Data Exam

Do you have what it takes to be a Mutter? 🤓

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 GitLab repository we supplied.

You have up to 36 hours to complete the exam. Experienced candidates should be able to solve it in **under 3 hours and we suggest not taking more than 6 hours** to finish it but do as you please **:)** We will evaluate you though on the last commit 36 hours 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 the AWS instance, we have provided a reasonable configuration for a development environment. All the commands you use to install dependencies and software should be **pushed to the repo**. You can **definitely work locally**, using the instance as your staging environment to test that your code works. 💻

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 greatly 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.

Environment Setup

For the following exercises you will need to install different services in the AWS instance we have provided.

We **strongly recommend** you read the following installation guides, but you can do as you please. 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

Docker should be installed, if you find that Docker is not part of your terminal's commands, then you might find this guide helpful to install it. This answer might be relevant too.

Remember to **push this setup code** to the repo.

Relevant for later: setup the PostgreSQL server to listen on port 15432, which has inbound traffic permissions.

Python 3.8+

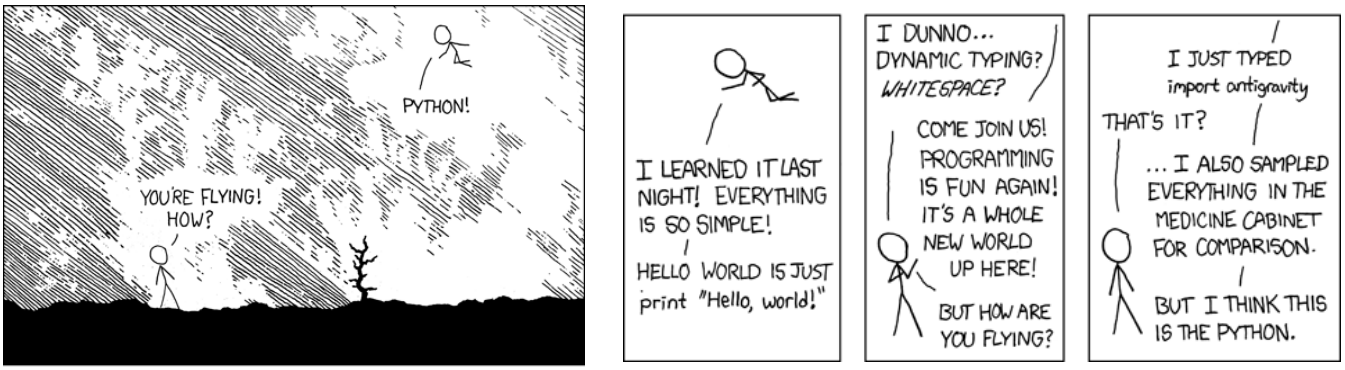
Here is a guide to install Python 3.8 that will work on an AWS EC2 server operating under Red Hat Enterprise LInux (RHEL):

https://techviewleo.com/how-to-install-python-on-amazon-linux/

Airflow

To install Airflow we strongly suggest to use the following docker images, else you can do it the traditional way (still fun 🤓): https://github.com/puckel/docker-airflow

1. Getting crypto token data 💸



Context

This exercise tries to evaluate developer experience in Python software. We'll be using a free API called **CoinGecko**, which tracks prices, volume and other information for more than 3K+ different crypto assets. The API doesn't require an access key.

For reference please consult the official API documentation.

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

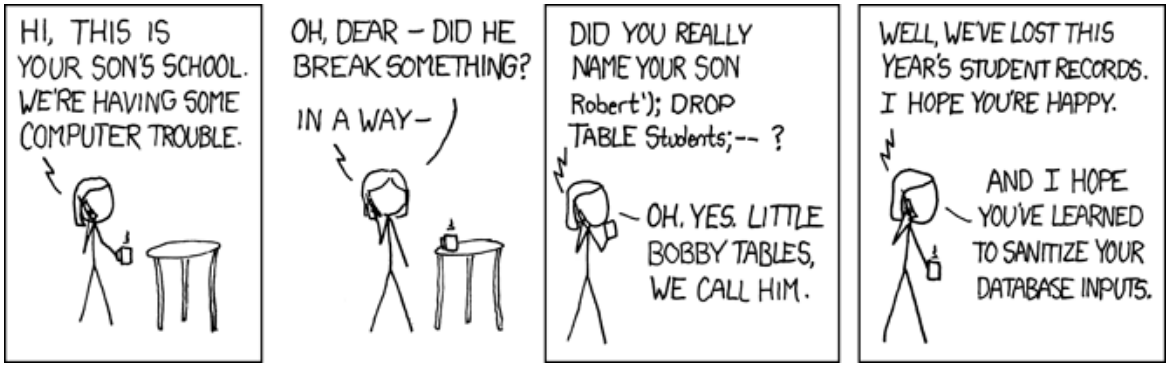
   Note: If you are having trouble working with the API and processing its response, then it is also valid to do this exercise with a mocked response.

2. Add proper Python logging to the script, and configure a CRON entry in the instance 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' 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

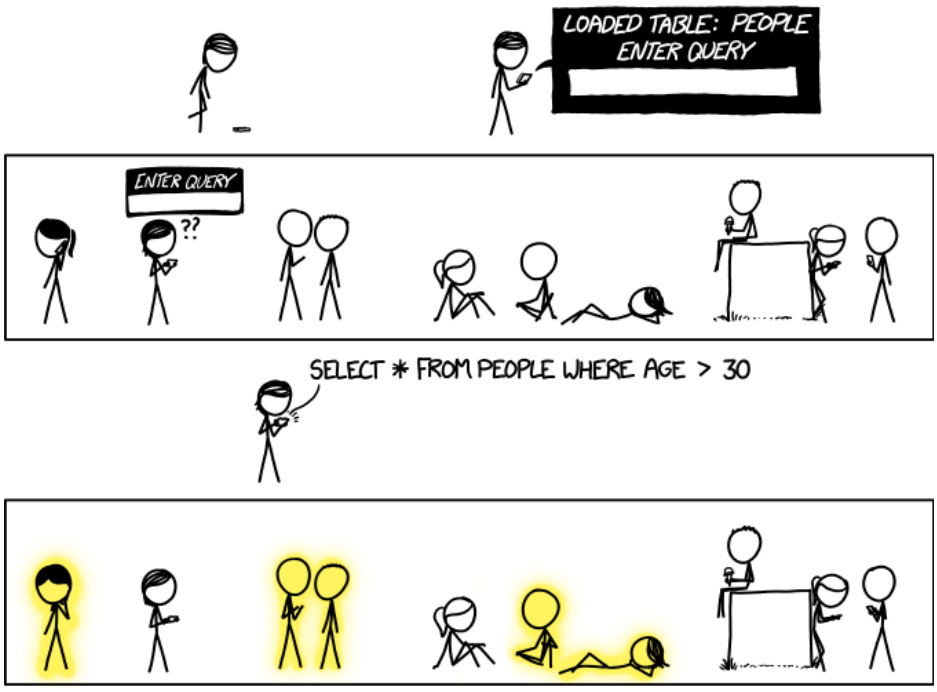
Data's ephemeral in a script without a database. For this exercise we'll be using the command line app that downloads the data and we will store it in the Postgres instance that we previously created to persist data.

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.
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:

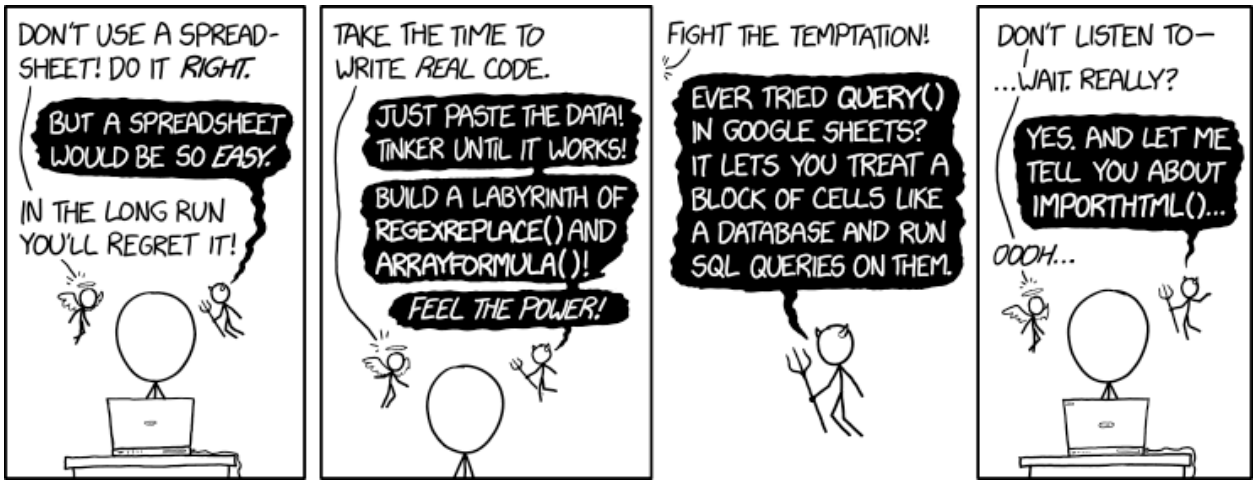
Couldn't get the data into Postgres or couldn't set up Postgres to work on the instance? We've got your back. In port **15432** there is a hidden Postgres already set up with the tables + data that you can use for this exercise. The user, password and db are `muttexam`.

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. Airflow 🏭



Setup

As the processing logic gets bigger and new tasks with complex dependencies are added, a simple CRON file will not suffice anymore. For this we bring the big guns -- we'll be using **Apache Airflow** to orchestrate the workflow.

If you aren't quite familiar with Airflow or if you need a refresher, you can always consult our *Friendly Intro* slides. You can also check the official docs**.**

Airflow is already installed in the instance, as you can see by running

```
$ airflow --help
```

It is already set up with a PostgreSQL backend, using a `LocalExecutor` setting.

To develop on it, please note that the dags folder is located in:

```
~/airflow/dags
```

where you'll need to put your dag files in order for them to schedule. Airflow is configured using the system Python3, so by running `pip3` the packages will also be accessible for the workers.

You can start up the components using the command

```
$ airflow standalone
```

and after everything is set up you can access the Airflow Webserver using port `8080`, with the username `admin` and password `muttexam`.

Exercises

1. Create an Airflow DAG that automates the daily data download from the crypto API, replacing the cron used in the previous exercise.
   The Airflow DAG should use variables defined in Airflow to download data for your specified currency or stock and for a given time window passed as arguments.
2. Add a connection in Airflow to your Postgres database that has the stock data.

3. Create an Airflow DAG with a sensor that waits for the data for a particular currency, and date to be loaded in the db. Once that sensor succeeds, plot the open and close prices for the last 30 days and store it locally in a file that you can commit to the repo.

   This DAG should also **log** the average difference between the open and closing price for all 30 days.

Bonus Section

So you are feeling adventurous? 🤓 Choose one of the following:

1. Add one unit-test to your previous python API code, for whichever function you find fit to be tested.
2. Use Poetry to manage library dependencies for the first and fourth exercise.