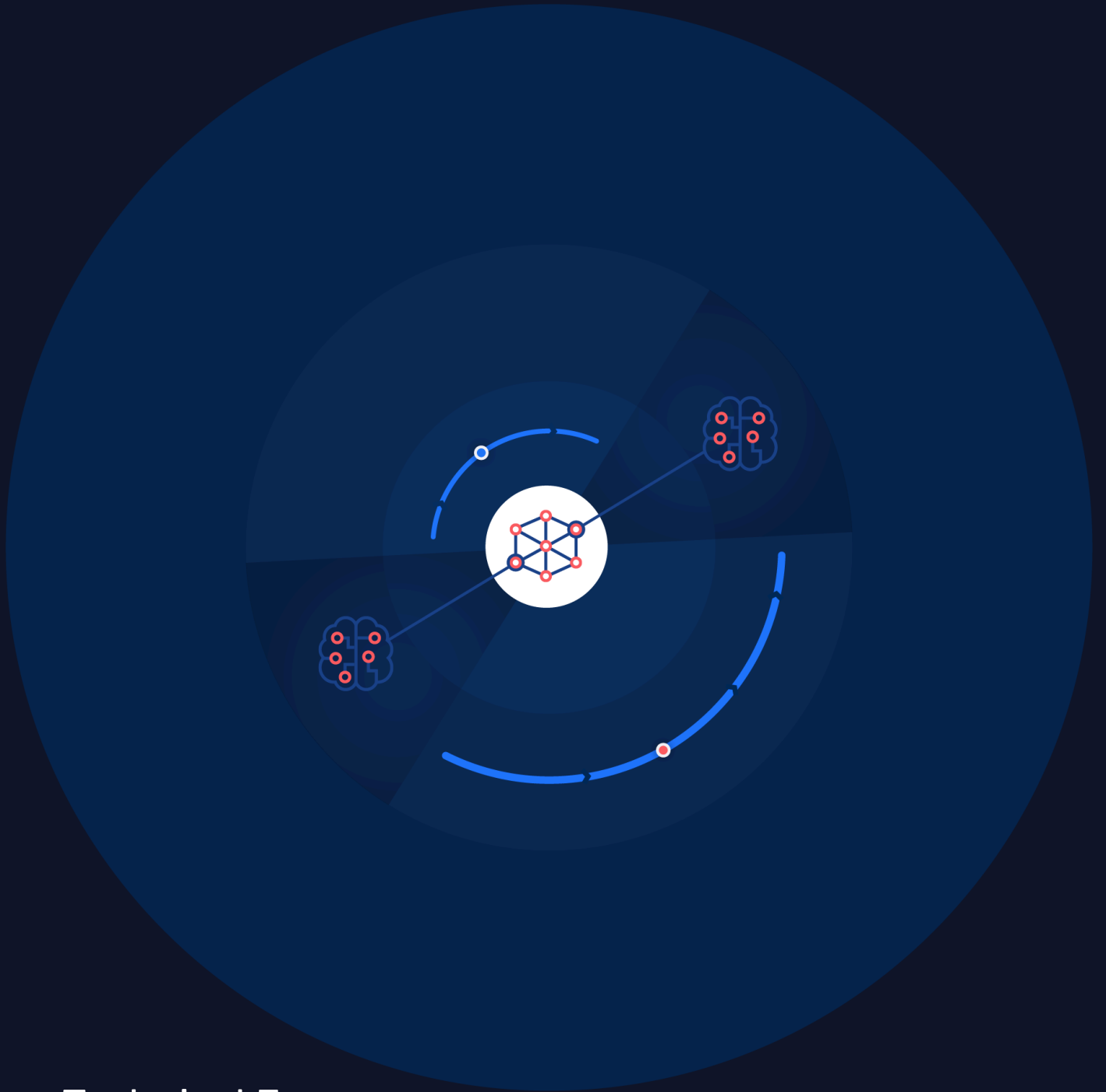




MUTT DATA




Technical Exam


**Do you have what it
takes to be a Mutt? 🧐**



This exam helps us evaluate your software engineering & data skills. There's **no need for you to complete all of the tasks - but the more, the better.**


We **strongly encourage you to** use Google, Stack Overflow or any other similar websites when you need to unblock yourself in the task or when you need further understanding.

This would be very ordinary on any given day of work. 


Obviously, we ask you to **refrain from** getting outside help from friends, colleagues or any other third-party people. 

Logistics


After confirming that you have received the exam, take your time to **push the code** with the answer to the GitHub repository we supplied.


You have up to 5 days to complete the exam. We will evaluate you on the last commit 5 days after the start of the test. This will be considered the final working version. 

If you see time running out, prioritize **showing your knowledge about the different areas** evaluated in the exam rather than just focusing on one of the exercises.

For some of the exercises you will need to **install PostgreSQL (or other tools)** in your local development environment. We'll provide you with a reasonable configuration to get you started. All the commands you used to install dependencies and software should be **documented in the README.md** with the goal of making your code reproducible in another machine. Even better if you use + share a **script, Dockerfile** or **docker-compose.yaml** or any tool to make this whole process even easier .

Evaluation Criteria

We are interested in getting a general idea of how you develop the solution. Also, you should **prioritize having clear code**, with classes, modules and following basic Object Oriented Programming practices, which are well looked upon. 

At Mutt we  to code following **best software engineering practices**. We **strongly encourage** these in your code -- be it via linting, documenting (in the form of docstrings and README.md), formatting, creating **reproducible environments**, providing quality



code, following SOLID principles and, why not, maybe seeing some software patterns traits in the answers.

Remember, we care less about having performant or “correct” code, and care more for **working, maintainable, reproducible** and **portable** software.

We hope you enjoy the challenge! 🧑💻

Exam Overview

Mutt’s core activity is to work with data, day in and day out. Data is extracted, processed, moved, transformed, stored and analyzed. This exam proposes a scenario where we will develop the code to create a small *data project*.

We will use a data source concerning crypto assets, with data about volume, prices and other metadata from various crypto tokens. To work with this information, we will use available programming tools at our disposal where we will have to extract crypto data via an API, set up a database to store this data, and run some SQL analysis on the stored data.

This whole exam could be a possible scenario of working in a project at Mutt Data, where we can find lots of different types of data; time series is just one of them.

Disclaimer

We recognize the growing role of AI based tools in modern software development and do not forbid their use in completing this code exercises.

However, we emphasize that the primary focus of our evaluation is on the overall quality of your solutions. This includes but is not limited to:

- **Modular Code:** Solutions should be broken down into manageable, reusable modules.
- **Cohesion:** Code should be logically organized and maintain a single, clear purpose within each module.
- **Reusability:** Your code should be designed in a way that it can be easily reused in different contexts or projects.
- **Reproducibility:** The solutions provided should produce consistent results across different environments and use cases.



We encourage you to demonstrate your problem-solving abilities, coding best practices, and deep understanding of the principles of software development.

We will not evaluate negatively the use of AI tools, although they should complement your skills, not replace them.

Environment Setup

We **strongly recommend** you read the following installation guides, but feel free to disregard them if you don't need them. They might not be plug and play, but should work with some simple edits.

PostgreSQL

The best option for this will probably be via Docker using the following Postgres image: https://hub.docker.com/_/postgres

If you need help installing Docker in your local machine we [recommend following its official guide](#).

Remember to **push this setup code** to the repo! Even better if you feel comfortable using+pushing a Dockerfile or docker-compose file.

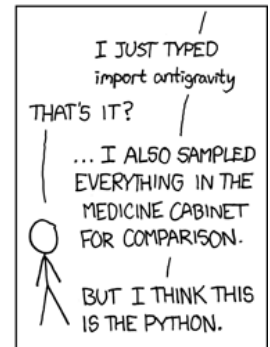
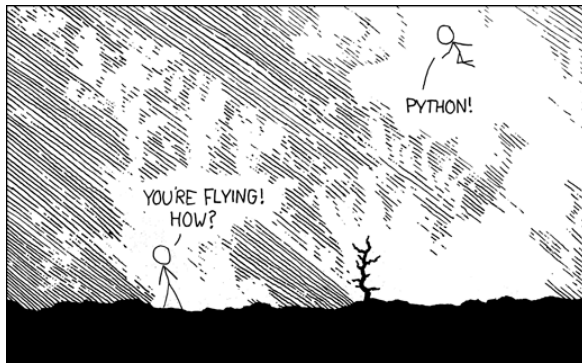
Relevant for later: setup the PostgreSQL server to listen on port 5432, which has inbound traffic permissions.

Python 3.8+

The code you develop should be compatible with any Python version ≥ 3.8 . Please document the Python version you've decided to use!



1. Getting crypto token data



Context

This exercise tries to evaluate your developer experience in Python software.

You will work with the **CoinGecko API**, which provides data on prices, volume, and other details for over 3,000 cryptocurrencies. To use the API, you'll need to generate an API key. Please follow the instructions in this [link](#) to create it.

To develop the following Python3 tasks, we recommend using the **requests**, **click (or argparse/typer)** and **logging** python libraries, but please use any libraries of your choice.

Tasks

1. Create a [command line Python app](#) that receives an [ISO8601 date](#) (e.g. 2017-12-30) and a coin identifier (e.g. bitcoin), and downloads data from `/coins/{id}/history?date=dd-mm-yyyy` endpoint for that whole day and stores it in a local file. To start, this app should store the API's response data in a local file. Save it in whichever file-format you judge as best for your problem with the corresponding date in the file's name.

An example endpoint would be :



<https://api.coingecko.com/api/v3/coins/bitcoin/history?date=30-12-2017>

Note: If you are having trouble working with the API and processing its response, then it is also valid to do this exercise with a mocked response.

2. Add proper Python logging to the script, and configure a CRON entry that will run the app every day at 3am for the identifiers bitcoin, ethereum and cardano. Document this in the README.md file of the repo.
3. Add an option in the app that will bulk-reprocess the data for a time range. It will receive the start and end dates, and store all the necessary days' data. Use whichever patterns, libraries and tools you prefer so that the progress for each day's process can be monitored and logged. Here you can bring your own assumptions if they simplify your work, just remember to comment them in the code or in the README.md.

Bonus: run the above process in a concurrent way (with a configurable limit), use any library/package of your preference for this such as **asyncio** or **multiprocessing**.

2. Loading data into the database



Context

Data's ephemeral in a script without a database. For this exercise we'll be using the command line app that downloads the data and we will store it in the Postgres instance



that we previously created to persist data.

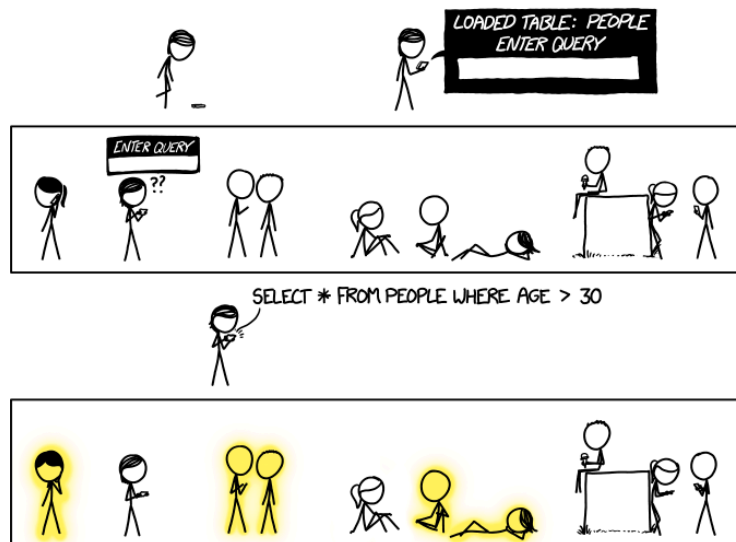
Remember to follow the recommended practices for dealing with databases in Python, such as using the **sqlalchemy** python library (and **alembic** if you find it helpful), but use whichever libraries you prefer.

Tasks

1. Create two tables in Postgres. The first one should contain the coin id, price in USD, date and a field that stores the whole JSON response. We'll be using a second table to store aggregated data, so it should contain the coin id, the year and month and the maximum/minimum values for that period. Write and push these SQL statements in one or multiple **.sql files**.
2. Modify the command line app so that an optional argument enables storing the data in a Postgres table, and updates the given month's max and min prices for the coin. Remember to add the code logic that deals with pre-existing data.



3. Analysing coin data with SQL



Context

This exercise tries to evaluate expertise in SQL language, specifically on DQL statements.

For the next task, we would like you to write SQL queries for certain cases/analysis and save them in one (or more) .sql files inside your repo.

Exam Help:

Couldn't get the data into Postgres? We've got your back: in the exam's Github repository you'll find a directory named **data** with a **coin_data.sql** file inside. This file contains the necessary data for your exercise. You can load it into a Docker Postgres container by running:

```
docker cp data/coin_data.sql <name-of-your-postgres-container>:/coin_data.sql
```

```
docker exec -it <name-of-your-postgres-container> bash
```

```
psql -U <your-postgres-user>
```

```
\i /coin_data.sql
```




Tasks

Create SQL queries that:

1. Get the average price for each coin by month.
2. Calculate for each coin, on average, how much its price has increased after it had dropped consecutively for more than 3 days. In the same result set include the current market cap in USD (obtainable from the JSON-typed column). Use any time span that you find best.



4. Finance meets Data Science

Context

Using the data stored in the Postgres database, we'd like to perform some further analysis with **pandas**. You may do the tasks in a Jupyter Notebook or as standalone scripts, whatever you prefer.

Exam Help:

Couldn't get the data into Postgres? We've got your back: in the exam's Github repository you'll find a directory named **data** with a **coin_data.sql** file inside. This file contains the necessary data for your exercise. You can load it into a Docker Postgres container by running:

Unset

```
docker cp data/coin_data.sql
<name-of-your-postgres-container>:/coin_data.sql

docker exec -it <name-of-your-postgres-container> bash
psql -U <your-postgres-user>
\i /coin_data.sql
```

Tasks

Develop python code to:



1. Plot the prices of bitcoin, ethereum and cardano for the last 30 days, commit those plots images in your repo.

Hint: You can use pandas' [read_sql_query](#), or [read_sql_table](#) to query data from Postgres.

2. Loading all data from the daily stock value table as a Dataframe, generate the following new features:

- a. Define 3 types of coins: "High Risk" if it had a 50% price drop on any two consecutive days, during a given calendar month, "Medium risk" if it dropped more than 20%, and "Low risk" for the rest (in this same fashion of consecutive days).
- b. For each row-day, add a column indicating the general trend of the price for the previous 7 days (T0 vs. T-1 through T-8), and the variance of the price for the same 7 days period.

3. To perform predictions on this time series we'll need to have a slightly different structure in the dataframe. Instead of having only the price and date for each row (plus the newly generated features), we'll also want the price of the last 7 days as columns as well as the next day's price, which will be used as a target variable.

- a. You can now add extra features if you'd like, for example with [feature scaling](#) or with [the skewness of the stock](#).
- b. Add any other time features you'd like such as what day of the week is this? Is this a weekend or weekday? What week of the year is this? What month? Etc.
- c. Play with the data regarding the volume transacted to create new features.
- d. **Bonus:** Add extra features for both China's and US' holidays using Python's [holiday](#) package.

4. **Regression:** For every (coin, date) row in your dataset with date **T0**, **predict the next day's price (T1) based on the previous 7 days of the stock's price and the previous features**. You can use a simple linear regression model for this task, however you can



present any model you think is best for the problem, even better if there is shown evidence of improvement over a linear regression.

Note, the seven rows with the earliest date will not have any previous dates on which to predict!

So you need to skip these predictions



MUTT DATA