# Data Engineer test

Here at Gameloft every day is filled with new challenges to overcome and as our new recruit here are few puzzles for you to solve.

This technical test is divided into multiple sections each with its own set of questions.

These questions will test your overall skills across a variety of topics. Try to answer as many as you can but it is ok if you miss few. If you do not know the answer for any of the questions don’t stay stuck too long on it or you’ll be short on time, move forward and come back later.

# SQL

## 1.1 Consider a table GAMERS with the following columns: USER\_ID, INSTALL\_DATE, GAME, COUNTRY, DEVICE, INSTALL\_SOURCE, TOTAL\_AMOUNT\_SPENT

* Write a query to display total number of unique users, and sum of total amount spent for each game and install source
* Write a query that returns the top 3 spenders (users with positive amount spent) by country that are installed from ‘ua’.
* Write a query that gives the daily average revenue per game, with daily average revenue = sum of total revenue spent in that day / total unique players in that day

## 1.2 Write a query to get the max gap between active dates for each user in the following table:

| user\_id | Active\_date |
| --- | --- |
| 1 | 2019-01-02 |
| 1 | 2019-01-05 |
| 2 | 2019-01-01 |
| 2  2  2 | 2019-01-07  2019-01-21  2019-02-21 |
| 3 | 2019-01-01 |

(e.g, user 1 will have max. gap of 3 days, user 2 will have 30 days, and user 3 will have null) Note that:

* A user might not be active on a given date. E.g. user\_id 1 was not active on 2019-01-03 and 2019-01-04.
* A user might only be active once during his/her entire lifecycle. E.g. user\_id 3

## 1.3 You are given a table that contains information about how much users have spent in a game per day. The table has the following structure:

| user | day | cash |
| --- | --- | --- |
| a1 | 1 | 6 |
| a1 | 4 | 3 |
| a1 | 0 | 7 |
| a1 | 5 | 1 |
| a2 | 2 | 2 |
| a2 | 3 | 8 |
| a2 | 4 | 2 |
| … | … | … |

Write a sql to:

* cumulatively sum how much money each user has spent following day ascending order, as in column cumsum below
* create another new column to calculate the percentage of the cumulative total the user has spent per day. The output of your sql should look like this:

| user | day | cash | cumsum | percentage |
| --- | --- | --- | --- | --- |
| a1 | 0 | 7 | 7 | 41% |
| a1 | 1 | 6 | 13 | 76% |
| a1 | 4 | 3 | 16 | 94% |
| a1 | 5 | 1 | 17 | 100% |
| a2 | 2 | 2 | 2 | 17% |
| a2 | 3 | 8 | 10 | 83% |
| a2 | 4 | 2 | 12 | 100% |
| … | … | … | … | … |

Table name user\_spent assuming that:

1. The table only contains day\_number of ONE month.

2. Total\_cash(in my queries) >0

# Python

## General Programming

In this section we’ll test your overall python know how to.

Some of our games are using a third party tools for logs. These logs are stored in the cloud in the form of s3 buckets. You download those files in preparation to loading them into our data warehouse but you find that they are a complete mess. You’ll need to load those files and parse them with regular expressions to extract only the useful information, initialize corresponding python objections and at the end implement some extra functionality needed. The functionality can be used later to generate extra data that can be loaded into the data warehouse.

Also redefine/ change the implementation of any class/ method if the word “define” is mentioned as a comment before its code.

Also Add any missing import.

# define the class as an Enum type

**class** **CountryCode**:

FR = 'France'

US = 'USA'

# define the class as a dataclass

**class** **Game**:

game\_name: str *# unique identifier*

release\_date: str

# define the method as a classmethod with the appropriate argument

**def** read\_games\_from\_file() -> list:

*"""Write a class function that read games from a file called games.txt.*

*the file should be in the same directory as the running python script.*

*Use regular expressions to parse the text file and extract needed data*

*This function should be a class function*

*"""*

**pass**

**def** has\_game\_been\_released(self):

*"""In this function you'll check whether the game has been released or not.*

*by chekcing the current\_date and the release\_date.*

*you can use the datetime library to get today's date.*

*Note that transaction\_date is a str and not an object in this case.*

*"""*

**pass**

# define the class as a dataclass

**class** **Player**:

name: str *# unique identifier*

age: int

country: Enum

Games: Union[str, List[str]]

# define the method as a classmethod with the appropriate argument

**def** read\_players\_from\_file() -> list:

*"""Write a class function that read players from a file called players.txt.*

*the file should be in the same directory as the running python script.*

*Use regular expressions to parse the text file and extract needed data*

*This function should be a class function*

*"""*

**pass**

# define the class as a dataclass

**class** **Transaction**:

player: Player

Game: Game

transaction\_date: str

amount: float

**def** is\_transaction\_valid(self):

*"""Transaction is considered valid if it happens after game release.*

*Transactions with date in the future are considered invalid*

*"""*

**def** is\_transaction\_recent(self) -> bool:

*"""In this method you're going to check if the transaction happened in the past week.*

*You have the transaction\_date, and you can use the datetime library to get today's date.*

*Note that transaction\_date is a str and not an object in this case.*

*"""*

**pass**

# define the method as a classmethod with the appropriate argument

**def** read\_transactions\_from\_file() -> list:

*"""Write a class function that read transactions from a file called transactions.txt.*

*the file should be in the same directory as the running python script.*

*Use regular expressions to parse the text file and extract needed data*

*This function should be a class function*

*"""*

# define the method as a staticmethod, change arguments if needed

**def** calculate\_revenue(transactions, player) -> float:

*"""In this function given a player instance, search the list of transactions made by*

*the instance "player" and sum the amout in order to calculate the total revenue made by "player" instance.*

*"""*

**pass**

# define the class as a dataclass

**class** **RefundableTransaction**(Transaction):

*"""Add a data member to distinguish between refundable and non-refundable transactions*

*"""*

# define the method as a classmethod with the appropriate argument

**def** read\_transactions\_from\_file() -> list:

*"""override this function to load transactions that are refundable*

*"""*

**def** can\_refund(self):

*"""In this method you're going to check whether a transactions can be refunded or not.*

*A transaction is refundable if it meets the following creteria:*

*- transaction is recent*

*- the amout of the transaction is > 20*

*- has the refundable data attribute as only a portion of transactions are refundable*

*even if the meet other conditions*

*If any of the above conditions was unknown the transaction is considered non refundable*

*"""*

**pass**

**def** sort\_transactions(transactions, sort\_by='amount'):

*"""In this function you'll be implementing your method of choice for sorting transactions.*

*Transactions will be sorted depending on the sort\_by parameter, by default it's 'amount' but can have*

*the following list of values:*

*- amount*

*- transaction\_date*

*Choose the most efficient sort method you know.*

*"""*

**pass**

## General Data

As our new data engineer at Gameloft we’d like you to improve our infrastructure that query Snowflake our data warehouse. One of your colleagues has developed a small class to query the warehouse called Snowflake. It takes no arguments (credentials are encoded within the class) and has a single method called query(txt: str) -> pandas.DataFrame that send a query to Snowflake. You’ll be using that class to create a new one to help write and store query objects as pickle on local machine.

Snowflake class looks similar to

import pandas as pd  
  
  
class Snowflake():  
 def query(text: str) -> pd.DataFrame:  
 # Some code to execute queries against snowflake database  
 pass

It’s stored in a package named de\_tools so make sure to import it in your code when you need it.

The definition of the class you need to fill is here

class SnowflakeQuery():  
 def \_\_init\_\_(self, query: str, binary\_file\_name: str):  
 # Fill this function here  
 pass  
  
 # Fill any more code you need to allow the object to get pickled and stored in a binary file  
 # It could be a helper function or anything else to make the pickling process easy  
 # example  
 # def store(self):  
 # pass

Then you need to write a python function that prompts the user on the command line to write his/her multiline query and store it as a pickle object on their machine using the SnowflakeQuery class you have developed.

def take\_user\_query():  
 # Fill in this method section  
 # This method needs to prompt the user to provide their input query on the command line  
 pass

The output of the function should be the following

Hello, could you please provide the name of the file for your query, the file will be stored on your machine.  
<user input>  
Could you please provide the query you want to store the query definition will end with ‘;’  
<user input: query>

<user input: query>

<user input: query>;  
Thank you, your query has been registered under the file name <replace with the file name already provided by the user>

## DataFrame manipluation

As our new Data Engineer hire we’d like you to help cleaning and wrangling some data we have received from one of our game teams.

Given the two data files you have data1.csv and data2.csv please do the following:

* Load the data and combine it into a new pandas DataFrame by joining them on the key column user\_id
* Add the following two columns day, month that have the numerical day and month value taken from the column event\_date
* Filter the data frame to unique rows based on the key column (user\_id)
* Store the final DataFrame on local machine with the file name cleaned\_data.csv
* Fill in the following functions

def filter\_on\_column(df:pd.DataFrame, col\_name: str, filter\_value) -> pd.DataFrame:  
 # return the passed data frame df after filtering it, removing all data that doesn't have 'filter\_value' in column 'col\_name'.  
 pass  
  
  
def df\_to\_dict(df:pd.DataFrame) -> dict:  
 # convert the data frame df into a dictionary where each column in the data frame  
 # correspond to a key in the dictionary and its value are all the column values.  
 # e.g.,  
 # | A | B | ----\ {'A': [1], 'B': [2]}  
 # | 1 | 2 | ----|  
 pass

## Airflow

You have been delegated a task by one of your colleagues to schedule some Snowflake queries, and since you are a big Airflow fan you decided to use it for this task. Create a file called schedule\_snowflake\_queries\_dag.py that will schedule the airflow tasks as defined below. All described tasks that mention ‘execute a query’ do a very simple thing, given a string query the task will execute it. Use the class SnowflakeQuery that you have defined earlier to help you with each task that require query execution. You don’t need to write any SQL queries as they will be filled by your colleagues later and you can leave them as empty strings. The Airflow dag needs to be run daily at midnight, if a task fail it need to retry twice, and in case of failure sends an email to global-de@gameloft.com. Here are the tasks you need to define:

* start\_task: a dummy task that uses the DummyOperator that will be the first task to execute, execution order is (1)
* import\_user\_data\_task: a task using PythonOperator that executes a query that will be filled later by colleague to import user data, execution order is (2)
* import\_currency\_conversion\_task: a task using PythonOperator that executes a query that will be filled later by colleague to import currency a conversion table, execution order is (2)
* aggregate\_transaction\_task: a task using PythonOperator executes a query that will be filled later by colleague to aggregate imported data on a daily level, execution order is (3)
* end\_task: a dummy task that uses the DummyOperator that will be the last task to execute, execution order is (4)

Please notice that any task starting with ‘import\_’ doesn’t require you to import any data yourself since in snowflake you can run queries that import data from some data storage types. Make sure to define task execution order correctly, and the file schedule\_snowflake\_queries\_dag.py need to be self contained and have all the code needed for it to run.

# Git

At a Friday evening you have finished your awesome feature and you want to push your code to git. You have a clean repository created with a main branch called master. It has no commits and you need to set it up before you pushing your code. List the commands needed to:

* Clone the repo to your machine
* Create a new branch named my\_awesome\_feature
* Add your code to that branch
* Push the new branch to its origin/remote
* Merge the new branch locally on your machine to master
* Push the new master changes to origin/remote