Proto Challenge

This is an R Markdown Notebook to address the coding challenge presented to Peter Edstrom on January 5th, 2018.

Challenge Goals

The full challenge text can be found in README.md. We aim to parse data.dat and answer the five questions:

- What is the total amount in dollars of debits?
- What is the total amount in dollars of credits?
- How many autopays were started?
- How many autopays were ended?
- What is balance of user ID 2456938384156277127?

Log Specification

MPS7 transaction log specification:

Header: | 4 byte magic string "MPS7" | 1 byte version | 4 byte (uint32) # of records |

Record: | 1 byte record type enum | 4 byte (uint32) Unix timestamp | 8 byte (uint64) user ID |

Record type enum:

- 0x00: Debit
- 0x01: Credit
- 0x02: StartAutopay
- 0x03: EndAutopay

Setup and Helper Functions

I started experimenting with the `readBin()` function. However, I found found that using some of the built-in modes such as the obvious `character` would return too much of the file. For example:

```
to.read = file("byte-reader/data.dat", "rb")
readBin(to.read, character(), n=1, size=4)
```

```
## [1] "MPS7\001"
```

```
close(to.read)
```

Notice the trailing `\001`. My understanding is that `character` is dependent on a zero-terminator character string, which we clearly can not count on. The `size=4` appears to be ignored in `character` modes.

Using the `raw` mode, and converting to a character string afterwords seems like a decent fall-back. However, I ran into a number of issues:

- `size` is always of size 1 in `raw` mode
- `n`, used for retrieving multiple records is also unavailable in `raw` mode
- and `raw` seems to have no option other than to retrieve 1 byte at a time.

```
to.read = file("byte-reader/data.dat", "rb")
raw_data <- readBin(to.read, raw())
raw_data
```

```
## [1] 4d
```

```
rawToChar(raw_data)
```

```
## [1] "M"
```

```
close(to.read)
```

As you can see in these results, `4d` converts to our very first character, `M`.

Having not found a sufficient way to read a specific number of bytes in one go, I decided to write a short function. If someone can find a better way to do this upon code review, I absolutely welcome a refactor.

```
retrieveNbytes <- function(file_name, number_of_bytes) {
  count <- number_of_bytes
  raw_bytes <- c()
  while (count > 0) {
    count <- count - 1
    raw_bytes <- c(raw_bytes, readBin(file_name, raw()))
  }
  return(raw_bytes)
}
```

Header Parsing

Putting the new `retrieveNbytes` function to use in extracting the first 4 characters:

```
to.read = file("byte-reader/data.dat", "rb")
rawToChar(retrieveNbytes(to.read,4))
```

```
## [1] "MPS7"
```

```
close(to.read)
```

We have found the expected magic string!

Reading the rest of the header:

```
to.read = file("byte-reader/data.dat", "rb")
magic_string <- rawToChar(retrieveNbytes(to.read,4))
magic_string
```

```
## [1] "MPS7"
```

```
version <- readBin(to.read, integer(), size=1)
version
```

```
## [1] 1
```

```
records <- as.integer(retrieveNbytes(to.read,4))[4]
records
```

```
## [1] 71
```

```
close(to.read)
```

Note that this is not quite the right for the record count. I'm reading 4 bytes, but each byte is stored in an array and for simplicity I'm just jumping to the 4th item and using it. There are 71 records reported, but for any record count greater than 255, this will fail.

Let's fix this with a quick function:

```
bytesToInteger <- function(file_name, number_of_bytes) {
  raw_bytes <- retrieveNbytes(file_name,number_of_bytes)
  count <- number_of_bytes
  total <- 0
  while (count > 0) {
    t <- as.integer(raw_bytes)[count] * 2^((number_of_bytes-count)*8)
    total <- total + t
    count <- count - 1
  }
  return(total)
}
```

Notice the similarities already emerging between `retrieveNbytes` and `bytesToInteger`. Perhaps a strategy that would create a function similar to Ruby's `unpack` function would make sense. Curiously, I spent some time researching alternatives to `unpack` and found very few. Even in Python, it appears that there is not a function that works as well as Ruby's implementation.

Regardless, we have removed the 255 record limit.

Let's extract the header processing into a function:

```
processHeader <- function() {
  magic_string <- rawToChar(retrieveNbytes(to.read,4))
  version <- readBin(to.read, integer(), size=1)
  records <- bytesToInteger(to.read,4)
}
```

```
to.read = file("byte-reader/data.dat", "rb")
processHeader()
magic_string
```

```
## [1] "MPS7"
```

```
version
```

```
## [1] 1
```

```
records
```

```
## [1] 71
```

```
close(to.read)
```

Record Parsing

```
to.read = file("byte-reader/data.dat", "rb")
processHeader()
type_enum <- readBin(to.read, integer(), size=1)
type_enum
```

```
## [1] 0
```

```
timestamp <- bytesToInteger(to.read,4)
timestamp
```

```
## [1] 1393108945
```

```
user_id <- bytesToInteger(to.read,8)
user_id
```

```
## [1] 4.136354e+18
```

```
close(to.read)
```

For the type_enum, `0` is a Debit.

For the timestamp, I validated it with https://www.unixtimestamp.com, which converts the UNIX timestamp into a more readable representation, `02/22/2014 @ 10:42pm (UTC)`. This date-time is neither wildly in the past nor in the far future.

For the user_id, `4.136354e+18` seems like an unusually long user ID, however when compared to the supplied user ID in the challenge `2456938384156277127` we find that they are both 19 digits long.

So I'd say these results seem reasonable! Let's carry on.

As a debit, this record comes with an additional field, an 8 byte float for the amount.

```
to.read = file("byte-reader/data.dat", "rb")
processHeader()
type_enum <- readBin(to.read, integer(), size=1)
timestamp <- bytesToInteger(to.read,4)
user_id <- bytesToInteger(to.read,8)
amount <- readBin(to.read, double(), n=1, size=8, endian="big")
amount
```

```
## [1] 604.2743
```

```
close(to.read)
```

`604.2743` *feels* like a reasonable amount, however I should note that at this point in the exercise that there is very little in the data that will provide feedback on the validity of the results. The data is numerical and there is little guidance we can assume on valid minimums or maximums.

I see three assumptions we might be able to make:

- Timestamps will be in the future of the UNIX epoch.
- A transaction log would not include records from the future.
- Record types are always and *only* the 4 types listed.

Let's put a function together that will read a single record:

```
processRecord <- function() {
  type_enum <- readBin(to.read, integer(), size=1)
  if (length(type_enum) == 0) { return(FALSE) }
  timestamp <- bytesToInteger(to.read,4)
  user_id <- bytesToInteger(to.read,8)
  if (type_enum == 0 | type_enum == 1) {
    amount <- readBin(to.read, double(), n=1, size=8, endian="big")
  } else {
    amount <- NA
  }
  print(paste("Type=", type_enum, " Timestamp=", timestamp, " User ID=", user_id, " Amount=", amount, s
  return(TRUE)
}

to.read = file("byte-reader/data.dat", "rb")
```

```
processHeader()
p <- processRecord()
```

```
## [1] "Type=0 Timestamp=1393108945 User ID=4136353673894269440 Amount=604.274335557087"
```

```
p <- processRecord()
```

```
## [1] "Type=1 Timestamp=1416458602 User ID=1486945396868222976 Amount=825.129614841758"
```

```
p <- processRecord()
```

```
## [1] "Type=0 Timestamp=1398140960 User ID=1019092597011251456 Amount=668.673048640753"
```

```
p <- processRecord()
```

```
## [1] "Type=2 Timestamp=1390539963 User ID=3724427934598139904 Amount=NA"
```

```
p <- processRecord()
```

```
## [1] "Type=0 Timestamp=1416760748 User ID=6837272077571506176 Amount=397.738761904711"
```

```
close(to.read)
```

Reading the first 5 records seems to work!

As an aside: I feel like I'm getting sloppy with the global variables - these functions are not quite as atomic as I would prefer. Making a note of this to-do for future consideration.

Next up: Loop through all of the records and stop successfully at the end.

```
to.read = file("byte-reader/data.dat", "rb")
processHeader()
while (processRecord()) {
  #do a thing
}
```

```
## [1] "Type=0 Timestamp=1393108945 User ID=4136353673894269440 Amount=604.274335557087"
## [1] "Type=1 Timestamp=1416458602 User ID=1486945396868222976 Amount=825.129614841758"
## [1] "Type=0 Timestamp=1398140960 User ID=1019092597011251456 Amount=668.673048640753"
## [1] "Type=2 Timestamp=1390539963 User ID=3724427934598139904 Amount=NA"
## [1] "Type=0 Timestamp=1416760748 User ID=6837272077571506176 Amount=397.738761904711"
## [1] "Type=1 Timestamp=1416425125 User ID=7979830878773244928 Amount=608.134300502391"
## [1] "Type=2 Timestamp=1407728826 User ID=169747289235870464 Amount=NA"
## [1] "Type=0 Timestamp=1407057140 User ID=6555455521637047296 Amount=163.328360505277"
## [1] "Type=0 Timestamp=1416615422 User ID=6823688420765684736 Amount=384.616707684406"
## [1] "Type=3 Timestamp=1389541294 User ID=7178585665953385472 Amount=NA"
## [1] "Type=1 Timestamp=1402457849 User ID=3018469034978866176 Amount=154.663904470346"
## [1] "Type=0 Timestamp=1405988060 User ID=1518491488393966848 Amount=882.478098901413"
## [1] "Type=3 Timestamp=1413632673 User ID=2477346412368114688 Amount=NA"
## [1] "Type=1 Timestamp=1411685979 User ID=3281373847403844608 Amount=828.564720114962"
## [1] "Type=2 Timestamp=1416958471 User ID=685213522303989632 Amount=NA"
## [1] "Type=0 Timestamp=1405300715 User ID=4596876061716608000 Amount=654.859304506223"
## [1] "Type=2 Timestamp=1417477182 User ID=4011359550169803264 Amount=NA"
## [1] "Type=0 Timestamp=1388954072 User ID=6648738534997006336 Amount=286.292471682929"
## [1] "Type=3 Timestamp=1401299706 User ID=7979830878773244928 Amount=NA"
## [1] "Type=0 Timestamp=1404427639 User ID=5508906111153314816 Amount=812.367313370376"
## [1] "Type=3 Timestamp=1391406507 User ID=904175229436884864 Amount=NA"
## [1] "Type=0 Timestamp=1409938379 User ID=2015796113853353216 Amount=600.594773530842"
## [1] "Type=1 Timestamp=1415278068 User ID=3875821099972261376 Amount=950.714846514806"
```

```
## [1] "Type=1 Timestamp=1405640186 User ID=6768616184571698176 Amount=596.257320840476"
## [1] "Type=2 Timestamp=1404620303 User ID=7812633826533729280 Amount=NA"
## [1] "Type=1 Timestamp=1418913116 User ID=1247117451352823296 Amount=222.005488417048"
## [1] "Type=0 Timestamp=1405282673 User ID=2456938384156277248 Amount=248.584931533825"
## [1] "Type=0 Timestamp=1412703694 User ID=8505906760983331840 Amount=955.291966268199"
## [1] "Type=0 Timestamp=1388934017 User ID=4228385537401050624 Amount=873.446497687561"
## [1] "Type=0 Timestamp=1412688779 User ID=3797840465501570560 Amount=308.101896264497"
## [1] "Type=2 Timestamp=1392255769 User ID=2038354632042919168 Amount=NA"
## [1] "Type=0 Timestamp=1397081853 User ID=7471037767326702592 Amount=826.83306114578"
## [1] "Type=2 Timestamp=1399551130 User ID=3513954178729516032 Amount=NA"
## [1] "Type=0 Timestamp=1397205932 User ID=4765185273242274816 Amount=480.114434344636"
## [1] "Type=0 Timestamp=1414233821 User ID=6842348953158377472 Amount=534.83373879882"
## [1] "Type=0 Timestamp=1411844475 User ID=8219753787156836352 Amount=998.117737057352"
## [1] "Type=3 Timestamp=1407106856 User ID=8424619375577601024 Amount=NA"
## [1] "Type=0 Timestamp=1395001682 User ID=1266660370390831104 Amount=400.585154174679"
## [1] "Type=0 Timestamp=1396336714 User ID=7724259785917765632 Amount=279.470452162317"
## [1] "Type=0 Timestamp=1394285231 User ID=7756793223429038080 Amount=162.765221711003"
## [1] "Type=3 Timestamp=1416562451 User ID=8599028105427697664 Amount=NA"
## [1] "Type=0 Timestamp=1390185454 User ID=4724875543908343808 Amount=142.334881091114"
## [1] "Type=0 Timestamp=1389850890 User ID=2691316960514504704 Amount=633.149717991664"
## [1] "Type=2 Timestamp=1418380705 User ID=1461491922069312000 Amount=NA"
## [1] "Type=1 Timestamp=1394357363 User ID=5600924393587988480 Amount=741.304332782179"
## [1] "Type=0 Timestamp=1415943274 User ID=4674073639784954880 Amount=509.362588430989"
## [1] "Type=0 Timestamp=1416613150 User ID=6710715718024909824 Amount=309.838527001328"
## [1] "Type=0 Timestamp=1396779961 User ID=242253255677188736 Amount=439.661322679957"
## [1] "Type=2 Timestamp=1406564840 User ID=8761626118042981376 Amount=NA"
## [1] "Type=0 Timestamp=1391796075 User ID=1674879938132494592 Amount=274.754743561738"
## [1] "Type=0 Timestamp=1417532923 User ID=4270785598083309568 Amount=392.855866342082"
## [1] "Type=0 Timestamp=1404193974 User ID=897079919269759744 Amount=600.818070547545"
## [1] "Type=3 Timestamp=1402339411 User ID=1274435346255131648 Amount=NA"
## [1] "Type=1 Timestamp=1394845735 User ID=788787457839692032 Amount=682.976969410962"
## [1] "Type=0 Timestamp=1419199127 User ID=1518491488393966848 Amount=98.8324081353579"
## [1] "Type=0 Timestamp=1405000357 User ID=61122968712918072 Amount=272.348299005577"
## [1] "Type=0 Timestamp=1405110299 User ID=4170845066679832576 Amount=556.436996104262"
## [1] "Type=0 Timestamp=1403503729 User ID=3828453040100642816 Amount=537.911317455447"
## [1] "Type=2 Timestamp=1415281776 User ID=5498777567991819264 Amount=NA"
## [1] "Type=0 Timestamp=1389762186 User ID=4280841143732940800 Amount=313.447374499911"
## [1] "Type=0 Timestamp=1395510618 User ID=6573152408751521792 Amount=707.339948854931"
## [1] "Type=3 Timestamp=1412761292 User ID=1210013397032407040 Amount=NA"
## [1] "Type=1 Timestamp=1398697332 User ID=9016747369828896768 Amount=552.293130341084"
## [1] "Type=0 Timestamp=1391793265 User ID=8859025831836354560 Amount=891.239204267491"
## [1] "Type=1 Timestamp=1416760748 User ID=6837272077571506176 Amount=397.738761904711"
## [1] "Type=1 Timestamp=1416615422 User ID=6823688420765684736 Amount=384.616707684406"
## [1] "Type=1 Timestamp=1405282673 User ID=2456938384156277248 Amount=248.584931533825"
## [1] "Type=1 Timestamp=1412703694 User ID=8505906760983331840 Amount=955.291966268199"
## [1] "Type=1 Timestamp=1389850890 User ID=2691316960514504704 Amount=633.149717991664"
## [1] "Type=1 Timestamp=1416613150 User ID=6710715718024909824 Amount=309.838527001328"
## [1] "Type=1 Timestamp=1391796075 User ID=1674879938132494592 Amount=274.754743561738"
## [1] "Type=1 Timestamp=1395510618 User ID=6573152408751521792 Amount=707.339948854931"
```

```
close(to.read)
```

Refactor the Data into a Data Frame

Let's refactor our functions and store the values into a data frame.

In the following revised `processRecord` function we now a vector of the record values:

```
processRecord <- function() {
  type_enum <- readBin(to.read, integer(), size=1)
  if (length(type_enum) == 0) { return(c()) }
  timestamp <- bytesToInteger(to.read,4)
  user_id <- bytesToInteger(to.read,8)
  if (type_enum == 0 | type_enum == 1) {
    amount <- readBin(to.read, double(), n=1, size=8, endian="big")
  } else {
    amount <- NA
  }
  return(c(type_enum, timestamp, user_id, amount))
}
```

In our main function we capture the vector returned from `processRecord` and add the results to a data.frame, `df`.

```
to.read = file("byte-reader/data.dat", "rb")
processHeader()
df <- data.frame(matrix(ncol = 4, nrow = 0))
while (length(r <- processRecord()) > 1) {
  df <- rbind(df, r)
}
colnames(df) <- c("type_enum", "timestamp", "user_id", "amount")
close(to.read)
df
```

```
##    type_enum  timestamp      user_id    amount
## 1          0 1393108945 4.136354e+18 604.27434
## 2          1 1416458602 1.486945e+18 825.12961
## 3          0 1398140960 1.019093e+18 668.67305
## 4          2 1390539963 3.724428e+18        NA
## 5          0 1416760748 6.837272e+18 397.73876
## 6          1 1416425125 7.979831e+18 608.13430
## 7          2 1407728826 1.697473e+17        NA
## 8          0 1407057140 6.555456e+18 163.32836
## 9          0 1416615422 6.823688e+18 384.61671
## 10         3 1389541294 7.178586e+18        NA
## 11         1 1402457849 3.018469e+18 154.66390
## 12         0 1405988060 1.518491e+18 882.47810
## 13         3 1413632673 2.477346e+18        NA
## 14         1 1411685979 3.281374e+18 828.56472
## 15         2 1416958471 6.852135e+17        NA
## 16         0 1405300715 4.596876e+18 654.85930
## 17         2 1417477182 4.011360e+18        NA
## 18         0 1388954072 6.648739e+18 286.29247
## 19         3 1401299706 7.979831e+18        NA
## 20         0 1404427639 5.508906e+18 812.36731
## 21         3 1391406507 9.041752e+17        NA
## 22         0 1409938379 2.015796e+18 600.59477
## 23         1 1415278068 3.875821e+18 950.71485
```

```
## 24         1 1405640186 6.768616e+18 596.25732
## 25         2 1404620303 7.812634e+18        NA
## 26         1 1418913116 1.247117e+18 222.00549
## 27         0 1405282673 2.456938e+18 248.58493
## 28         0 1412703694 8.505907e+18 955.29197
## 29         0 1388934017 4.228386e+18 873.44650
## 30         0 1412688779 3.797840e+18 308.10190
## 31         2 1392255769 2.038355e+18        NA
## 32         0 1397081853 7.471038e+18 826.83306
## 33         2 1399551130 3.513954e+18        NA
## 34         0 1397205932 4.765185e+18 480.11443
## 35         0 1414233821 6.842349e+18 534.83374
## 36         0 1411844475 8.219754e+18 998.11774
## 37         3 1407106856 8.424619e+18        NA
## 38         0 1395001682 1.266660e+18 400.58515
## 39         0 1396336714 7.724260e+18 279.47045
## 40         0 1394285231 7.756793e+18 162.76522
## 41         3 1416562451 8.599028e+18        NA
## 42         0 1390185454 4.724876e+18 142.33488
## 43         0 1389850890 2.691317e+18 633.14972
## 44         2 1418380705 1.461492e+18        NA
## 45         1 1394357363 5.600924e+18 741.30433
## 46         0 1415943274 4.674074e+18 509.36259
## 47         0 1416613150 6.710716e+18 309.83853
## 48         0 1396779961 2.422533e+17 439.66132
## 49         2 1406564840 8.761626e+18        NA
## 50         0 1391796075 1.674880e+18 274.75474
## 51         0 1417532923 4.270786e+18 392.85587
## 52         0 1404193974 8.970799e+17 600.81807
## 53         3 1402339411 1.274435e+18        NA
## 54         1 1394845735 7.887875e+17 682.97697
## 55         0 1419199127 1.518491e+18  98.83241
## 56         0 1405000357 6.112297e+16 272.34830
## 57         0 1405110299 4.170845e+18 556.43700
## 58         0 1403503729 3.828453e+18 537.91132
## 59         2 1415281776 5.498778e+18        NA
## 60         0 1389762186 4.280841e+18 313.44737
## 61         0 1395510618 6.573152e+18 707.33995
## 62         3 1412761292 1.210013e+18        NA
## 63         1 1398697332 9.016747e+18 552.29313
## 64         0 1391793265 8.859026e+18 891.23920
## 65         1 1416760748 6.837272e+18 397.73876
## 66         1 1416615422 6.823688e+18 384.61671
## 67         1 1405282673 2.456938e+18 248.58493
## 68         1 1412703694 8.505907e+18 955.29197
## 69         1 1389850890 2.691317e+18 633.14972
## 70         1 1416613150 6.710716e+18 309.83853
## 71         1 1391796075 1.674880e+18 274.75474
## 72         1 1395510618 6.573152e+18 707.33995
```

A summary of the data frame below reveals a fair amount of consistency. *Amounts* range between $98.83 and $998.12. *Timestamps* are closely clustered, and most *user_ids* are 19 digits long. Though there is at least one user_id that is only 17 digits long.

```
summary(df)
```

```
##    type_enum        timestamp           user_id              amount
##  Min.   :0.0000   Min.   :1.389e+09   Min.   :6.112e+16   Min.   : 98.83
##  1st Qu.:0.0000   1st Qu.:1.395e+09   1st Qu.:1.931e+18   1st Qu.:308.54
##  Median :0.5000   Median :1.405e+09   Median :4.250e+18   Median :536.37
##  Mean   :0.8611   Mean   :1.405e+09   Mean   :4.513e+18   Mean   :523.65
##  3rd Qu.:1.2500   3rd Qu.:1.414e+09   3rd Qu.:6.827e+18   3rd Qu.:701.25
##  Max.   :3.0000   Max.   :1.419e+09   Max.   :9.017e+18   Max.   :998.12
##                                                           NA's   :18
```

Answering the Questions

What is the total amount in dollars of debits?

```
debit_records <- df[df$type_enum==0,]
sum_of_debits <- sum(debit_records$amount)
sum_of_debits
```

```
## [1] 18203.7
```

The sum of the debit amounts is **$18,203.70**

What is the total amount in dollars of credits?

```
credit_records <- df[df$type_enum==1,]
sum_of_credits <- sum(credit_records$amount)
sum_of_credits
```

```
## [1] 10073.36
```

The sum of the credit amounts is **$10,073.36**

How many autopays were started?

```
start_autopay_records <- df[df$type_enum==2,]
count_start_autopay <- nrow(start_autopay_records)
count_start_autopay
```

```
## [1] 10
```

There are **10** StartAutopay records.

How many autopays were ended?

```
end_autopay_records <- df[df$type_enum==3,]
count_end_autopay <- nrow(end_autopay_records)
count_end_autopay
```

```
## [1] 8
```

There are **8** StartAutopay records.

What is balance of user ID 2456938384156277127?

```
user_records <- df[df$user_id==2456938384156277127,]
user_records
```

```
##    type_enum  timestamp      user_id   amount
## 27         0 1405282673 2.456938e+18 248.5849
## 67         1 1405282673 2.456938e+18 248.5849
```

As you can see, there are two records for the user in question. It is pretty easy to tell that the two records will cancel each other out (one is a debit, one is a credit, and both have the same amount). However if this was a larger set of data I'd calculate it something like the following:

```
sum_of_user_credits <- sum(user_records[user_records$type_enum==1,]$amount)
sum_of_user_debits <- sum(user_records[user_records$type_enum==0,]$amount)
user_balance <- sum_of_user_credits - sum_of_user_debits
user_balance
```

```
## [1] 0
```

And there it is. The balance for user ID 2456938384156277127 is **$0.00**.

Final Code

```
# Functions

processHeader <- function() {
  magic_string <- rawToChar(retrieveNbytes(to.read,4))
  version <- readBin(to.read, integer(), size=1)
  records <- bytesToInteger(to.read,4)
}

retrieveNbytes <- function(file_name, number_of_bytes) {
  count <- number_of_bytes
  raw_bytes <- c()
  while (count > 0) {
    count <- count - 1
    raw_bytes <- c(raw_bytes, readBin(file_name, raw()))
  }
  return(raw_bytes)
}

bytesToInteger <- function(file_name, number_of_bytes) {
  raw_bytes <- retrieveNbytes(file_name,number_of_bytes)
  count <- number_of_bytes
  total <- 0
  while (count > 0) {
    t <- as.integer(raw_bytes)[count] * 2^((number_of_bytes-count)*8)
    total <- total + t
    count <- count - 1
  }
  return(total)
}
```

```r
processRecord <- function() {
  type_enum <- readBin(to.read, integer(), size=1)
  if (length(type_enum) == 0) { return(c()) }
  timestamp <- bytesToInteger(to.read,4)
  user_id <- bytesToInteger(to.read,8)
  if (type_enum == 0 | type_enum == 1) {
    amount <- readBin(to.read, double(), n=1, size=8, endian="big")
  } else {
    amount <- NA
  }
  return(c(type_enum, timestamp, user_id, amount))
}

# Read the file and create a data frame with the information

to.read = file("byte-reader/data.dat", "rb")
processHeader()
df <- data.frame(matrix(ncol = 4, nrow = 0))
while (length(r <- processRecord()) > 1) {
  df <- rbind(df, r)
}
colnames(df) <- c("type_enum", "timestamp", "user_id", "amount")
close(to.read)

# Answer the Questions

debit_records <- df[df$type_enum==0,]
sum_of_debits <- sum(debit_records$amount)

credit_records <- df[df$type_enum==1,]
sum_of_credits <- sum(credit_records$amount)

start_autopay_records <- df[df$type_enum==2,]
count_start_autopay <- nrow(start_autopay_records)

end_autopay_records <- df[df$type_enum==3,]
count_end_autopay <- nrow(end_autopay_records)

user_records <- df[df$user_id==2456938384156277127,]
sum_of_user_credits <- sum(user_records[user_records$type_enum==1,]$amount)
sum_of_user_debits <- sum(user_records[user_records$type_enum==0,]$amount)
user_balance <- sum_of_user_credits - sum_of_user_debits
```

Areas for further consideration and refinement

There are many areas that could be addressed next and prioritizing any one of these would be an exercise in quantifying trade-offs. I advise estimating the work with the engineering team and collaborating with the business/product team to determine which items represent the greatest value to the business.

Logic considerations

- Is there a need to address the discrepancy in the record count in the header 71 and the actual record

count found? I choose to read all 72 records but perhaps I should have ignored the last one. This may be gap in documentation of the log specification.
- Consider where fractional pennies make sense, if any. Round at the appropriate time.

Code quality

- Potentially consolidate the similarities in the `retrieveNbytes` and `bytesToInteger` functions, similar to Ruby's `unpack` function.
- Create constants for the 4 record types so that the code is easier to read.
- Reduce the use of global variables. Make the functions more atomic.
- Build in error handling for the case where the data file or records are malformed.
- Build tests that demonstrate and validate the functions are working properly.
- There might be value in utilizing Docker or Vagrant to wrap this code in a consistent environment.

Future-proofing

- Process the UNIX timestamp data into a more usable R time object.
- Depending on the size of the production data set, refactor this code for performance. For example: this code will run out of memory for sufficiently large data sets.