Getting started with CHROMA

Chroma is a database for building AI applications with embeddings. It comes with everything you need to get started built in, and runs on your machine. A hosted version is coming soon!

1. Install

```
pip install chromadb
```

* chromadb currently does not support Python 3.11 because of pytorch

2. Get the Chroma Client

```
import chromadb
chroma_client = chromadb.Client()
```

3. Create a collection

Collections are where you'll store your embeddings, documents, and any additional metadata. You can create a collection with a name:

```
collection = chroma_client.create_collection(name="my_collection")
```

4. Add some text documents to the collection

Chroma will store your text, and handle tokenization, embedding, and indexing automatically.

```
collection.add(
    documents=["This is a document", "This is another document"],
    metadatas=[{"source": "my_source"}, {"source": "my_source"}],
    ids=["id1", "id2"]
)
```

If you have already generated embeddings yourself, you can load them directly in:

```
collection.add(
    embeddings=[[1.2, 2.3, 4.5], [6.7, 8.2, 9.2]],
    documents=["This is a document", "This is another document"],
    metadatas=[{"source": "my_source"}, {"source": "my_source"}],
    ids=["id1", "id2"]
)
```

5. Query the collection

You can query the collection with a list of query texts, and Chroma will return the n most similar results. It's that easy!

```
results = collection.query(
    query_texts=["This is a query document"],
    n_results=2
)
```

By default data stored in Chroma is ephemeral making it easy to prototype scripts. It's easy to make Chroma persistent so you can reuse every collection you create and add more documents to it later. It will load your data automatically when you start the client, and save it automatically when you close it. Check out the Usage Guide for more info.

Find chromadb on PyPI.

🗄 Next steps

Chroma is designed to be simple enough to get started with quickly and flexible enough to meet many use-cases. You can use your own embedding models, query Chroma with your own embeddings, and filter on metadata. To learn more about Chroma, check out the Usage Guide and API Reference.
Chroma is integrated in LangChain (python and js), making it easy to build AI applications with Chroma. Check out the integrations page to learn more.
You can deploy a persistent instance of Chroma to an external server, to make it easier to work on larger projects or with a team.
Coming Soon
A hosted version of Chroma, with an easy to use web UI and API
Multiple datatypes, including images, audio, video, and more

Initiating the Chroma client

```
import chromadb
```

By default Chroma uses an in-memory database, which gets persisted on exit and loaded on start (if it exists). This is fine for many experimental / prototyping workloads, limited by your machine's memory.

```
from chromadb.config import Settings
client = chromadb.Client(Settings(
    chroma_db_impl="duckdb+parquet",
    persist_directory="/path/to/persist/directory" # Optional, defaults to .chromadb/ in the current directory
))
```

The persist_directory is where Chroma will store its database files on disk, and load them on start.

JUPYTER NOTEBOOKS
In a normal python program, .persist() will happening automatically if you set it. But in a Jupyter Notebook you will need to manually call client.persist().

The client object has a few useful convenience methods.

```
client.heartbeat() # returns a nanosecond heartbeat. Useful for making sure the client remains connected.
client.reset() # Empties and completely resets the database. ⚠ This is destructive and not reversible.
```

Running Chroma in client/server mode
Chroma can also be configured to use an on-disk database, useful for larger data which doesn't fit in memory. To run Chroma in client server mode, run the docker container:

```
docker-compose up -d --build
```

Then update your chroma client to point at the docker container. Default: localhost:8000

```
import chromadb
from chromadb.config import Settings
chroma_client = chromadb.Client(Settings(chroma_api_impl="rest",
                                        chroma_server_host="localhost",
                                        chroma_server_http_port="8000"
                                    ))
```

That's it! Chroma's API will run in client-server mode with just this change.

Using collections
Chroma lets you manage collections of embeddings, using the collection primitive.

Creating, inspecting, and deleting Collections
Chroma uses collection names in the url, so there are a few restrictions on naming them:

The length of the name must be between 3 and 63 characters.
The name must start and end with a lowercase letter or a digit, and it can contain dots, dashes, and underscores in between.
The name must not contain two consecutive dots.
The name must not be a valid IP address.
Chroma collections are created with a name and an optional embedding function. If you supply an embedding function, you must supply it every time you get the collection.

```
collection = client.create_collection(name="my_collection", embedding_function=emb_fn)
collection = client.get_collection(name="my_collection", embedding_function=emb_fn)
```

CAUTION
If you later wish to get_collection, you MUST do so with the embedding function you supplied while creating the collection

The embedding function takes text as input, and performs tokenization and embedding. If no embedding function is supplied, Chroma will use sentence transfomer as a default.

You can learn more about 🧬 embedding functions, and how to create your own.

Existing collections can be retrieved by name with .get_collection, and deleted with .delete_collection. You can also use .get_or_create_collection to get a collection if it exists, or create it if it doesn't.

```
collection = client.get_collection(name="test") # Get a collection object from an existing collection, by name. Will
raise an exception if it's not found.
collection = client.get_or_create_collection(name="test") # Get a collection object from an existing collection, by
name. If it doesn't exist, create it.
client.delete_collection(name="my_collection") # Delete a collection and all associated embeddings, documents, and
metadata. ⚠ This is destructive and not reversible
```

Collections have a few useful convenience methods.

```
collection.peek() # returns a list of the first 10 items in the collection
collection.count() # returns the number of items in the collection
collection.modify(name="new_name") # Rename the collection
```

Adding data to a Collection
Add data to Chroma with .add.

Raw documents:

```
collection.add(
    documents=["lorem ipsum...", "doc2", "doc3", ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "5"}, {"chapter": "29", "verse": "11"},
...],
    ids=["id1", "id2", "id3", ...]
)
```

If Chroma is passed a list of documents, it will automatically tokenize and embed them with the collection's embedding function (the default will be used if none was supplied at collection creation). Chroma will also store the documents themselves. If the documents are too large to embed using the chosen embedding function, an exception will be raised.

Each document must have a unique associated id. Chroma does not track uniqueness of ids for you, it is up to the caller to not add the same id twice. An optional list of metadata dictionaries can be supplied for each document, to store additional information and enable filtering.

Alternatively, you can supply a list of document-associated embeddings directly, and Chroma will store the associated documents without embedding them itself.

```
await collection.add(
    documents=["doc1", "doc2", "doc3", ...],
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "5"}, {"chapter": "29", "verse": "11"},
...],
    ids=["id1", "id2", "id3", ...]
)
```

If the supplied embeddings are not the same dimension as the collection, an exception will be raised.

You can also store documents elsewhere, and just supply a list of embeddings and metadata to Chroma. You can use the ids to associate the embeddings with your documents stored elsewhere.

```
collection.add(
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "5"}, {"chapter": "29", "verse": "11"},
...],
    ids=["id1", "id2", "id3", ...]
)
```

Querying a Collection
Chroma collections can be queried in a variety of ways, using the .query method.

You can query by a set of query_embeddings.

```
collection.query(
    query_embeddings=[[11.1, 12.1, 13.1],[1.1, 2.3, 3.2] ...]
    n_results=10,
```

```
    where={"metadata_field": "is_equal_to_this"},
    where_document={"$contains":"search_string"}
)
```

The query will return the n_results closest matches to each query_embedding, in order. An optional where filter dictionary can be supplied to filter the results by the metadata associated with each document. Additionally, an optional where_document filter dictionary can be supplied to filter the results by contents of the document.

If the supplied query_embeddings are not the same dimension as the collection, an exception will be raised.

You can also query by a set of query_texts. Chroma will first embed each query_text with the collection's embedding function, and then perform the query with the generated embedding.

```
collection.query(
    query_texts=["doc10", "thus spake zarathustra", ...]
    n_results=10,
    where={"metadata_field": "is_equal_to_this"},
    where_document={"$contains":"search_string"}
)
```

You can also retrieve items from a collection by id using .get.

```
collection.get(
    ids=["id1", "id2", "id3", ...],
    where={"style": "style1"}
)
```

.get also supports the where and where_document filters. If no ids are supplied, it will return all items in the collection that match the where and where_document filters.

Choosing which data is returned
When using get or query you can use the include parameter to specify which data you want returned - any of embeddings, documents, metadatas, and for query, distances. By default, Chroma will return the documents, metadatas and in the case of query, the distances of the results. embeddings are excluded by default for performance and the ids are always returned. You can specify which of these you want returned by passing an array of included field names to the includes parameter of the query or get method.

```
# Only get documents and ids
collection.get(
    include=["documents"]
)

collection.query(
    query_embeddings=[[11.1, 12.1, 13.1],[1.1, 2.3, 3.2] ...],
    include=["documents"]
)
```

Using Where filters
Chroma supports filtering queries by metadata and document contents. The where filter is used to filter by metadata, and the where_document filter is used to filter by document contents.

Filtering by metadata
In order to filter on metadata, you must supply a where filter dictionary to the query. The dictionary must have the following structure:

```
{
    "metadata_field": {
        <Operator>: <Value>
    },
    "metadata_field": {
        <Operator>: <Value>
    },
}
```

Filtering metadata supports the following operators:

$eq - equal to (string, int, float)
$ne - not equal to (string, int, float)

$gt - greater than (int, float)
$gte - greater than or equal to (int, float)
$lt - less than (int, float)
$lte - less than or equal to (int, float)
Using the $eq operator is equivalent to using the where filter.

```
{
    "metadata_field": "search_string"
}

# is equivalent to

{
    "metadata_field": {
        "$eq": "search_string"
    }
}
```

Filtering by document contents

In order to filter on document contents, you must supply a where_document filter dictionary to the query. The dictionary must have the following structure:

```
# Filtering for a search_string
{
    "$contains": "search_string"
}
```

Using logical operators

You can also use the logical operators $and and $or to combine multiple filters.

An $and operator will return results that match all of the filters in the list.

```
{
    "$and": [
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        },
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        }
    ]
}
```

An $or operator will return results that match any of the filters in the list.

```
{
    "$or": [
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        },
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        }
    ]
}
```

Updating data in a collection

Any property of items in a collection can be updated using .update.

```
collection.update(
```

```
    ids=["id1", "id2", "id3", ...],
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "5"}, {"chapter": "29", "verse": "11"}, ...],
    documents=["doc1", "doc2", "doc3", ...],
)
```

If an id is not found in the collection, an exception will be raised. If documents are supplied without corresponding embeddings, the embeddings will be recomupted with the collection's embedding function.

If the supplied embeddings are not the same dimension as the collection, an exception will be raised.

Deleting data from a collection

Chroma supports deleting items from a collection by id using .delete. The embeddings, documents, and metadata associated with each item will be deleted. ⚠ Naturally, this is a destructive operation, and cannot be undone.

```
collection.delete(
    ids=["id1", "id2", "id3",...],
    where={"chapter": "20"}
)
```

.delete also supports the where filter. If no ids are supplied, it will delete all items in the collection that match the where filter.

Embeddings are the A.I-native way to represent any kind of data, making them the perfect fit for working with all kinds of A.I-powered tools and algorithms. They can represent text, images, and soon audio and video. There are many options for creating embeddings, whether locally using an installed library, or by calling an API.

Chroma provides lightweight wrappers around popular embedding providers, making it easy to use them in your apps. You can set an embedding function when you create a Chroma collection, which will be used automatically, or you can call them directly yourself.

To get Chroma's embedding functions, import the chromadb.utils.embedding_functions module.

```
from chromadb.utils import embedding_functions
```

Default: Sentence Transformers

By default, Chroma uses Sentence Transformers to create embeddings. Sentence Transformers is a library for creating sentence and document embeddings that can be used for a wide variety of tasks. It is based on the Transformers library from Hugging Face. This embedding function runs locally on your machine, and may require you download the model files (this will happen automatically).

```
sentence_transformer_ef = embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-v2")
```

You can pass in an optional model_name argument, which lets you choose which Sentence Transformers model to use. By default, Chroma uses all-MiniLM-L6-v2. You can see a list of all available models here.

OpenAI

Chroma provides a convenient wrapper around OpenAI's embedding API. This embedding function runs remotely on OpenAI's servers, and requires an API key. You can get an API key by signing up for an account at OpenAI.

This embedding function relies on the openai python package, which you can install with pip install openai.

```
openai_ef = embedding_functions.OpenAIEmbeddingFunction(
                api_key="YOUR_API_KEY",
                model_name="text-embedding-ada-002"
            )
```

You can pass in an optional model_name argument, which lets you choose which OpenAI embeddings model to use. By default, Chroma uses text-embedding-ada-002. You can see a list of all available models here.

Cohere

Chroma also provides a convenient wrapper around Cohere's embedding API. This embedding function runs remotely on Cohere’s servers, and requires an API key. You can get an API key by signing up for an account at Cohere.

This embedding function relies on the cohere python package, which you can install with pip install cohere.

```
cohere_ef  = embedding_functions.CohereEmbeddingFunction(api_key="YOUR_API_KEY",  model_name="large")
cohere_ef(texts=["document1","document2"])
```

You can pass in an optional model_name argument, which lets you choose which Cohere embeddings model to use. By default, Chroma uses large model. You can see the available models under Get embeddings section here.

Multilingual model example

```
cohere_ef  = embedding_functions.CohereEmbeddingFunction(
        api_key="YOUR_API_KEY",
        model_name="multilingual-22-12")

multilingual_texts  = [ 'Hello from Cohere!', 'مرحبًا من كوهير!',
        'Hallo von Cohere!', 'Bonjour de Cohere!',
        '¡Hola desde Cohere!', 'Olá do Cohere!',
        'Ciao da Cohere!', '您好，来自 Cohere！',
    'कोहेरे से नमस्ते!' ]

cohere_ef(texts=multilingual_texts)
```

For more information on multilingual model you can read here.

Instructor models

The instructor-embeddings library is another option, especially when running on a machine with a cuda-capable GPU. They are a good local alternative to OpenAI (see the Massive Text Embedding Benchmark rankings). The embedding function requires the InstructorEmbedding package. To install it, run pip install InstructorEmbedding.

There are three models available. The default is hkunlp/instructor-base, and for better performance you can use hkunlp/instructor-large or hkunlp/instructor-xl. You can also specify whether to use cpu (default) or cuda. For example:

```
#uses base model and cpu
ef = embedding_functions.InstructorEmbeddingFunction()
```

or

```
ef = embedding_functions.InstructorEmbeddingFunction(
model_name="hkunlp/instructor-xl", device="cuda")
```

Keep in mind that the large and xl models are 1.5GB and 5GB respectively, and are best suited to running on a GPU.

Custom Embedding Functions

You can create your own embedding function to use with Chroma, it just needs to implement the EmbeddingFunction protocol.

```
from chromadb.api.types import Documents, EmbeddingFunction, Embeddings

class MyEmbeddingFunction(EmbeddingFunction):
    def __call__(self, texts: Documents) -> Embeddings:
        # embed the documents somehow
        return embeddings
```

We welcome contributions! If you create an embedding function that you think would be useful to others, please consider submitting a pull request to add it to Chroma's embedding_functions module.

We welcome pull requests to add new Embedding Functions to the community.

In-memory chroma

```
import chromadb
client = chromadb.Client()
```

In-memory chroma with saving/loading to disk

In this mode, Chroma will persist data between sessions. On load - it will load up the data in the directory you specify. And on exit - it will save to that directory.

```
import chromadb
from chromadb.config import Settings
client = chromadb.Client(Settings(chroma_db_impl="duckdb+parquet",
```

```
                                    persist_directory="/path/to/persist/directory"
                                ))
```

Run chroma just as a client to talk to a backend service

For production use cases, an in-memory database will not cut it. Run docker-compose up -d --build to run a production backend in Docker on your local computer. Simply update your API initialization and then use the API the same way as before.

```
import chromadb
from chromadb.config import Settings
chroma_client = chroma.Client(Settings(chroma_api_impl="rest",
                                        chroma_server_host="localhost",
                                        chroma_server_http_port="8000"
                                    ))
```

Methods on Client

Methods related to Collections

COLLECTION NAMING

Collections are similar to AWS s3 buckets in their naming requirements because they are used in URLs in the REST API. Here's the full list.

```
# list all collections
client.list_collections()

# make a new collection
collection = client.create_collection("testname")

# get an existing collection
collection = client.get_collection("testname")

# get a collection or create if it doesn't exist already
collection = client.get_or_create_collection("testname")

# delete a collection
client.delete_collection("testname")
```

Utility methods

```
# resets entire database - this *cant* be undone!
client.reset()

# returns timestamp to check if service is up
client.heartbeat()
```

Methods on Collection

```
# change the name or metadata on a collection
collection.modify(name="testname2")

# get the number of items in a collection
collection.count()

# add new items to a collection
# either one at a time
collection.add(
    embeddings=[1.5, 2.9, 3.4],
    metadatas={"uri": "img9.png", "style": "style1"},
    documents="doc1000101",
    ids="uri9",
)
# or many, up to 100k+!
collection.add(
    embeddings=[[1.5, 2.9, 3.4], [9.8, 2.3, 2.9]],
    metadatas=[{"style": "style1"}, {"style": "style2"}],
    ids=["uri9", "uri10"],
)
collection.add(
    documents=["doc1000101", "doc288822"],
    metadatas=[{"style": "style1"}, {"style": "style2"}],
    ids=["uri9", "uri10"],
```

```
)

# update items in a collection
collection.update()

# get items from a collection
collection.get()

# convenience, get first 5 items from a collection
collection.peek()

# do nearest neighbor search to find similar embeddings or documents, supports filtering
collection.query(
    query_embeddings=[[1.1, 2.3, 3.2], [5.1, 4.3, 2.2]],
    n_results=2,
    where={"style": "style2"}
)

# delete items
collection.delete()

# advanced: manually create the embedding search index
collection.create_index()
```