Cryptocurrency Tracker With A Stateless Cache

Python Project Report

Submitted by

Prateek Jayaswal, I036
Vedant Maheshwari, I051

MUKESH PATEL SCHOOL OF TECHNOLOGY MANAGEMENT & ENGINEERING

Mukesh Patel School of Technology Management & Engineering
NMIMS (Deemed –to-be University)
JVPD Scheme Bhaktivedanta Swami Marg,
Ville Parle (W), Mumbai-400 056.

April, 2023

Project Introduction

Cryptocurrency is a digital or virtual currency that uses cryptography to secure and verify transactions and our Cryptocurrency tracker is a tool that helps in monitoring the exchange rates of different currencies in the market.

The objective of this project is to develop a cryptocurrency tracker with and underlying stateless cache based on an sqlite3 database.

An API provides us the exchange rates of INR, EUR, GBP, DOGE and LTC against USD to compare the target exchange rates for DOGE and LTC with the current exchange rates to determine whether it is profitable to buy DOGE or LTC in your home currency.

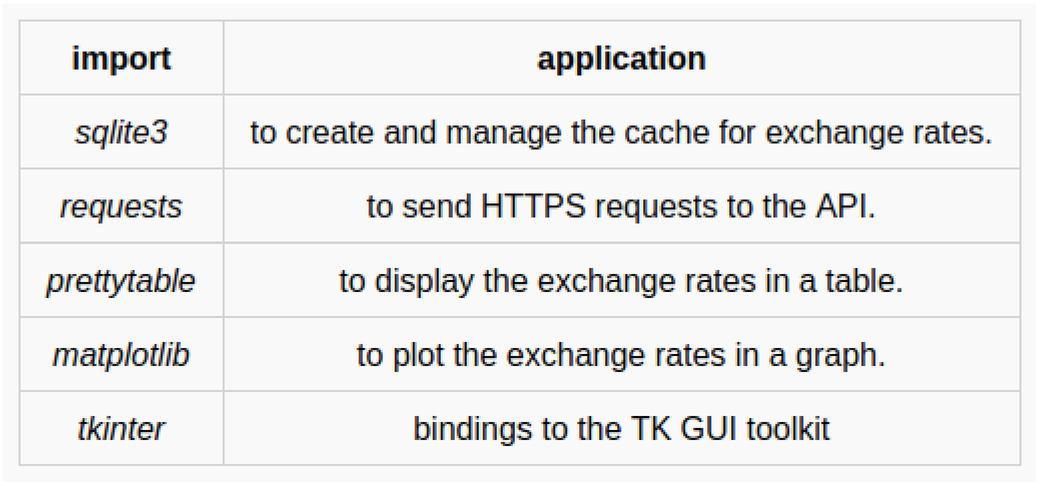
When the program fetches the currency rates from an external API, it caches them in a sqlite3 database for future use.

The database is used as a stateless cache for the rates, so that the program doesn't have to request it from the API again every time it runs.

In this program, the caching is stateless, which means that the server hosting the API doesn't store the cached rates, instead we store it locally in an sqlite3 database
This approach has a few advantages as it reduces the load on the API host by only making requests when necessary, it speeds up the program by avoiding unnecessary requests and database writes.

Tools & Technologies

The project is developed in Python programming language and the following Python libraries have been used in our project:



import	application
sqlite3	to create and manage the cache for exchange rates.
requests	to send HTTPS requests to the API.
prettytable	to display the exchange rates in a table.
matplotlib	to plot the exchange rates in a graph.
tkinter	bindings to the TK GUI toolkit

Scope of Project

A Cryptocurrency tracker using stateless cache has the following applications in various industries:

- **Finance**: The tracker can be used by financial analysts to monitor the exchange rates of different currencies and make informed decisions regarding investments.
- **E-commerce**: The tracker can be integrated with e-commerce websites to provide real-time exchange rates to customers and allow them to make purchases in different currencies.
- **Travel**: The tracker can be used by travelers to monitor the exchange rates of different currencies and make informed decisions regarding foreign currency exchange.
- **Education**: The tracker can be used as a learning tool to teach students about cryptocurrency and how it works. The tracker can also be used to teach students about exchange rates and how they fluctuate over time.

Implementation - Code

backend.py

```python
import datetime as dt
import sqlite3
import requests
import prettytable
import matplotlib.pyplot as plt
import numpy as np


class backend:
    def __init__(self, homeCurrency: str, numOfDOGEToBuy:
float, moneyToBuyDOGE: float, numOfLTCToBuy: float,
moneyToBuyLTC: float) -> None:
        self.homeCurrency: str = homeCurrency
        self.numOfDOGEToBuy: float = numOfDOGEToBuy
        self.moneyToBuyDOGE: float = moneyToBuyDOGE
        self.numOfLTCToBuy: float = numOfLTCToBuy
        self.moneyToBuyLTC: float = moneyToBuyLTC
        print(f"constructed backend object with
{homeCurrency}, {numOfDOGEToBuy}, {moneyToBuyDOGE},
{numOfLTCToBuy}, {moneyToBuyLTC}\n")

    def fetchRates(self, date: str = "latest") -> dict[str,
str | float]:
        url: str = f"https://api.exchangerate.host/{date}"

        response: requests.Response = requests.get(url,
params={"base": "USD", "symbols": "INR,EUR,GBP", "places":
4}, timeout=10)
        data: dict = response.json()
        rates: dict[str, float] = data["rates"]
        entry: dict[str, str | float] = {
            "time": data["date"],
            "INR": rates["INR"],
            "EUR": rates["EUR"],
            "GBP": rates["GBP"],
        }

        response = requests.get(url, params={"base": "USD",
"source": "crypto", "symbols": "DOGE,LTC"}, timeout=10)
        data = response.json()
        rates = data["rates"]
        entry["DOGE"] = rates["DOGE"]
        entry["LTC"] = rates["LTC"]

        print(f"\nGET: {url}: {entry}\n")
        return entry

    def connect2cache(self) -> tuple[sqlite3.Cursor,
sqlite3.Connection]:
        cachedRatesdb: sqlite3.Connection =
sqlite3.connect("cachedRates.db")
        cursor: sqlite3.Cursor = cachedRatesdb.cursor()
        cursor.execute(
            """CREATE TABLE IF NOT EXISTS cache (
                timestamp text,
                INR real,
                EUR real,
                GBP real,
                DOGE real,
                LTC real
            )"""
        )
        return cursor, cachedRatesdb

    def compareTarget(self) -> dict[str, bool]:
        # self.test(1)
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()
        today: str = str(dt.date.today() -
dt.timedelta(days=1))
```

```
        cursor.execute("SELECT timestamp FROM cache")
        timestamps: list[str] = [row[0] for row in
cursor.fetchall()]
        if today in timestamps:
            cursor.execute(f"SELECT * FROM cache WHERE
timestamp = '{today}'")
            row = cursor.fetchone()
            rates = {"time": row[0], "INR": row[1], "EUR":
row[2], "GBP": row[3], "DOGE": row[4], "LTC": row[5]}
            print(f"rates for {today} already in sqlite3
cache {rates}")
        else:
            rates: dict[str, str | float] =
self.fetchRates()
            cursor.execute(f"INSERT INTO cache VALUES
('{rates['time']}', {rates['INR']}, {rates['EUR']},
{rates['GBP']}, {rates['DOGE']}, {rates['LTC']})")
            print(f"cached {rates}")

        res: dict[str, bool] = {"DOGE": False, "LTC":
False}
        if self.homeCurrency != "USD":
            res["DOGE"] = self.moneyToBuyDOGE /
(rates[self.homeCurrency] * self.numOfDOGEToBuy) >=
rates["DOGE"]
            res["LTC"] = self.moneyToBuyLTC /
(rates[self.homeCurrency] * self.numOfLTCToBuy) >=
rates["LTC"]
        else:
            res["DOGE"] = self.moneyToBuyDOGE /
self.numOfDOGEToBuy >= rates["DOGE"]
            res["LTC"] = self.moneyToBuyLTC /
self.numOfLTCToBuy >= rates["LTC"]

        cachedRatesdb.close()
        return res

    def ratesInThePast(self) -> list[dict[str, str |
float]]:
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()
        today: dt.date = dt.date.today()
        ratesThisWeekAsListOfDicts: list[dict[str, str |
float]] = list(dict())

        cursor.execute("SELECT timestamp FROM cache")
        timestamps: list[str] = [row[0] for row in
cursor.fetchall()]
        for i in range(14, 0, -1):
            date = str(today - dt.timedelta(days=i))
            if date not in timestamps:

ratesThisWeekAsListOfDicts.append(self.fetchRates(date))
            else:
                print(f"rates for {date} already cached")

        cachedRatesdb.close()
        return ratesThisWeekAsListOfDicts

    def dbHandler(self) -> None:
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()
        pastRates: list[dict[str, str | float]] =
self.ratesInThePast()

        for dc in pastRates:
            cursor.execute(f"INSERT INTO cache VALUES
('{dc['time']}', {dc['INR']}, {dc['EUR']}, {dc['GBP']},
{dc['DOGE']}, {dc['LTC']})")
            print(f"cached {dc}")

        cachedRatesdb.commit()
        cachedRatesdb.close()
        self.printCACHE()

    def printCACHE(self) -> None:
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()

        cursor.execute("SELECT * FROM cache")
        table: prettytable.PrettyTable | None =
prettytable.from_db_cursor(cursor)
        print(table)

        cachedRatesdb.close()

    def plot(self, coin: str) -> None:
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()

        cursor.execute(f"SELECT timestamp, {coin} FROM
cache")
        result: list[tuple[str, float]] = cursor.fetchall()
        timestamps: np.ndarray[str, np.dtype[np.string_]] =
np.array([result[0] for result in result])
        coinRates: np.ndarray[float, np.dtype[np.float64]]
= np.array([result[1] for result in result])
        cachedRatesdb.close()

        plt.style.use("dark_background")
        plt.plot(timestamps, coinRates, color="#a01bf2")
        plt.title(f"Historical Exchange Rate Of {coin} in
USD")
        plt.xlabel("Timestamps (in days)")
        plt.ylabel(f"{coin}'s exchange rate (in USD)")
        figManager: plt.FigureManagerBase =
plt.get_current_fig_manager()
        figManager.window.state("normal")
        plt.show()

    def test(self, rowsTBDel: int) -> None:
        if rowsTBDel < 1:
            print("Bad Input: cache unchanged")
            return
        cursor: sqlite3.Cursor
        cachedRatesdb: sqlite3.Connection
        cursor, cachedRatesdb = self.connect2cache()

        cursor.execute("SELECT COUNT(*) FROM cache")
        num_rows: int = cursor.fetchone()[0]
        if num_rows >= rowsTBDel:
            cursor.execute(f"DELETE FROM cache WHERE ROWID
IN (SELECT ROWID FROM cache ORDER BY ROWID DESC LIMIT
{rowsTBDel})")
            cachedRatesdb.commit()
            print(f"Last {rowsTBDel} rows successfully
deleted")
            cachedRatesdb.commit()
            self.printCACHE()
        else:
            print(f"There are not enough rows in cache to
delete the last {rowsTBDel} rows")

        cachedRatesdb.close()


if __name__ == "__main__":
    instance = backend("INR", -1, -1, -1, -1)
    backend.dbHandler(self=instance)
    backend.compareTarget(self=instance)
```

- **constructor** for the backend class requires:
  - `homeCurrency: str` can be `"INR"` , `"USD"` , `"EUR"` , `"GBP"`
  - `numOfDOGEToBuy: float` is the number of DOGE coins you aim to buy for `moneyToBuyDOGE: float` money in your home currency
  - `numOfLTCToBuy: float` is the number of DOGE coins you aim to buy for `moneyToBuyLTC: float` money in your home currency

- `fetchRates(self, date: str = "latest") → dict[str, str | float]`
    - returns the latest rates when called without any arguments otherwise can return rates for a specific date provided as a string in the format `"YYYY-MM-DD"`
        - from API hosted on the domain `https://api.exchangerate.host/<date>"`
    - sends 2 GET requests
        - one asks for the rates of Fiat Currencies INR, EUR, GBP against USD
        - another asks for rates of Crypto Currencies Dogecoin and Litecoin against USD
    - it extracts the rates from the request object and appends it to a dictionary called `entry` that with a timestamp
    - `entry` is returned
- `connect2cache(self) → tuple[sqlite3.Cursor, sqlite3.Connection]`
    - creates the cache as a sqlite3 database if it doesn't exist already
    - returns an opened connection to the cache and a cursor to the cache as objects
    - decoupled to conform to the DRY principle
- `compareTarget(self) → dict[str, bool]`
    - opens connection to the cache using `connect2cache(self) → tuple[sqlite3.Cursor, sqlite3.Connection]`
    - extracts the entire column for timestamps from the cache and checks for the timestamp for the present day
        - if the timestamp for the present day is not found, it calls `fetchRates(self, date: str = "latest") → dict[str, str | float]` and caches the rates
    - then it makes precise calculations using both, the latest exchange rates between Crypto Currencies against USD and

    the exchange rates between your home currency against USD
        - from these calculations it predicts if the current rate meets the requirement for buying a set number of tokens for a set amount of money passed during the creation of the object
        - it returns a dictionary like `{"DOGE": True/False, "LTC": True/False}` after closing the connection to the cache
- `ratesInThePast(self) → list[dict[str, str | float]]`
    - opens connection to the cache using `connect2cache(self) → tuple[sqlite3.Cursor, sqlite3.Connection]`
    - extracts the entire column for timestamps from the cache
    - loops over the last 14 days using variable `date`
        - it calls `fetchRates(self, date: str = "latest") → dict[str, str | float]` with `str = date` if `date not in timestamps`
    - it returns a list of rates that we fetched from `fetchRates()` (which is later cached by `dbHandler(self) → None`) after closing the connection to the cache
- `dbHandler(self) → None`
    - opens connection to the cache using `connect2cache(self) → tuple[sqlite3.Cursor, sqlite3.Connection]`
    - stores the result of `ratesInThePast(self) → list[dict[str, str | float]]` in variable `pastRates`
    - caches the rates in `pastRates`
    - commits and closes the connection to the cache
    - prints the entire cache in a tabular form using `printCACHE(self) → None`
- `plot(self, coin: str) → None` where coin can be either `"DOGE"` or `"LTC"`
    - opens connection to the cache using `connect2cache(self) → tuple[sqlite3.Cursor, sqlite3.Connection]`

        - queries the cache for entire column of timestamps and the rates for `coin` and stores them into numpy arrays called `timestamps` and `coinRates` respectively
        - close the connection to the cache
        - plots a line chart showing fluctuations of prices of `coin` over time with `timestamps` on X-Axis and `coinRates` on Y-Axis
    - `test(self, rowsTBDel: int) → None` where `rowsTBDel` is the number of rows to be deleted
        - deletes `rowsTBDel` number of rows from cache from the bottom after validating `rowsTBDel`
        - used for debugging and forcefully demonstrating caching mechanism by test_cache.py
            - if the last 2 rows are deleted, the rates for last 2 days are deleted
            - when `dbHandler(self) → None` is called it calls `ratesInThePast(self) → list[dict[str, str | float]]`
            - `ratesInThePast(self) → list[dict[str, str | float]]` will only fetch the rates for last 2 days as the rates before it are already cached

frontend.py

```
import tkinter
import tkinter.messagebox
import socket
import customtkinter
from PIL import Image
import backend

customtkinter.set_appearance_mode("Dark")
customtkinter.set_default_color_theme("blue")

fg1 = "#a01bf2"
hvr = "#3f1369"
```

```python
fg2 = "#6a16b7"


class App(customtkinter.CTk):
    def __init__(self):
        super().__init__()

        fnt = customtkinter.CTkFont(family="HGGothicE", size=15)

        self.backendObj = backend.backend("INR", -1, -1, -1, -1)
        self.backendObj.dbHandler()  # builds cache if missing as soon as the program starts

        self.title("CRYTO-SPHERE")

        self.grid_columnconfigure(1, weight=1)
        self.grid_columnconfigure((2, 3), weight=0)
        self.grid_rowconfigure((0, 1, 2), weight=1)

        # create sidebar frame with widgets
        self.sidebar_frame = customtkinter.CTkFrame(self, width=250, corner_radius=0)
        self.sidebar_frame.grid(row=0, column=0, rowspan=4, sticky="nsew")
        self.sidebar_frame.grid_rowconfigure(4, weight=1)

        my_image = customtkinter.CTkImage(light_image=Image.open("cryptologo.png"), dark_image=Image.open("cryptologo.png"), size=(200, 200))

        self.logo_label = customtkinter.CTkLabel(self.sidebar_frame, text="", font=customtkinter.CTkFont(size=20, weight="bold"), image=my_image)
        self.logo_label.grid(row=0, column=0, padx=20, pady=(20, 10))
        self.Currency_choice_label =
customtkinter.CTkLabel(self.sidebar_frame, text="Currency
", anchor="w", font=fnt)
        self.Currency_choice_label.grid(row=5, column=0,
padx=20, pady=(10, 0))

        self.Currency_choice_optionemenu =
customtkinter.CTkOptionMenu(self.sidebar_frame, values=
["INR(₹) ", "USD($)", "EUR(€)", "GBP(£)"], fg_color=fg1,
button_color=fg2, button_hover_color=hvr, width=180,
dropdown_hover_color=fg2)
        self.Currency_choice_optionemenu.grid(row=6,
column=0, padx=20, pady=(10, 10))

        self.Wifi_label =
customtkinter.CTkLabel(self.sidebar_frame, text="Wi-Fi ",
anchor="w", font=fnt)
        self.Wifi_label.grid(row=7, column=0, padx=20,
pady=(10, 0))

        self.wifi_connect_label =
customtkinter.CTkLabel(self.sidebar_frame, fg_color=fg1,
width=180, corner_radius=8, text="Connected")
        self.wifi_connect_label.grid(row=8, column=0,
padx=20, pady=(10, 20))

        self.main_frame = customtkinter.CTkFrame(self,
width=1000, height=600, fg_color="transparent",
border_color=fg2, border_width=4)
        self.main_frame.grid(row=0, rowspan=10, column=1,
padx=20, pady=(20, 0), sticky="nsew")

        # initializes and displays necessary widgets when
device is offline
        if self.isConnected():
            self.plot_frame =
customtkinter.CTkFrame(self.main_frame, width=960,
height=50, fg_color="transparent", corner_radius=8)
```

```
            self.plot_frame.grid(row=0, column=0, padx=20,
pady=(60, 20), sticky="ew")

            self.disp_label =
customtkinter.CTkLabel(self.plot_frame, fg_color=hvr,
font=fnt, text="Enter target price and tokens in front of
respective crypto currency and click check", width=650,
corner_radius=8)
            self.disp_label.pack(padx=(10, 5), pady=(10,
50), side="left")

            self.ref_button =
customtkinter.CTkButton(self.plot_frame, fg_color=fg1,
hover_color=hvr, font=fnt, text="CHECK", width=80,
command=self.cmpr)
            self.ref_button.pack(padx=(5, 10), pady=(10,
50), side="left")

            self.plot_button =
customtkinter.CTkButton(self.plot_frame, fg_color=fg1,
hover_color=hvr, font=fnt, text="PLOT", width=180,
command=self.plt)
            self.plot_button.pack(padx=10, pady=(10, 50),
side="left")

            self.crypt1_frame =
customtkinter.CTkFrame(self.main_frame, width=960,
height=200, fg_color=fg2, corner_radius=8)
            self.crypt1_frame.grid(row=1, column=0,
padx=20, pady=10)

            self.crypt1_cb =
customtkinter.CTkCheckBox(self.crypt1_frame,
corner_radius=8, fg_color=hvr, border_color=hvr,
hover_color=hvr, border_width=2, width=180, height=180,
font=fnt, text="DOGE")
            self.crypt1_cb.pack(padx=30, pady=10,
side="left")
```

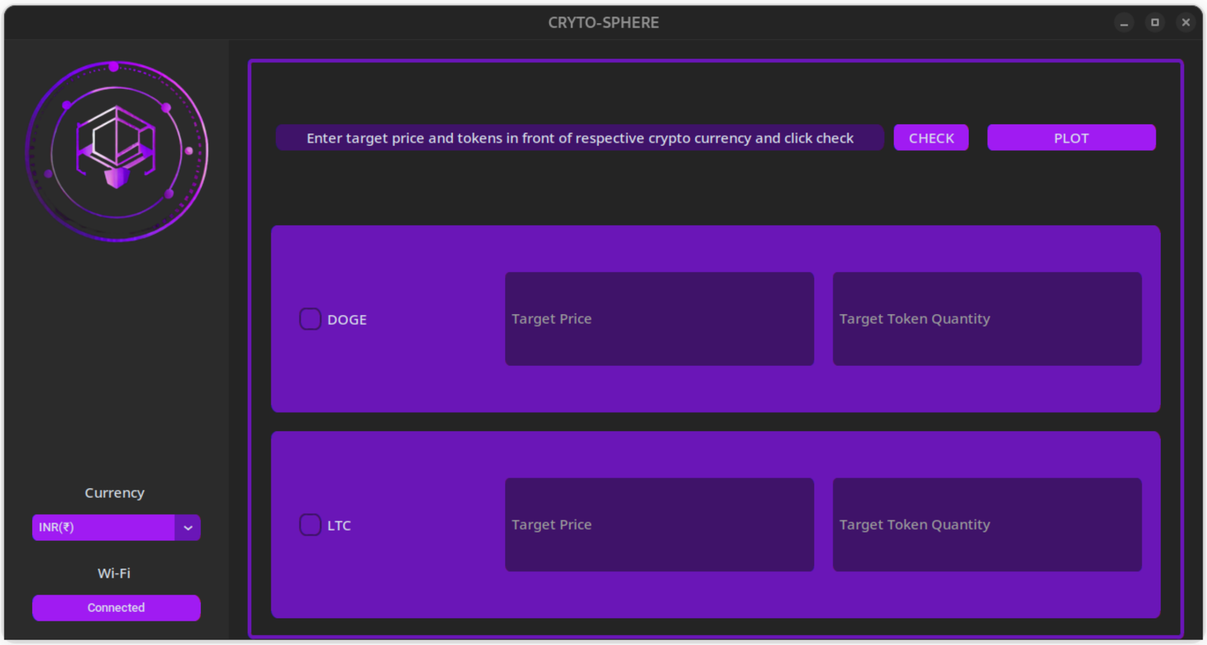
```
            self.crypt1_ent1 =
customtkinter.CTkEntry(self.crypt1_frame, fg_color=hvr,
border_color=hvr, border_width=2, width=330, height=100,
font=fnt, placeholder_text="Target Price")
            self.crypt1_ent1.pack(side="left", padx=10,
pady=50)
            self.crypt1_ent1.bind("<FocusOut>", lambda
event: self.validate_float(self.crypt1_ent1))

            self.crypt1_ent2 =
customtkinter.CTkEntry(self.crypt1_frame, fg_color=hvr,
border_color=hvr, border_width=2, width=330, height=100,
font=fnt, placeholder_text="Target Token Quantity")
            self.crypt1_ent2.pack(side="left", padx=(10,
20), pady=50)
            self.crypt1_ent2.bind("<FocusOut>", lambda
event: self.validate_float(self.crypt1_ent2))

            self.crypt2_frame =
customtkinter.CTkFrame(self.main_frame, width=960,
height=200, fg_color=fg2, corner_radius=8)
            self.crypt2_frame.grid(row=2, column=0,
padx=20, pady=10)

            self.crypt2_cb =
customtkinter.CTkCheckBox(self.crypt2_frame,
corner_radius=8, fg_color=hvr, border_color=hvr,
hover_color=hvr, border_width=2, width=180, height=180,
font=fnt, text="LTC")
            self.crypt2_cb.pack(padx=30, pady=10,
side="left")

            self.crypt2_ent1 =
customtkinter.CTkEntry(self.crypt2_frame, fg_color=hvr,
border_color=hvr, border_width=2, width=330, height=100,
font=fnt, placeholder_text="Target Price")
            self.crypt2_ent1.pack(side="left", padx=10,
pady=50)
            self.crypt2_ent1.bind("<FocusOut>", lambda
event: self.validate_float(self.crypt2_ent1))

            self.crypt2_ent2 =
customtkinter.CTkEntry(self.crypt2_frame, fg_color=hvr,
border_color=hvr, border_width=2, width=330, height=100,
font=fnt, placeholder_text="Target Token Quantity")
            self.crypt2_ent2.pack(side="left", padx=(10,
20), pady=50)
            self.crypt2_ent2.bind("<FocusOut>", lambda
event: self.validate_float(self.crypt2_ent2))

        # initializes and displays necessary widgets when
device is offline
        else:
            self.off_label =
customtkinter.CTkLabel(self.main_frame, text="Please
connect to Wi-Fi and restart, press 'OK' to close the
Application", font=customtkinter.CTkFont(size=20,
weight="bold"))
            self.off_label.place(relx=0.5, rely=0.5,
anchor=tkinter.CENTER)
            self.re_button =
customtkinter.CTkButton(self.main_frame, fg_color=fg1,
hover_color=hvr, width=180, corner_radius=8, text="OK",
command=self.restart_program)
            self.re_button.place(relx=0.5, rely=0.6,
anchor=tkinter.CENTER)

    # displays error message on a pop-up message box
    def error_box(self, txt):
        window = customtkinter.CTkToplevel(self)
        window.geometry("400x200+800+200")
        customtkinter.CTkLabel(window, font=("HGGothicE",
15), width=380, height=180, text=txt).pack(padx=10,
pady=10)
        return 0

    # checks if wifi is connected
    def isConnected(self):
        try:
            s =
socket.create_connection(("www.geeksforgeeks.org", 80))
            if s is not None:
                s.close()
                return True
        except OSError:
            self.wifi_connect_label.configure(text="NOT
CONNECTED", fg_color="#FF0000")

self.Currency_choice_optionemenu.configure(state="disabled"
)
            return False

    # destroys app window
    def restart_program(self):
        self.destroy()

    # makes sure entryboxes can only store 6 decimal float
values above zeros, rounds up or clears invalid values
    def validate_float(self, entry):
        try:
            value = float(entry.get())
            if value <= 0:
                entry.select_clear()
                entry.delete(0, "end")
                return False
            elif round(value, 6) == value:
                return True
            else:
                entry.select_clear()
                entry.delete(0, "end")
                entry.insert(0, round(value, 6))
                return False
        except ValueError:
            entry.select_clear()
            entry.delete(0, "end")
            return False
    # displays comparison values
    def cmpr_disp(self, cmprd):
        c1 = self.crypt1_cb.get()
        c2 = self.crypt2_cb.get()
        if c1 and c2:
            txt = "Target reached: DOGE: " +
str(cmprd["DOGE"]) + " LTC: " + str(cmprd["LTC"])
        elif c1:
            txt = "Target reached: DOGE: " +
str(cmprd["DOGE"])
        elif c2:
            txt = "Target reached: LTC: " +
str(cmprd["LTC"])
        else:
            pass
        self.disp_label.configure(text=txt)

    # compares values entered in entry boxes with actual
crypto rates, raises errors wherever necessary
    def cmpr(self):
        c1 = self.crypt1_cb.get()
        c2 = self.crypt2_cb.get()
        crypt1_box1 = 0
        crypt1_box2 = 0
        crypt2_box1 = 0
        crypt2_box2 = 0
        if c1 and c2:
            if len(self.crypt1_ent1.get()) > 0:
                crypt1_box1 = float(self.crypt1_ent1.get())
                if len(self.crypt1_ent2.get()) > 0:
                    crypt1_box2 =
float(self.crypt1_ent2.get())
                    if len(self.crypt2_ent1.get()) > 0:
                        crypt2_box1 =
float(self.crypt2_ent1.get())
                        if len(self.crypt2_ent2.get()) > 0:
                            crypt2_box2 =
float(self.crypt2_ent2.get())
                        else:
                            self.error_box("PLEASE FILL
NECESSARY FIELDS")
                    else:
                        self.error_box("PLEASE FILL
NECESSARY FIELDS")
                else:
                    self.error_box("PLEASE FILL NECESSARY
FIELDS")
            else:
                self.error_box("PLEASE FILL NECESSARY
FIELDS")
        elif c1:
            if len(self.crypt1_ent1.get()) > 0:
                crypt1_box1 = float(self.crypt1_ent1.get())
                if len(self.crypt1_ent2.get()) > 0:
                    crypt1_box2 =
float(self.crypt1_ent2.get())
                else:
                    self.error_box("PLEASE FILL NECESSARY
FIELDS")
            else:
                self.error_box("PLEASE FILL NECESSARY
FIELDS")
            crypt2_box1 = -1
            crypt2_box2 = -1
        elif c2:
            crypt1_box1 = -1
            crypt1_box2 = -1
            if len(self.crypt2_ent1.get()) > 0:
                crypt2_box1 = float(self.crypt2_ent1.get())
                if len(self.crypt2_ent2.get()) > 0:
                    crypt2_box2 =
float(self.crypt2_ent2.get())
                else:
                    self.error_box("PLEASE FILL NECESSARY
FIELDS")
            else:
                self.error_box("PLEASE FILL NECESSARY
FIELDS")
```

```
        else:
            self.error_box("PLEASE SELECT ATLEAST ONE
CRYPTO-CURRENCY")
        if crypt1_box1 and crypt1_box2 and crypt2_box1 and
crypt1_box2:
            actual_backendObj =
backend.backend(self.Currency_choice_optionemenu.get()
[0:3], crypt1_box2, crypt1_box1, crypt2_box2, crypt2_box1)

self.cmpr_disp(actual_backendObj.compareTarget())

    # checks cryptos selected and plots them, raises
error_box wherever necessary
    def plt(self):
        c1 = self.crypt1_cb.get()
        c2 = self.crypt2_cb.get()
        if c1 and c2:
            self.backendObj.plot("DOGE")
            self.backendObj.plot("LTC")
        elif c1:
            self.backendObj.plot("DOGE")
        elif c2:
            self.backendObj.plot("LTC")
        else:
            self.error_box("PLEASE SELECT ATLEAST ONE
CRYPTO-CURRENCY")


if __name__ == "__main__":
    app = App()
    app.state("normal")
    app.mainloop()
```

- `error_box(self, txt)` displays error message on a pop-up message box
- `isConnected(self)` checks if wifi is connected
- `restart_program(self)` destroys app window

- `validate_float(self, entry)` makes sure entryboxes can only store 6 decimal float values above zeros, rounds up or clears invalid values
- `cmpr_disp(self, cmprd)` displays comparison values
- `cmpr(self)` compares values entered in entry boxes with actual crypto rates, raises errors wherever necessary
- `plt(self)` checks cryptos selected and plots them, raises `error_box` wherever necessary

Screenshots

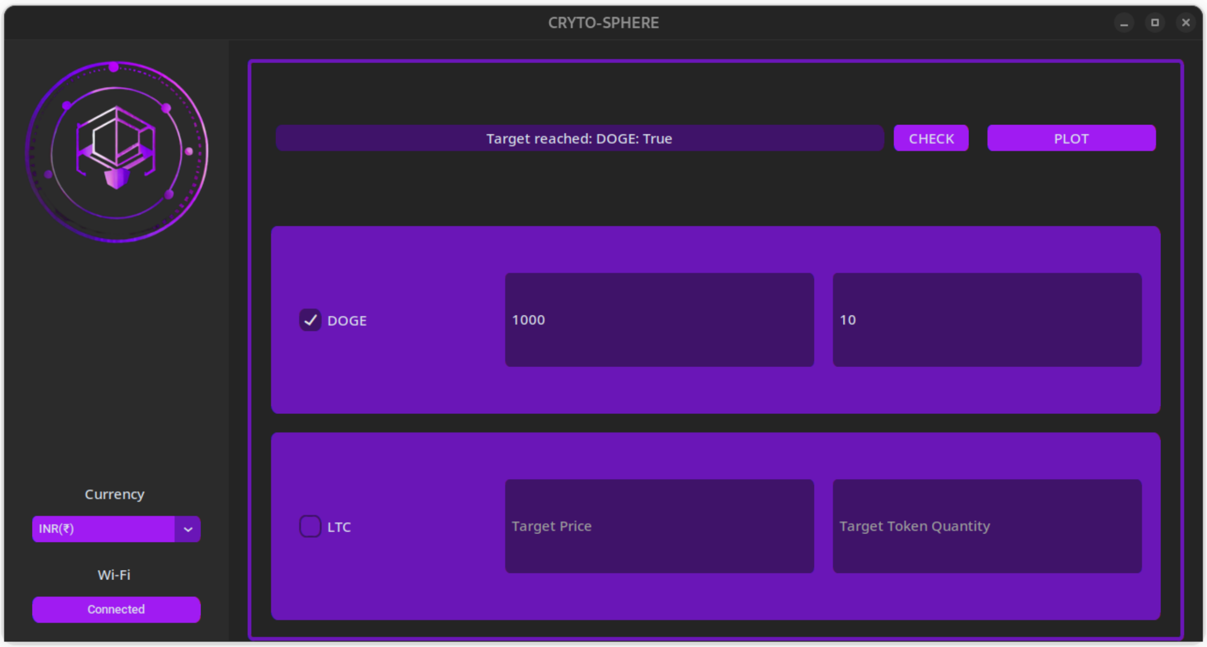


Enter target price and tokens in front of respective crypto currency and click check

☐ DOGE	Target Price	Target Token Quantity
☐ LTC	Target Price	Target Token Quantity

Currency: INR(₹)

Wi-Fi: Connected

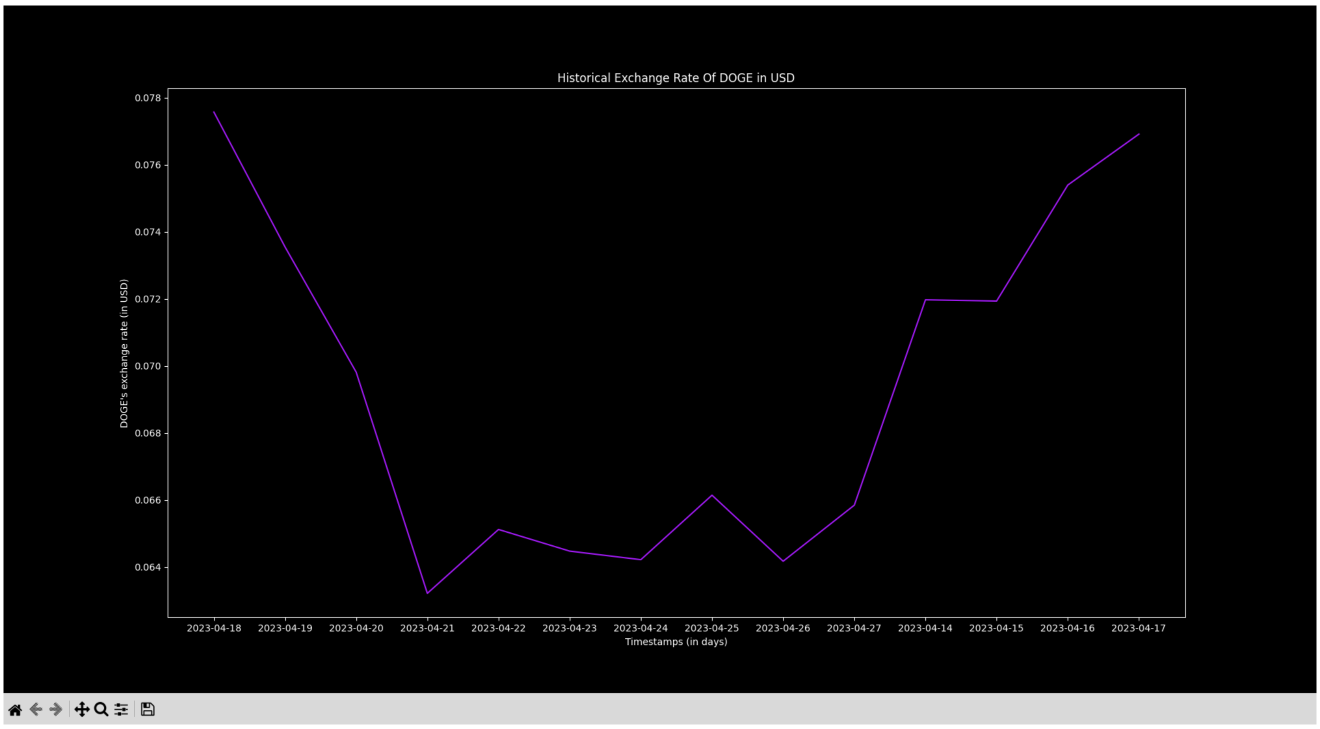


Target reached: DOGE: True

✓ DOGE	1000	10
☐ LTC	Target Price	Target Token Quantity

Currency: INR(₹)

Wi-Fi: Connected


Historical Exchange Rate Of DOGE in USD
0.078
0.076
0.074
0.072
0.070
0.068
0.066
0.064
DOGE's exchange rate (in USD)
2023-04-18
2023-04-19
2023-04-20
2023-04-21
2023-04-22
2023-04-23
2023-04-24
2023-04-25
2023-04-26
2023-04-27
2023-04-14
2023-04-15
2023-04-16
2023-04-17
Timestamps (in days)