Exercise1

Uniform currencies

In this exercise and throughout this chapter, you will be working with a retail banking dataset stored in the banking DataFrame. The dataset contains data on the amount of money stored in accounts (acct_amount), their currency (acct_cur), amount invested (inv_amount), account opening date (account_opened), and last transaction date (last_transaction) that were consolidated from American and European branches.

You are tasked with understanding the average account size and how investments vary by the size of account, however in order to produce this analysis accurately, you first need to unify the currency amount into dollars. The pandas package has been imported as pd, and the banking DataFrame is in your environment.

Instructions

- Find the rows of acct_cur in banking that are equal to 'euro' and store them in the variable acct_eu.
- Find all the rows of acct_amount in banking that fit the acct_eu condition, and convert them to USD by multiplying them with 1.1.
- Find all the rows of acct_cur in banking that fit the acct_eu condition, set them to 'dollar'.

In [ ]:
```python
# Find values of acct_cur that are equal to 'euro'
acct_eu = banking['____'] == '____'

# Convert acct_amount where it is in euro to dollars
banking.loc[____, '____'] = banking.loc[____, '____'] * ____

# Unify acct_cur column by changing 'euro' values to 'dollar'
banking.loc[____, '____'] = ____

# Assert that only dollar currency remains
assert banking['acct_cur'].unique() == 'dollar'

#_____________________#
#Solutions

# Find values of acct_cur that are equal to 'euro'
acct_eu = banking['acct_cur'] == 'euro'

# Convert acct_amount where it is in euro to dollars
banking.loc[acct_eu, 'acct_amount'] = banking.loc[acct_eu, 'acct_amount'] * 1.1

# Unify acct_cur column by changing 'euro' values to 'dollar'
banking.loc[acct_eu, 'acct_cur'] = 'dollar'

# Assert that only dollar currency remains
assert banking['acct_cur'].unique() == 'dollar'
```

Exercise2

Uniform dates

After having unified the currencies of your different account amounts, you want to add a temporal dimension to your analysis and see how customers have been investing their money given the size of their account over each year. The account_opened column represents when customers opened their accounts and is a good proxy for segmenting customer activity and investment over time.

However, since this data was consolidated from multiple sources, you need to make sure that all dates are of the same format. You will do so by converting this column into a datetime object, while making sure that the format is inferred and potentially incorrect formats are set to missing. The banking DataFrame is in your environment and pandas was imported as pd.

Instructions

- Print the header of account_opened from the banking DataFrame and take a look at the different results.
- Convert the account_opened column to datetime, while making sure the date format is inferred and that erroneous formats that raise error return a missing value.
- Extract the year from the amended account_opened column and assign it to the acct_year column.
- Print the newly created acct_year column.

In [ ]:
```
# Print the header of account_opened
print(____)

# Print the header of account_opened
print(banking['account_opened'].head())

# Convert account_opened to datetime
banking['account_opened'] = pd.to_datetime(____,
                                           # Infer datetime format
                                           infer_datetime_format = ____,
                                           # Return missing value for error
                                           errors = ____)

#_____________________#
#Solutions

# Print the header of account_opend
print(banking['account_opened'].head())

# Convert account_opened to datetime
banking['account_opened'] = pd.to_datetime(banking['account_opened'],
                                           # Infer datetime format
                                           infer_datetime_format = True,
                                           # Return missing value for error
                                           errors = 'coerce')

# Get year of account opened
banking['acct_year'] = banking['account_opened'].dt.strftime('%Y')

# Print acct_year
print(banking['acct_year'])
```

Exercise3

How's our data integrity?

New data has been merged into the banking DataFrame that contains details on how investments in the inv_amount column are allocated across four different funds A, B, C and D.

Furthermore, the age and birthdays of customers are now stored in the age and birth_date columns respectively.

You want to understand how customers of different age groups invest. However, you want to first make sure the data you're analyzing is correct. You will do so by cross field checking values of inv_amount and age against the amount invested in different funds and customers' birthdays. Both pandas and datetime have been imported as pd and dt respectively.

Instructions

- Find the rows where the sum of all rows of the fund_columns in banking are equal to the inv_amount column.
- Store the values of banking with consistent inv_amount in consistent_inv, and those with inconsistent ones in inconsistent_inv.
- Store today's date into today, and manually calculate customers' ages and store them in ages_manual.
- Find all rows of banking where the age column is equal to ages_manual and then filter banking into consistent_ages and inconsistent_ages.

```
In [ ]: # Store fund columns to sum against
        fund_columns = ['fund_A', 'fund_B', 'fund_C', 'fund_D']

        # Find rows where fund_columns row sum == inv_amount
        inv_equ = banking[____].____(____) == ____

        # Store consistent and inconsistent data
        consistent_inv = ____[____]
        inconsistent_inv = ____[____]

        # Store consistent and inconsistent data
        print("Number of inconsistent investments: ", inconsistent_inv.shape[0])

        # Store today's date and find ages
        today = ____
        ages_manual = today.year - ____.____.____

        # Find rows where age column == ages_manual
        age_equ = ____ == ____

        # Store consistent and inconsistent data
        consistent_ages = ____
        inconsistent_ages = ____

        # Store consistent and inconsistent data
        print("Number of inconsistent ages: ", inconsistent_ages.shape[0])

        #_____________________#
        #Solutions

        # Store fund columns to sum against
        fund_columns = ['fund_A', 'fund_B', 'fund_C', 'fund_D']

        # Find rows where fund_columns row sum == inv_amount
        inv_equ = banking[fund_columns].sum(axis = 1) == banking['inv_amount']

        # Store consistent and inconsistent data
        consistent_inv = banking[inv_equ]
        inconsistent_inv = banking[~inv_equ]

        # Store consistent and inconsistent data
        print("Number of inconsistent investments: ", inconsistent_inv.shape[0])

        # Store today's date and find ages
        today = dt.date.today()
        ages_manual = today.year - banking['birth_date'].dt.year

        # Find rows where age column == ages_manual
        age_equ = banking['age'] == ages_manual

        # Store consistent and inconsistent data
        consistent_ages = banking[age_equ]
        inconsistent_ages = banking[~age_equ]

        # Store consistent and inconsistent data
        print("Number of inconsistent ages: ", inconsistent_ages.shape[0])
```

Exercise4

Missing investors

Dealing with missing data is one of the most common tasks in data science. There are a variety of types of missingness, as well as a variety of types of solutions to missing data.

You just received a new version of the banking DataFrame containing data on the amount held and invested for new and existing customers. However, there are rows with missing inv_amount values.

You know for a fact that most customers below 25 do not have investment accounts yet, and suspect it could be driving the missingness. The pandas, missingno and matplotlib.pyplot packages have been imported as pd, msno and plt respectively. The banking DataFrame is in your environment.

Instructions

- Print the number of missing values by column in the banking DataFrame.
- Plot and show the missingness matrix of banking with the msno.matrix() function.
- Isolate the values of banking missing values of inv_amount into missing_investors and with non-missing inv_amount values into investors.
- Sort the banking DataFrame by the age column and plot the missingness matrix of banking_sorted.

In [ ]:
```
# Print number of missing values in banking
print(____)

# Visualize missingness matrix
____
____

# Isolate missing and non missing values of inv_amount
missing_investors = ____
investors = ____

# Sort banking by age and visualize
banking_sorted = ____
____
plt.show()

#_____________________#
#Solutions

# Print number of missing values in banking
print(banking.isna().sum())

# Visualize missingness matrix
msno.matrix(banking)
plt.show()

# Isolate missing and non missing values of inv_amount
missing_investors = banking[banking['inv_amount'].isna()]
investors = banking[~banking['inv_amount'].isna()]

# Sort banking by age and visualize
banking_sorted = banking.sort_values(by = 'age')
msno.matrix(banking_sorted)
plt.show()
```

Exercise5

Follow the money

In this exercise, you're working with another version of the banking DataFrame that contains missing values for both the cust_id column and the acct_amount column.

You want to produce analysis on how many unique customers the bank has, the average amount held by customers and more. You know that rows with missing cust_id don't really help you, and that on average acct_amount is usually 5 times the amount of inv_amount.

In this exercise, you will drop rows of banking with missing cust_ids, and impute missing values of acct_amount with some domain knowledge.

Instructions

- Use .dropna() to drop missing values of the cust_id column in banking and store the results in banking_fullid.
- Use inv_amount to compute the estimated account amounts for banking_fullid by setting the amounts equal to inv_amount * 5, and assign the results to acct_imp.
- Impute the missing values of acct_amount in banking_fullid with the newly created acct_imp using .fillna().

In [ ]:
```
# Drop missing values of cust_id
banking_fullid = banking.____(subset = ['____'])

# Compute estimated acct_amount
acct_imp = ____

# Impute missing acct_amount with corresponding acct_imp
banking_imputed = banking_fullid.____({'____':____})

# Print number of missing values
print(banking_imputed.isna().sum())


#_____________________#
#Solutions

# Drop missing values of cust_id
banking_fullid = banking.dropna(subset = ['cust_id'])

# Compute estimated acct_amount
acct_imp = banking_fullid['inv_amount'] * 5

# Impute missing acct_amount with corresponding acct_imp
banking_imputed = banking_fullid.fillna({'acct_amount':acct_imp})

# Print number of missing values
print(banking_imputed.isna().sum())
```