break up the data into multiple tables and connect them.

For the `person` table, this means there will be no changes. To represent the new note information, you'll create a new table called `note`.

The `note` table will look like this:

id	**person_id**	**content**	**timestamp**
1	1	I brush my teeth after each meal.	2022-01-06 17:10:24
2	1	The other day a friend said, I have big teeth.	2022-03-05 22:17:54
3	1	Do you pay per gram?	2022-03-05 22:18:10
4	2	I swear, I'll do better this year.	2022-01-01 09:15:03
5	2	Really! Only good deeds from now on!	2022-02-06 13:09:21
6	3	Please keep the current inflation rate in mind!	2022-01-07 22:47:54
7	3	No need to hide the eggs this time.	2022-04-06 13:03:17

Notice that, like the `person` table, the `note` table has a unique identifier called `id`, which is the **primary key** for the `note` table. The `person_id` column creates the relationship to the `person` table.

Whereas `id` is the primary key for the table, `person_id` is what's known as a **foreign key**. The foreign key gives each entry in the `note` table the primary key of the `person` record that it's associated with. Using this, SQLAlchemy can gather all the notes associated with each person by connecting the `person.id` primary key to the `note.person_id` foreign key, creating a relationship.

Why not stick with one table? Show/Hide

By breaking the dataset into two tables and introducing the concept of a foreign key, you'll make the data a little more complex to think about. But you'll resolve the disadvantages of a single table representation.

The biggest advantage of related tables is the fact that there's no redundant data in the database. There's only one person entry for each person you want to store in the database.

If the Easter Bunny still wants to change names, then you'll only have to change a single row in the `person` table, and anything else related to that row will immediately take advantage of the change.

Also, the column naming is more consistent and meaningful. Because person and note data exist in separate tables, the creation or update timestamp can be named consistently in both tables, as there's no conflict for names across tables.

But enough with the theory! In the next section, you'll create the models that represent the database table relationships you came up with.

Extending Your Database

In this section, you’ll extend your database. You’re going to modify the `People` data structure in `models.py` to give each person a list of notes associated with them. Finally, you’ll populate the database with some initial data.

Create SQLAlchemy Models

To use the two tables above and leverage the relationship between them, you’ll need to create SQLAlchemy models that are aware of both tables and the relationship between them.

Start by updating the `Person` model in `models.py` to include a relationship to a collection of `notes`:

Python

```python
 1  # models.py
 2
 3  from datetime import datetime
 4  from config import db, ma
 5
 6  class Person(db.Model):
 7      __tablename__ = "person"
 8      person_id = db.Column(db.Integer, primary_key=True)
 9      lname = db.Column(db.String(32), unique=True)
10      fname = db.Column(db.String(32))
11      timestamp = db.Column(
12          db.DateTime, default=datetime.utcnow, onupdate=date
13      )
14      notes = db.relationship(
15          Note,
16          backref="person",
17          cascade="all, delete, delete-orphan",
18          single_parent=True,
19          order_by="desc(Note.timestamp)"
20      )
21
22  # ...
```

In lines 14 to 20, you create a new attribute in the `Person` class called `.notes`. This new `.notes` attribute is defined in the following lines of code:

- **Line 14:** Similar to what you've done for other attributes of the class, here you create a new attribute called `.notes` and set it equal to an instance of an object called `db.relationship`. This object creates the relationship that you're adding to the `Person` class, and it's created with all of the parameters defined in the lines that follow.

- **Line 15:** The parameter `Note` defines the SQLAlchemy class that the `Person` class will be related to. The `Note` class isn't defined yet, so it won't work at the moment. Sometimes it might be easier to refer to classes as strings to avoid issues with which class is defined first. For example, you could use `"Note"` instead of `Note` here.

- **Line 16:** The `backref="person"` parameter creates what's known as a backwards reference in `Note` objects. Each instance of `Note` will contain an attribute called `.person`. The `.person` attribute references the parent object that a particular `Note` instance is associated with. Having a reference to the parent object (`Person` in this case) in the child can be very useful if your code iterates over notes and has to include information about the parent.

- **Line 17:** The `cascade="all, delete, delete-orphan"` parameter determines how to treat `Note` instances when changes are made to the parent `Person` instance. For example, when a `Person` object is deleted, SQLAlchemy will create the SQL necessary to delete the `Person` object from the database. This parameter tells SQLAlchemy to also delete all the `Note` instances associated with it. You can read more about these options in the SQLAlchemy documentation.

- **Line 18:** The `single_parent=True` parameter is required if `delete-orphan` is part of the previous `cascade` parameter. This tells SQLAlchemy not to allow an orphaned `Note` instance—that is, a `Note` without a parent `Person` object—to exist, because each `Note` has a single parent.

- **Line 19:** The `order_by="desc(Note.timestamp)"` parameter tells SQLAlchemy how to sort the `Note` instances associated with a `Person` object. When a `Person` object is retrieved, by default the `notes` attribute list will contain `Note` objects in an unknown order. The SQLAlchemy `desc()` function will sort the notes in descending order from newest to oldest, rather than the default ascending order.

Now that your `Person` model has the new `.notes` attribute, and this represents the one-to-many relationship to `Note` objects, you'll need to define a SQLAlchemy model for a `Note` object. Since you're referencing `Note` from within `Person`, add the new `Note` class right before the `Person` class definition:

Python

```python
 1  # models.py
 2
 3  from datetime import datetime
 4  from config import db, ma
 5
 6  class Note(db.Model):
 7      __tablename__ = "note"
 8      id = db.Column(db.Integer, primary_key=True)
 9      person_id = db.Column(db.Integer, db.ForeignKey("perso
10      content = db.Column(db.String, nullable=False)
11      timestamp = db.Column(
12          db.DateTime, default=datetime.utcnow, onupdate=date
13      )
14
15  class Person(db.Model):
16      # ...
17
18  # ...
```

The `Note` class defines the attributes that make up a note, as you learned in your sample `note` database table above. With this code, you define the attributes:

- **Line 6** creates the `Note` class, inheriting from `db.Model`, exactly as you did before when creating the `Person` class.

- **Line 7** tells the class what database table to use to store `Note` objects.

- **Line 8** creates the `.id` attribute, defining it as an integer value and as the primary key for the `Note` object.

- **Line 9** creates the `.person_id` attribute and defines it as the foreign key, relating the `Note` class to the `Person` class using the `.person.id` primary key. This and the `Person.notes` attribute are how SQLAlchemy knows what to do when interacting with `Person` and `Note` objects.

- **Line 10** creates the `.content` attribute, which contains the actual text of the note. The `nullable=False` parameter indicates that new notes must contain content.

- **Lines 11 to 13** create the `.timestamp` attribute, and exactly like in the `Person` class, this attribute contains the creation or update time for any particular `Note` instance.

Now that you've updated `People` and created the model for `Note`, go on to update the database.

Feed the Database

Now that you've updated `Person` and created the `Note` model, you'll use them to rebuild the `people.db` database. To do this, create a helper Python script named `build_database.py`:

Python

```python
# build_database.py

from datetime import datetime
from config import app, db
from models import Person, Note

PEOPLE_NOTES = [
    {
        "lname": "Fairy",
        "fname": "Tooth",
        "notes": [
            ("I brush my teeth after each meal.", "2022-01-06 1
            ("The other day a friend said, I have big teeth.",
            ("Do you pay per gram?", "2022-03-05 22:18:10"),
        ],
    },
    {
        "lname": "Ruprecht",
        "fname": "Knecht",
        "notes": [
            ("I swear, I'll do better this year.", "2022-01-01
            ("Really! Only good deeds from now on!", "2022-02-0
        ],
    },
    {
        "lname": "Bunny",
        "fname": "Easter",
        "notes": [
            ("Please keep the current inflation rate in mind!",
            ("No need to hide the eggs this time.", "2022-04-06
        ],
    },
]

with app.app_context():
    db.drop_all()
    db.create_all()
    for data in PEOPLE_NOTES:
        new_person = Person(lname=data.get("lname"), fname=data
        for content, timestamp in data.get("notes", []):
            new_person.notes.append(
                Note(
                    content=content,
                    timestamp=datetime.strptime(timestamp, "%Y-
                )
            )
        db.session.add(new_person)
    db.session.commit()
```

In the code above, you're feeding your project's database with the content of `PEOPLE_NOTES`. You use `db` from your `config` module so Python knows how to handle `data` and commit it to the corresponding database tables and cells.

Note: When you execute `build_database.py`, you'll re-create `people.db`. Any existing data in `people.db` will be lost.

Running the `build_database.py` program from the command line will re-create the database with the new additions, getting it ready for use with the web application:

Shell

```shell
(venv) $ python build_database.py
```

Once your project contains a fresh database, you can adjust your project to display the notes in the front end.

Displaying People With Their Notes

Now that your database contains data to work with, you can start displaying the data in both the front end and your REST API.

Show Notes in the Front End

In the previous section, you created the relationship between a person and their notes by adding a `.notes` attribute to the `Person` class.

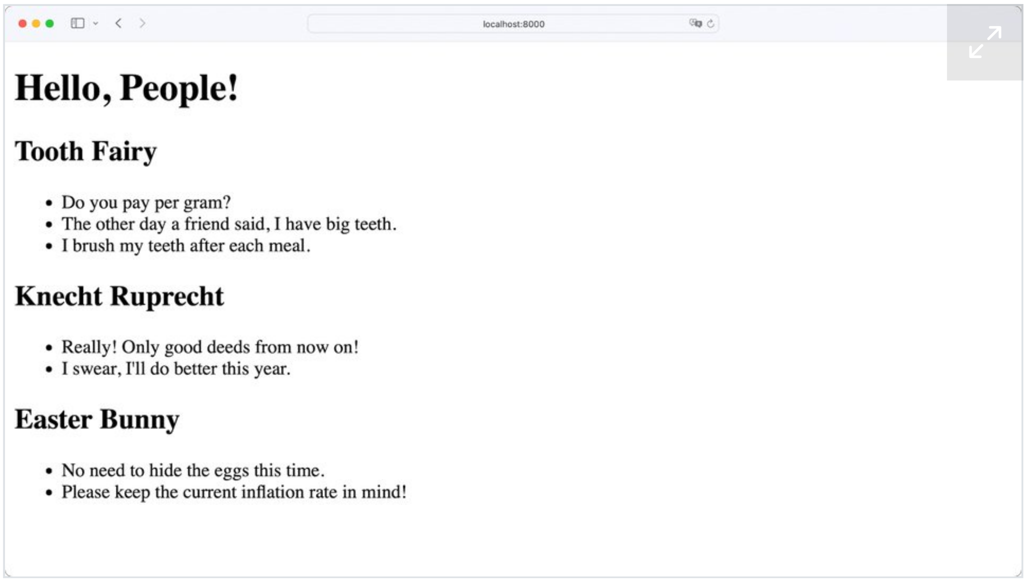
Update `home.html` in your `templates/` folder to access a person's notes:

HTML

```html
<!-- templates/home.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>RP Flask REST API</title>
</head>
<body>
    <h1>
        Hello, People!
    </h1>
    {% for person in people %}
    <h2>{{ person.fname }} {{ person.lname }}</h2>
    <ul>
        {% for note in person.notes %}
        <li>
            {{ note.content }}
        </li>
        {% endfor %}
    </ul>
    {% endfor %}
</body>
</html>
```

In the code above, you access the `.notes` attribute of each person. After that, you're looping through all the notes for a particular person to access a note's content.

Navigate to `http://localhost:8000` to check if your template renders as expected:



Hello, People!

Tooth Fairy

- Do you pay per gram?
- The other day a friend said, I have big teeth.
- I brush my teeth after each meal.

Knecht Ruprecht

- Really! Only good deeds from now on!
- I swear, I'll do better this year.

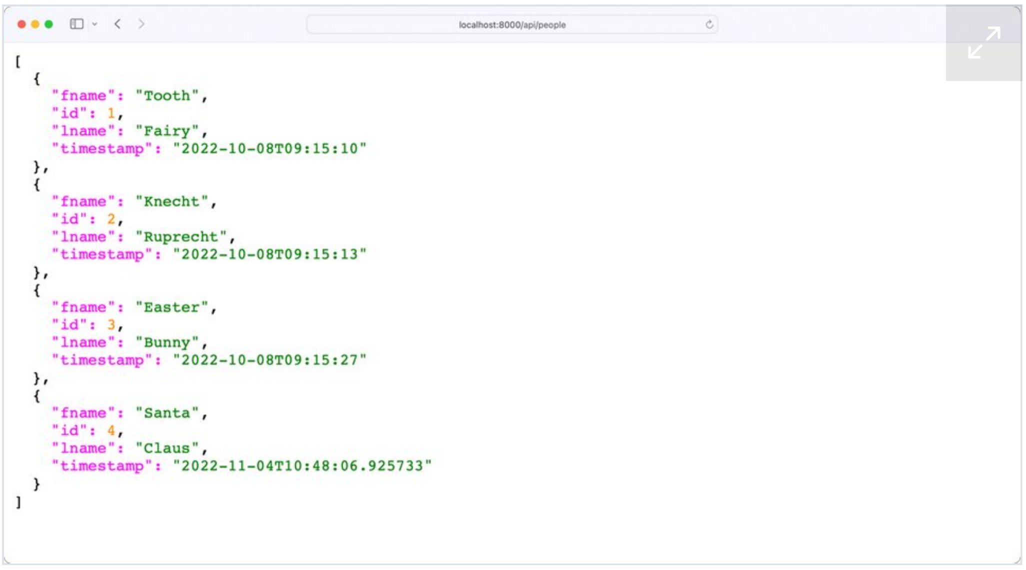
Easter Bunny

- No need to hide the eggs this time.
- Please keep the current inflation rate in mind!

Perfect, you can see the notes from each person listed in your front end. That means Flask successfully connects `Person` and `Notes` under the hood and serves you a `people` object that you can conveniently work with.

Respond With Notes

Next, check the `/api/people` endpoint of your API at `http://localhost:8000/api/people`:



```
[
  {
    "fname": "Tooth",
    "id": 1,
    "lname": "Fairy",
    "timestamp": "2022-10-08T09:15:10"
  },
  {
    "fname": "Knecht",
    "id": 2,
    "lname": "Ruprecht",
    "timestamp": "2022-10-08T09:15:13"
  },
  {
    "fname": "Easter",
    "id": 3,
    "lname": "Bunny",
    "timestamp": "2022-10-08T09:15:27"
  },
  {
    "fname": "Santa",
    "id": 4,
    "lname": "Claus",
    "timestamp": "2022-11-04T10:48:06.925733"
  }
]
```

You're receiving the people collection without any errors. However, there are no notes in the data you receive.

To investigate the issue, have a look at `read_all()` in `people.py`:

Python

```python
 1  # people.py
 2
 3  # ...
 4
 5  def read_all():
 6      people = Person.query.all()
 7      person_schema = PersonSchema(many=True)
 8      return person_schema.dump(people)
 9
10  # ...
```

The `.dump()` method in line 8 works with what it receives and doesn't filter out any data. So the issue may be in the definition of either `people` in line 6 or `person_schema` in line 7.

The query call to the database to populate `people` is exactly the same as the one in `app.py`:

Python

```python
Person.query.all()
```

This call successfully worked in the front end to show the notes for each person. This singles out `PersonSchema` as the most likely culprit.

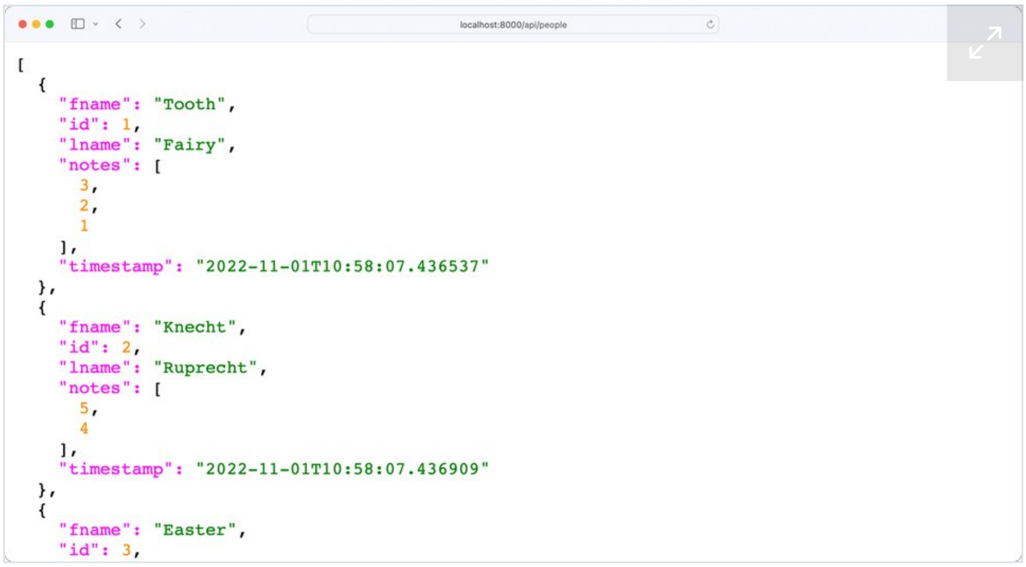
By default, a Marshmallow schema doesn't traverse into related database objects. You have to explicitly tell a schema to include relationships.

Open `models.py` and update `PersonSchema`:

Python

```python
# models.py

# ...

class PersonSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = Person
        load_instance = True
        sqla_session = db.session
        include_relationships = True
```

With `include_relationships` in the `Meta` class of `PersonSchema`, you tell Marshmallow to add any related objects to the person schema. However, the result still doesn't look as expected:



```
[
  {
    "fname": "Tooth",
    "id": 1,
    "lname": "Fairy",
    "notes": [
      3,
      2,
      1
    ],
    "timestamp": "2022-11-01T10:58:07.436537"
  },
  {
    "fname": "Knecht",
    "id": 2,
    "lname": "Ruprecht",
    "notes": [
      5,
      4
    ],
    "timestamp": "2022-11-01T10:58:07.436909"
  },
  {
    "fname": "Easter",
    "id": 3,
```

The response at `http://localhost:8000/api/people` now contains each person's notes. But instead of showing all the data a note contains, the `notes` object only contains a list of primary keys.

Create a Notes Schema

Your API response only listed the primary keys of each person’s notes. That’s fair, because you haven’t yet declared how Marshmallow should deserialize the notes.

Help Marshmallow out by creating `NoteSchema` in `models.py` beneath `Note` and above `Person`:

Python

```python
# models.py

# ...

class Note(db.Model):
    # ...

class NoteSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = Note
        load_instance = True
        sqla_session = db.session
        include_fk = True

class Person(db.Model):
    # ...

class PersonSchema(ma.SQLAlchemyAutoSchema):
    # ...

note_schema = NoteSchema()
# ...
```

You’re referencing `Note` from within `NoteSchema`, so you must place `NoteSchema` underneath your `Note` class definition to prevent errors. You also instantiate `NoteSchema` to create an object that you’ll refer to later.

Since your `Note` model contains a foreign key, you must set `include_fk` to `True`. Otherwise Marshmallow wouldn't recognize `person_id` during the serialization process.

With `NoteSchema` in place, you can reference it in `PeopleSchema`:

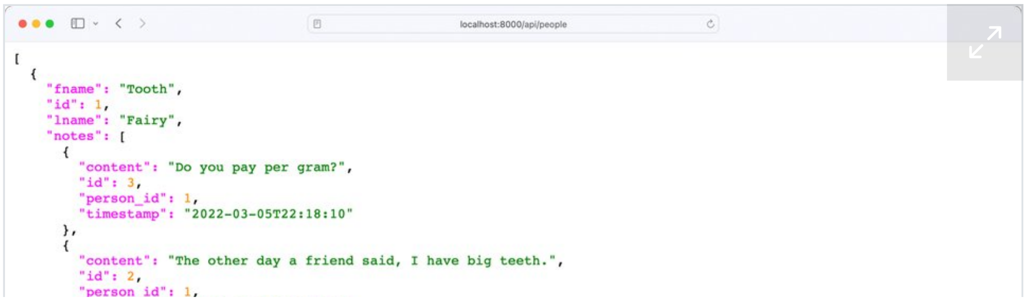
Python

```python
# models.py

from datetime import datetime
from marshmallow_sqlalchemy import fields

from config import db, ma

# ...

class PersonSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = Person
        load_instance = True
        sqla_session = db.session
        include_relationships = True

    notes = fields.Nested(NoteSchema, many=True)
```

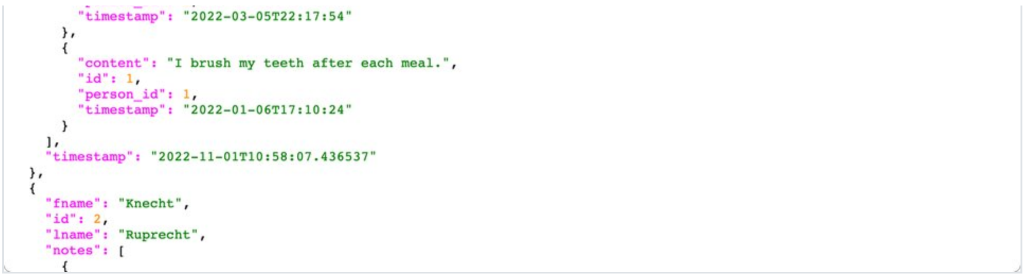
After importing `fields` from `marshmallow_sqlalchemy`, you can reference the related `Note` object by its `NoteSchema`. To avoid running into errors, verify that you defined `NoteSchema` above `PeopleSchema`.

Although you're working with `SQLAlchemyAutoSchema`, you have to explicitly create the `notes` field in `PersonSchema`. Otherwise Marshmallow doesn't receive all the information it needs to work with the `Notes` data. For example, it won't know that you're expecting a list of objects using the `many` argument.

With the changes in place, check the endpoint of your API at `http://localhost:8000/api/people`:



```
[
  {
    "fname": "Tooth",
    "id": 1,
    "lname": "Fairy",
    "notes": [
      {
        "content": "Do you pay per gram?",
        "id": 3,
        "person_id": 1,
        "timestamp": "2022-03-05T22:18:10"
      },
      {
        "content": "The other day a friend said, I have big teeth.",
        "id": 2,
        "person_id": 1,
```



```
      "timestamp": "2022-03-05T22:17:54"
    },
    {
      "content": "I brush my teeth after each meal.",
      "id": 1,
      "person_id": 1,
      "timestamp": "2022-01-06T17:10:24"
    }
  ],
  "timestamp": "2022-11-01T10:58:07.436537"
},
{
  "fname": "Knecht",
  "id": 2,
  "lname": "Ruprecht",
  "notes": [
    {
```

Perfect, your `read_all()` function not only returns all the people, but also all the notes that are attached to each person!

In the next section, you'll extend your Flask REST API to create, read, update, and delete a single note.

Handle Notes With Your REST API

You've updated the SQLAlchemy models and used them to read from the `people.db` database. Your notes are available as a nested schema in `People`. You receive the list of notes when you request a collection of people or a particular person:

Action	HTTP Verb	URL Path	Description
Read	`GET`	`/api/people`	Read a collection of people.
Read	`GET`	`/api/people/<lname>`	Read a particular person.

While you can read the notes over the endpoints shown in the table above, there's currently no way to read only one note or to manage any notes in your REST API.

Note: The URL parameters are **case sensitive**. For example, you must visit `http://localhost:8000/api/people/Ruprecht` with an uppercase R in the last name *Ruprecht*.

You can hop over to part one to recap how you built the existing `people` endpoints of your REST API. In this section of the tutorial, you’ll add additional endpoints to provide functionality to create, read, update, and delete notes:

Action	HTTP Verb	URL Path	Description
Create	`POST`	`/api/notes`	URL to create a new note
Read	`GET`	`/api/notes/<note_id>`	URL to read a single note
Update	`PUT`	`api/notes/<note_id>`	URL to update a single note
Delete	`DELETE`	`api/notes/<note_id>`	URL to delete a single note

You’ll start off by adding the functionality to read a single note. To do so, you’ll adjust your Swagger configuration file that contains your API definitions.

Read a Single Note

Currently you’re able to receive all the notes of a person when you request data from that particular person. To get information about one note, you’ll add another endpoint.

Before you add the endpoint, update your **Swagger configuration** by creating a `note_id` parameter component in the `swagger.yml` file:

YAML

```yaml
# swagger.yml
# ...

components:
  schemas:
    # ...

  parameters:
    lname:
      # ...
    note_id:
      name: "note_id"
      description: "ID of the note"
      in: path
      required: true
      schema:
        type: "integer"
# ...
```

The `note_id` in `parameters` will be part of your endpoints to identify which note you want to handle.

Continue to edit `swagger.yml` and add the data for the endpoint to read a single note:

YAML

```yaml
# swagger.yml

# ...

paths:
  /people:
    # ...
  /people/{lname}:
    # ...
  /notes/{note_id}:
    get:
      operationId: "notes.read_one"
      tags:
        - Notes
      summary: "Read one note"
      parameters:
        - $ref: "#/components/parameters/note_id"
      responses:
        "200":
          description: "Successfully read one note"
```

The structure of /notes/{note_id} is similar to /people/{lname}. You start with the get operation for the /notes/{note_id} path. The {note_id} substring is a placeholder for the ID of a note that you have to pass in as a **URL parameter**. So, for example, the URL http://localhost:8000/api/notes/1 will give you the data for the note with the primary key 1.

The operationId points to notes.read_one. That means your API expects a read_one() function in a notes.py file. Go on, create notes.py and add read_one():

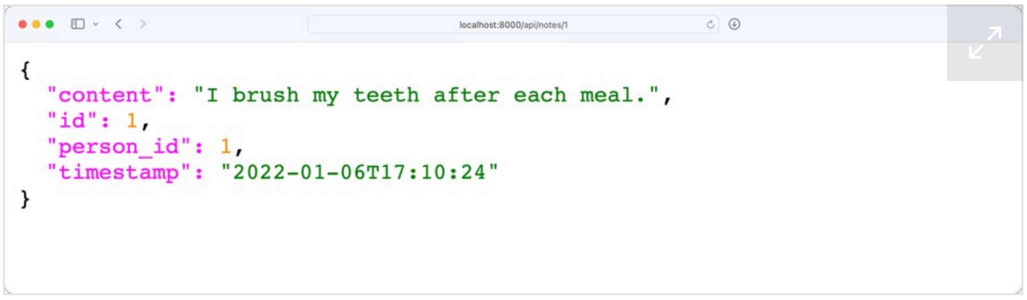
Python

```python
# notes.py

from flask import abort, make_response

from config import db
from models import Note, note_schema

def read_one(note_id):
    note = Note.query.get(note_id)

    if note is not None:
        return note_schema.dump(note)
    else:
        abort(
            404, f"Note with ID {note_id} not found"
        )
```

Although you're not using make_response() and db yet, you can go ahead and add them to your imports already. You'll use them in a bit when you're writing to the database.

For now, you're only reading from the database with the note_id parameter from the REST URL path. You use note_id in the query's .get() method to get the note with the primary key of the note_id integer.

If a note is found, then note contains a Note object and you return the serialized object. Go ahead and try it out by visiting `http://localhost:8000/api/notes/1` in your browser:


localhost:8000/api/notes/1

```
{
  "content": "I brush my teeth after each meal.",
  "id": 1,
  "person_id": 1,
  "timestamp": "2022-01-06T17:10:24"
}
```

Perfect, the API response with the note dataset looks exactly as expected! Next, you'll use the same endpoint to update and delete a note.

Update and Delete a Note

This time, you start by creating the functions in `notes.py` first, before creating the operations in `swagger.yml`.

Add `update()` and `delete()` to `notes.py`:

```python
# notes.py

# ...

def update(note_id, note):
    existing_note = Note.query.get(note_id)

    if existing_note:
        update_note = note_schema.load(note, session=db.sessior
        existing_note.content = update_note.content
        db.session.merge(existing_note)
        db.session.commit()
        return note_schema.dump(existing_note), 201
    else:
        abort(404, f"Note with ID {note_id} not found")

def delete(note_id):
    existing_note = Note.query.get(note_id)

    if existing_note:
        db.session.delete(existing_note)
        db.session.commit()
        return make_response(f"{note_id} successfully deleted",
    else:
        abort(404, f"Note with ID {note_id} not found")
```

When you compare `update()` with `delete()`, they share a similar structure. Both functions look for an existing note and work with a database session.

For `update()` to work, you also accept a `note` object as an argument, which contains the `.content` attribute that you may update.

In contrast, you only need to know the ID of the note that you want to get rid of when calling `delete()`.

Next, create two operations in `swagger.yml` that refer to `notes.update` and `notes.delete`:

YAML

```yaml
# swagger.yml

# ...

paths:
  /people:
    # ...
  /people/{lname}:
    # ...
  /notes/{note_id}:
    get:
      # ...
    put:
      tags:
        - Notes
      operationId: "notes.update"
      summary: "Update a note"
      parameters:
        - $ref: "#/components/parameters/note_id"
      responses:
        "200":
          description: "Successfully updated note"
      requestBody:
        content:
          application/json:
            schema:
              x-body-name: "note"
              type: "object"
              properties:
                content:
                  type: "string"
    delete:
      tags:
        - Notes
      operationId: "notes.delete"
      summary: "Delete a note"
      parameters:
        - $ref: "#/components/parameters/note_id"
      responses:
        "204":
          description: "Successfully deleted note"
```

Again, the structure of `put` and `delete` are similar. The main difference is that you need to provide a `requestBody` that contains the note data to update the database object.

You've now created the endpoints to work with existing notes. Next, you'll add the endpoint to create a note.

Create a Note for a Person

So far, you can read, update, and delete a single note. These are actions that you can perform on existing notes. Now it's time to add the functionality to your REST API to also create a new note.

Add `create()` to `notes.py`:

Python

```python
# notes.py

from flask import make_response, abort

from config import db
from models import Note, Person, note_schema

# ...
```

```
def create(note):
    person_id = note.get("person_id")
    person = Person.query.get(person_id)

    if person:
        new_note = note_schema.load(note, session=db.session)
        person.notes.append(new_note)
        db.session.commit()
        return note_schema.dump(new_note), 201
    else:
        abort(
            404,
            f"Person not found for ID: {person_id}"
        )
```

A note always needs a person to belong to. That’s why you need to work with the `Person` model when you create a new note.

First, you look for the owner of the note by using `person_id`, which you provide with the `notes` argument for `create()`. If this person exists in the database, then you go ahead to append the new note to `person.notes`.

Although you’re working with the `person` database table in this case, SQLAlchemy will take care that the note is added to the `note` table.

To access `notes.create` with your API, hop over to `swagger.yml` and add another endpoint:

YAML

```yaml
# swagger.yml

# ...

paths:
  /people:
    # ...
  /people/{lname}:
    # ...
  /notes:
    post:
      operationId: "notes.create"
      tags:
          - Notes
        summary: "Create a note associated with a person"
        requestBody:
            description: "Note to create"
            required: True
            content:
              application/json:
                schema:
                  x-body-name: "note"
                  type: "object"
                  properties:
                    person_id:
                      type: "integer"
                    content:
                      type: "string"
        responses:
          "201":
            description: "Successfully created a note"
    /notes/{note_id}:
      # ...
```

You add the `/notes` endpoint right before the `/notes/{noted_id}` endpoint. That way, you order your notes endpoints from general to specific. This order helps you to navigate your `swagger.yml` file when your API grows larger.

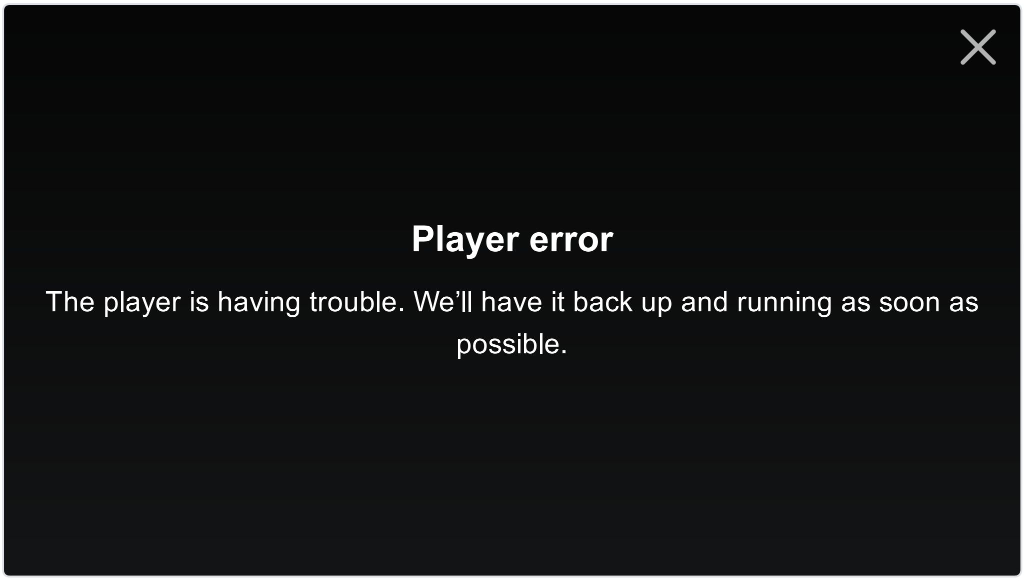
With the data in the `schema` block, you provide Marshmallow the information on how to serialize a note in your API. If you compare this `Note` schema to the `Note` model in `models.py`, then you'll notice that the names `person_id` and `content` match. The same goes for the fields' types.

You may also notice that not all the note model fields are present in the component schema. That's okay, because you'll only use this schema to post new notes. For each note, `id` and `timestamp` will be set automatically.

With all the endpoints to handle your notes in place, it's time to have a look at your API documentation.

Explore Your API Documentation

With the above changes in place, you can leverage your API to add, update, and remove notes. Visit your Swagger UI at `http://localhost:8000/api/ui` and explore your API endpoints:


Player error

The player is having trouble. We'll have it back up and running as soon as possible.

Awesome, your Flask REST API endpoints work! Any changes that you perform with your API appear on your front end, too.

Conclusion

In this tutorial, you adjusted your SQLite database to implement relationships. After that, you translated the `PEOPLE_NOTES` dictionary into data that conforms with your database structure, and you turned your Flask REST API into a note-keeping web application.

In the third part of this tutorial series, you learned how to:

- Work with **multiple tables** in a database
- Create **one-to-many** fields in your database
- Manage **relationships** with SQLAlchemy
- Leverage **nested schemas** with Marshmallow
- Display **related objects** in the front end

Knowing how to build and use database relationships gives you a powerful tool to solve many difficult problems. There are other relationship besides the one-to-many example from this tutorial. Other common ones are one-to-one, many-to-many, and many-to-one. All of them have a place in your tool belt, and SQLAlchemy can help you tackle them all!

You've successfully built a REST API to keep track of notes for people who may visit you throughout the year. Your database contains people like the Tooth Fairy, the Easter Bunny, and Knecht Ruprecht. By adding notes, you can keep track of your good deeds and hopefully receive valuable gifts from them.

To review your code, click the link below:

Source Code: Click here to download the free source code that you'll use to finish building a REST API with the Flask web framework.

Did you add a special person or note to your Flask REST API project? Let the Real Python community know in the comments below.

« Part 2: Database Persistence
Part 3: Database Relationships

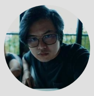
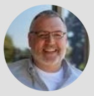
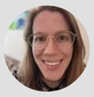
Mark as Completed

About **Philipp Acsany**

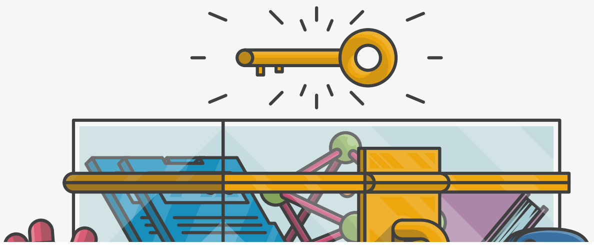
Philipp is a Berlin-based software engineer with a graphic design background and a passion for full-stack web development.

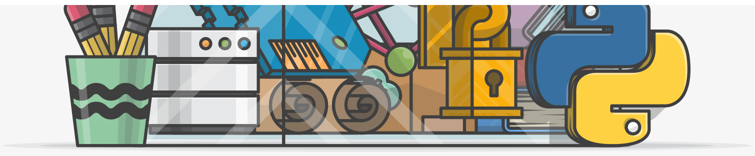
» More about Philipp

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren
Dan
 Doug
Geir Arne
Joanna
 Kate
Martin

Master Real-World Python Skills With Unlimited Access to Real Python





What Do You Think?

Rate this article:

Tweet Share in Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

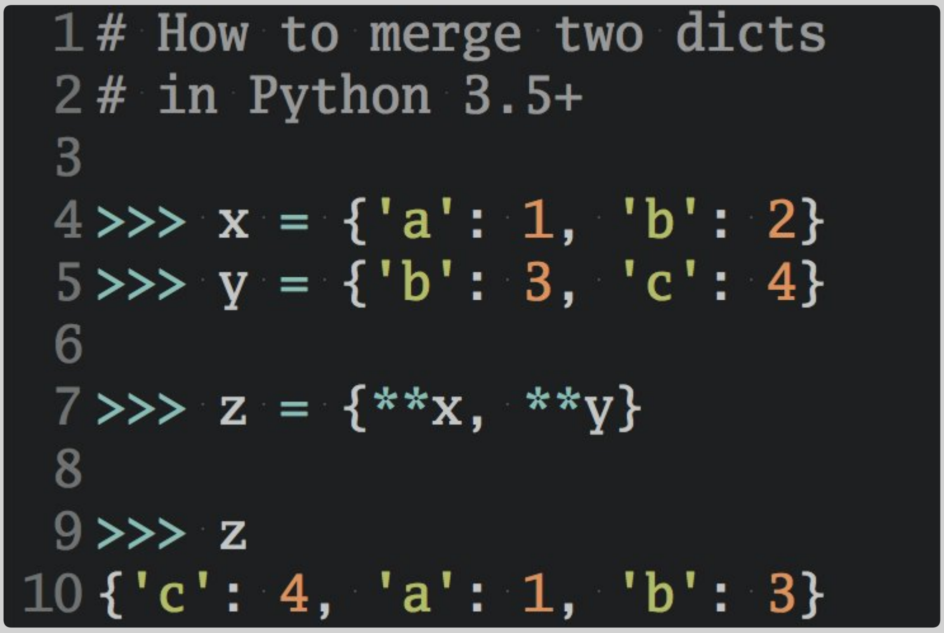
Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: api databases flask intermediate web-dev

— FREE Email Series —

Python Tricks



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

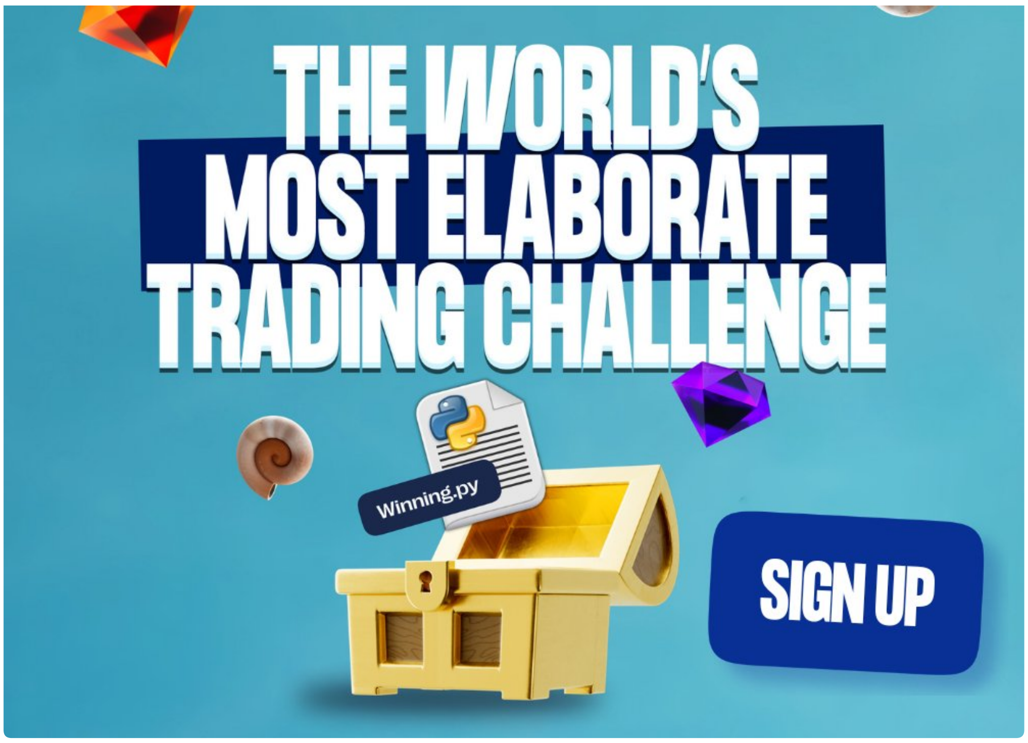


Table of Contents

Mark as Completed

Tweet Share Email