Data Analytics - Coursework 1

Oliwier Kulczycki

ASSESSMENT DECLARATION COVER SHEET

Please complete this cover sheet for each assessment submission. For group assessments, each group member must individually complete and submit a cover sheet.

STUDENT REGISTRATION NUMBER: 40663212

MODULE TITLE: Data Analytics

DATE OF SUBMISSION: 31/10/2025

I declare that, except where explicitly acknowledged*****, this assessment is my own work and has not been submitted for any other module or degree programme at Edinburgh Napier University or any other institution. This declaration is made in compliance with Edinburgh Napier University's Academic Integrity Regulations.

***IMPORTANT**: Contributions from other sources include incorrectly cited quotations or content generated by Generative Artificial Intelligence (Gen AI) tools, such as ChatGPT. See Artificial Intelligence Tools and Your Learning for more information.

Note: It is also important to check the associated Assessment Brief to understand what is permissible. Unless stated in the Assessment Brief, use of Grammarly for checking spelling and grammar is allowable in this assessment, but it cannot be used to generate content.

Please declare here your use of such tools in this assessment submission:
(Select one of the two options below)

□	Yes, I have used Gen AI tools for this submission, and I have described how below.
x	No, I have not used Gen AI tools for this submission.

Table of Contents

1 Introduction

In this coursework, I aim on tackling data cleaning, conversion, visualisation and analysis on the dataset provided – credits. This dataset contains many errors and mistakes, which are first alleviated. I will be using OpenRefine for cleaning, Weka for help with, data format conversions. Deeper conversions, visualisation and analysis will be done with python – (Jupyter notebook). This will allow me to more easily display the code as well as have proper annotations and run it in a more controlled manner. Natively, python is unable to do these levels of data analysis which is why I'll be utilising pre-written libraries: Pandas and NumPy for data wrangling, matplotlib for data visualisation and SciPy for scientific analysis.

2 Data Preparation

Before starting data cleaning with OpenRefine, I first added attribute names which were missing from the original. I done this by opening the dataset in excel and inserted a row above all the data. I used the attribute names described in the coursework specification.

2.1 Data Cleaning

Data cleaning is a process involving basic level analysis, with the support of tools, to view and correct any mistakes with the format of the data within the dataset. This could include correcting; obvious mistakes, data corruption, typos, language/writing differences, etc. This makes sure all data adheres to a certain format and that when later analysed, isn't subject to skewed results. A common example could be "Dataset originally takes monetary values as full pounds – data was accidentally inputted as pennies, (1000 instead of 10), for some fields."

Additionally, under the *'personal_status'* attribute, a possible category of 'female single' is mentioned by the coursework specification, however no such category appears in the dataset. I believe this is an important note as my graphs omit the empty x-tick instead of plotting 0.

2.1.1 Quotation marks

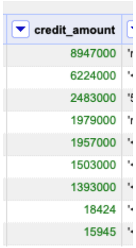
Because of issues, which are later encountered during the data processing section, I decided it would be best to remove all quotation marks – ", from the dataset. This included essentially every string's first and last character. I did this by going through each facet for each attribute and editing all instances.

Facets

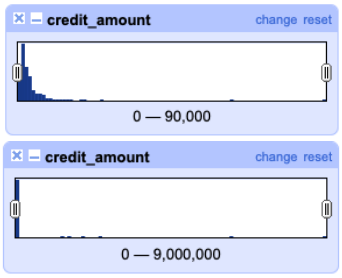
When performing operations on a dataset with OpenRefine, one of the most powerful tools at disposal are 'Facets'. This tool allows for easy viewing of unique entries in fields when using the 'Text Faucet' tool, as well as easy viewing of numeric ranges with the 'Numeric Faucet' tool.

2.1.2 credit_amount

The *credit_amount* attribute contained exceedingly large values, far outside the of anything else within the range. The 7 *out-of-range* entries all seemed to contain a lot of 0's at the end of them. Under the assumption that these were inserted as pennies, instead of full pounds, I removed the trailing two 0's for each of the 7 highest entries. This left the dataset with a much more believable amount of credit.

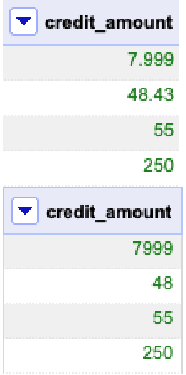


credit_amount
8947000
6224000
2483000
1979000
1957000
1503000
1393000
18424
15945



(before/after)

Additionally, on the low side of the spectrum, present were values with decimal points. Wanting to follow the precedent set by already existing data, I opted to remove them. I tried to follow number-writing conventions and convert these into real numbers accordingly. Because *'7.999'* is written similarly to how a visual showing of *thousand* - '7,999' for ease of readability, I decided to change this to 7999.
Unlike this however, 48.43 was rounded down to 48 as it more closely resembles a representation of pennies rather than a style of writing.



credit_amount
7.999
48.43
55
250

credit_amount
7999
48
55
250

2.1.3 class

The *class* attribute featured stand-out entries such as *'1'* and *'0'*. I deferred to using the **computing** approach and assuming that *'0'* means *'bad'*, *'1'* means *'good'*.

2.1.4 purpose

The *purpose* attribute required the most work. It features many mistakes: **typos**, **shortenings**, **format**, etc. I fixed the **typos** for; *business, education, furniture/equipment, radio/tv, used car*. These were all obvious typos/shortenings and didn't require much thought before editing. I followed through by cross-referencing the brief to see which values should be set as the 'correct' ones.

2.1.5 job

The *job* attribute had some incorrect entries such as the *'good'* and *'poor'* entries. I designated these to 'skilled' and 'unskilled resident' respectively. I also cleaned up and made the quotation marks consistent. *(Although this is addressed anyway).*

2.1.6 age

Some fields had the age set to exceptional values such as *1* – (Figure 3). I assumed these were misinputs and followed conventions seen in the data and assumed these were meant to be *19*, as seen, the two fields below share almost all other values except age.

2.2 Data Transformations and Conversions

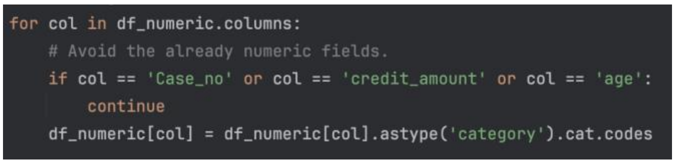
Transformations are done on the data by clicking on the field title, **Edit Cells -> Common transforms -> To text/number**. This is done to establish what type of data that field is. This is important for later data processing so that numeric values aren't treated as nominal values – (strings).

'To number' transformations are done on fields; *Case_no, credit_amount, age.* All the rest of the fields are also forcefully 'To text' transformed, just to be sure they are in the right format.

2.2.1 Numeric Conversion

I decided to use python for the numeric conversion.

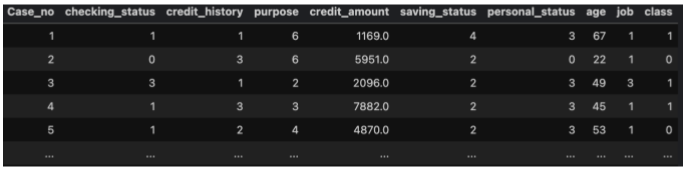
For this conversion, I took advantage of Pandas built-in Categorical Data (Pandas Documentation, 2025). This dataframe type only allows for a fixed number of possible values, which is perfect for this use case of essentially enumerating the values within the dataset.



```
for col in df_numeric.columns:
    # Avoid the already numeric fields.
    if col == 'Case_no' or col == 'credit_amount' or col == 'age':
        continue
    df_numeric[col] = df_numeric[col].astype('category').cat.codes
```

(**df_numeric** is a direct copy of the entire cleaned dataset extracted from a csv file.)

This code runs through each column in the dataframe, although avoids any previously already numeric fields such as *Case_no, credit_amount* and *age.* The following line casts each column within the dataframe into the dtype of `category`. As mentioned earlier, this is useful as it assigns each unique entry in that column a unique integer value and stores it inside of a temporary series object (Pandas Documentation, 2025). The `.cat.codes()` retrieves these mapping from the series object. The `df_numeric[col] =` at the beginning of the line finally applies this to the real dataframe by matching categories to the previously established codes.



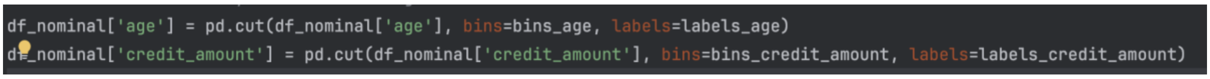
Case_no	checking_status	credit_history	purpose	credit_amount	saving_status	personal_status	age	job	class
1	1	1	6	1169.0	4	3	67	1	1
2	0	3	6	5951.0	2	0	22	1	0
3	3	1	2	2096.0	2	3	49	3	1
4	1	3	3	7882.0	2	3	45	1	1
5	1	2	4	4870.0	2	3	53	1	0
...	...	...	...	...	...	...	...	...	...

The result of this process is a fully mapped copy of the original dataframe, with the previously numeric fields un-coded.

2.2.2 Nominal Conversion

The conversion of the dataframe to nominal was much simpler than the conversion to numeric

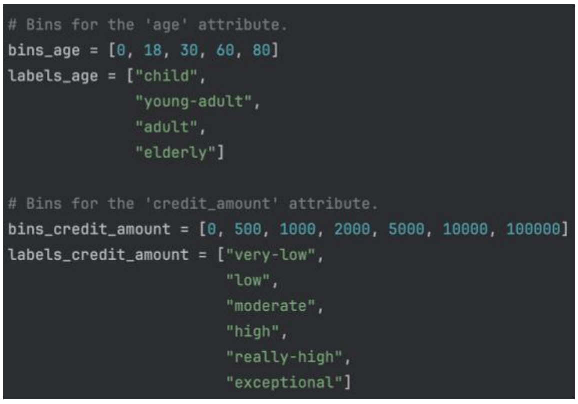
Firstly, I decided that the indexing attribute - 'Case_no', should keep its original values, however they should be cast to a string. This was done using the line `df_nominal['Case_no'] = df['Case_no'].astype(str)`. This is important as the entire point of that attribute is to differentiate each field in case all other attributes end up being the same.



```
df_nominal['age'] = pd.cut(df_nominal['age'], bins=bins_age, labels=labels_age)
df_nominal['credit_amount'] = pd.cut(df_nominal['credit_amount'], bins=bins_credit_amount, labels=labels_credit_amount)
```

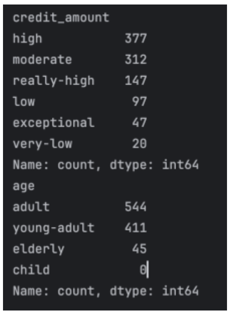
For a nominal conversion to be meaningful, categorisation of different value ranges within the dataset is essential. This can be done with a built-in function in pandas; `pandas.cut()` (Pandas Documentation, 2025).

Before using this function, you must first assign the data with your chosen ranges *(bins),* these will act as cut-off points and in unison with the *labels* will be used to replace values with strings inside the dataframe.
The bins array is self-explanatory for both and decides the ranges of values for the corresponding labels. Each label maps in between two values; for example, 'child' is mapped to in-between *'0'* and *'18'.*



```
# Bins for the 'age' attribute.
bins_age = [0, 18, 30, 60, 80]
labels_age = ["child",
              "young-adult",
              "adult",
              "elderly"]

# Bins for the 'credit_amount' attribute.
bins_credit_amount = [0, 500, 1000, 2000, 5000, 10000, 100000]
labels_credit_amount = ["very-low",
                        "low",
                        "moderate",
                        "high",
                        "really-high",
                        "exceptional"]
```

The age brackets were chosen based on approximate real-life definitions. The brackets for *'credit_amount'* , were much more difficult to decide as the range of values, and their distribution is skewed. I decided to follow the pattern seen in the data, of exponentially increasing values, and translate that to the different bins. A rough pattern of doubling the next value resulted in a relativity evenly split distribution. I believe this also splits the people in the dataset into meaningful wealth distribution buckets.



```
credit_amount
high           377
moderate       312
really-high    147
low             97
exceptional     47
very-low        20
Name: count, dtype: int64
age
adult          544
young-adult    411
elderly         45
child            0
Name: count, dtype: int64
```

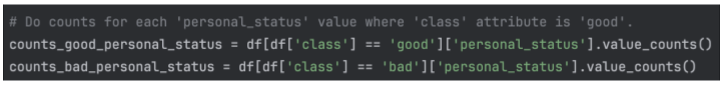
2.3 Data Framework and Visualisation

For the visualisation of the data, I opted for the use of the matplotlib library available in python. This allows me to easily create a figure and then graph values in many ways. For this specific task I chose to go with a twin bar chart, showing two separate bars per x-axis root.

For the purposes of conciseness, I will only be showing the methods used for the chart referencing 'personal_status'. However, the entire process is identical for the chart referencing 'saving_status', with the obvious change being 'personal_status' inside the code is replaced with 'saving_status'.

2.3.1 Filtering Data

Before being able to display any useful information, the correct data format must be gathered. I decided to go with a count of each unique value within the *'personal_status'* attribute, for both *'class'* attribute value options. These are then stored as separate small dataframes which are used to display the two separate bars when plotted.



```
# Do counts for each 'personal_status' value where 'class' attribute is 'good'.
counts_good_personal_status = df[df['class'] == 'good']['personal_status'].value_counts()
counts_bad_personal_status = df[df['class'] == 'bad']['personal_status'].value_counts()
```

The line of code creates a sort of **Boolean mask**, shortening the original dataframe by removing all rows where the *'class'* attribute *isn't* equal to *'good'*. A simple `value.counts()` operation is then performed on this shortened dataframe counting occurrences of each unique entry in the *'personal_status'* attribute. This is all done inside a temporary dataframe, thus not altering the original. This is repeated in the opposite way – removing all rows where *'class'* isn't equal to *'bad'*. The dataframe of gathered data looks like this. (Figure 4).

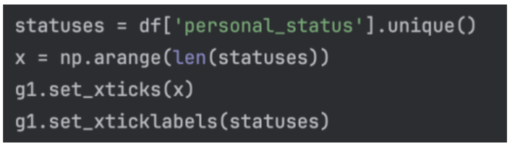
2.3.2 Graphing

2.3.2.1 *Creating the Figure*

Firstly, a figure is creating using `fig_personal_status, g1 = plt.subplots()`, this creates both the full graph - (`fig_personal_status`) and the axes - (`g1`). The axes will let me decorate and populate the graph, using some basic decoration functions such as setting a title, and axis labels.

2.3.2.2 *X-Ticks*

Because the graph features two bars, per tick-label, pre-calculating the root position for these was necessary to calculate the offset per bar to display them side-by-side. This is done simply by using the built-in `numpy.arange()` and `.set_xticks().`



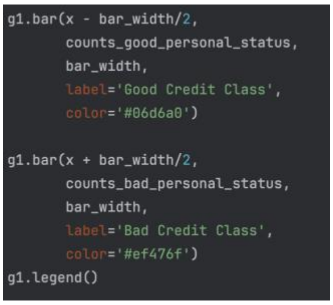
```
statuses = df['personal_status'].unique()
x = np.arange(len(statuses))
g1.set_xticks(x)
g1.set_xticklabels(statuses)
```

`.arange()` simply *"returns evenly spaces values across a specified range"* (NumPy Documentation, 2025), then `.set_xticks()` arranges these for the actual created graph. This creates evenly spaces ticks, which are then labelled using `.set_xticklabels`, which takes the argument `statuses`, which is a list of each unique entry in the *'personal_status'* attribute.

2.3.2.3 *Plotting the Bars*

Final graphs can be seen at (Figure 1) – for the Distribution of Credit Class by Personal Status, and (Figure 2) – for the Distribution of Credit Class by Saving Status.

To plot, I used the built-in bar plotting function. Location is usually represented by x, however, because there will be two bars displayed side-by-side, I need to offset the root of each bar by half the bar's width. The function call comes with a specific label and colour; it also takes the previously discussed dataframe as the data points. `bar_width` is declared earlier in the code as an arbitrary width of each bar.



```
g1.bar(x - bar_width/2,
       counts_good_personal_status,
       bar_width,
       label='Good Credit Class',
       color='#06d6a0')

g1.bar(x + bar_width/2,
       counts_bad_personal_status,
       bar_width,
       label='Bad Credit Class',
       color='#ef476f')
g1.legend()
```

2.4 Scientific Analysis

The method I chose to use for the analysis is the Chi-Squared Goodness of Fit Test. This technique is used to analyse whether a sample group of the data matches the expected data based on the entire population of the dataset. This is a perfect way to measure what is being asked – which values between *personal_status* and *saving_status,* influence the distribution of 'good' vs 'bad' values within the *class* attribute.

2.4.1 Chi-Squared Goodness of Fit Test Calculation

(Following text references Figure 5, among others).
To use the Chi-Squared Test, we must first calculate an expected dataset to compare to.

$$sum(\frac{(observed - expected)^2}{expected})$$

(Chi-Squared Equation)

This relies on first calculating the expected frequency and ratios of categories within the chosen attributes in the original-unaltered dataframe. In this scenario, the 'class' attribute has an inherently imbalanced distribution of around 70% of entries being 'good' and 30% being 'bad'. This will stand as the expected distribution for each category of both *personal_status* and *saving_status* and is represented within the code as a dataframe with values being decimal – (0.7, 0.3). For the calculated results, I opted to use Python's built-in dictionaries. They would provide me with the easiest way to combine the attribute value with the calculated data, as well as keep everything tidy and readable. I create empty dictionaries for both observed - (*labelled as local)* and expected data for both chosen attributes. For each chosen attribute, I then loop through the dataframe for each unique category within, and count occurrences of *class* values for every row in which the category for the looped attribute matches the current iterator (list of all categories for given attribute). I append these counts to the dictionaries created, with the key of the current iterator, and calculate the expected results for that size group by using the previously calculated global ratios. The counts are calculated using the previously seen method of Boolean masking.

For ease of calculating the ChiScore and PValues, I use SciPy's stats. It requires the input of two variables – expected and observed values. I loop through all keys for each chosen attribute's dictionaries and extract the observed and expected values, plugging them into SciPy's `stats.chisquare()`. (Figure 6).

2.4.2 Statistically Significant Categories to Credit Class

To define statistically significant, I decided to go with a Null Hypothesis threshold of 0.05. I believe this value represents a good point at which to define significance due to common practices of using this value as well as manual data inspection.

A simple list is created, containing all significant values and their categories (Figure 8). Most categories within 'personal_status' seem to follow the expected results, except for 'female div/dep/mar', which shows a statistically significant likelihood of deviating from the expected data. This suggests that most of the influence 'personal_status' has on a person's credit class is concentrated in one category.
Unlike that, 'saving_status' categories have a high significance across the board. However, a threshold of 0.05 is too loose at these scales. The single category of 'no known savings' is 6.5 times more significant than the next closest one, and so on. The pattern seen across this attribute suggests that lower savings are associated with a higher likelihood of a negative credit class.

Significant Categories

'female div/dep/mar'
PV = 0.0473

'no known savings'
PV = 0.0002

'<100'
PV = 0.0013

'500<=X<1000'
PV = 0.0298

'>=1000'
PV = 0.0081

Findings in (Figure 7).

3 Appendix

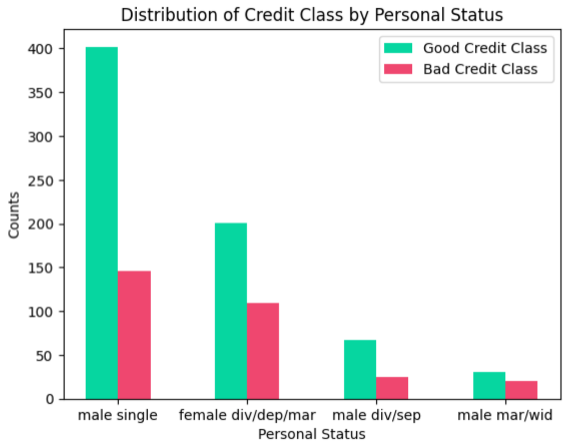


Figure 1

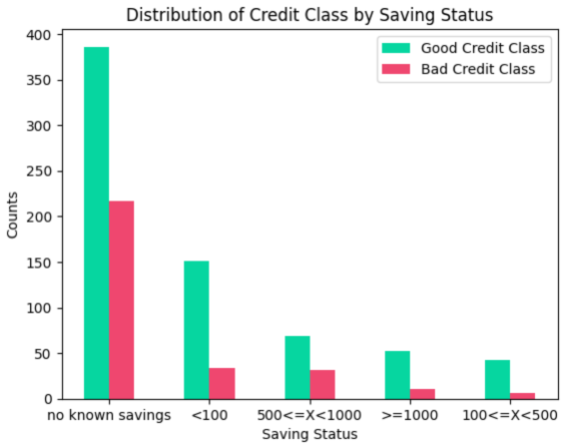
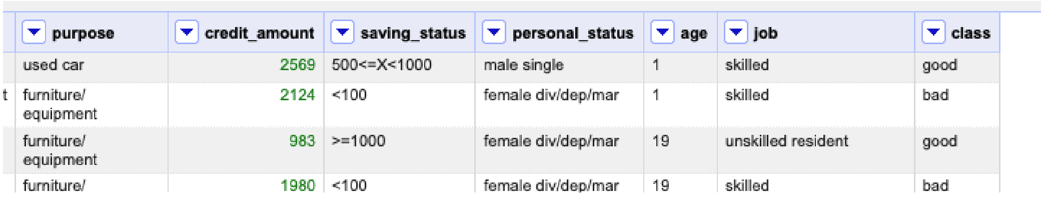
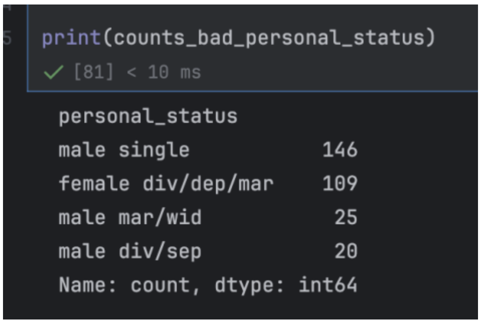


Figure 2



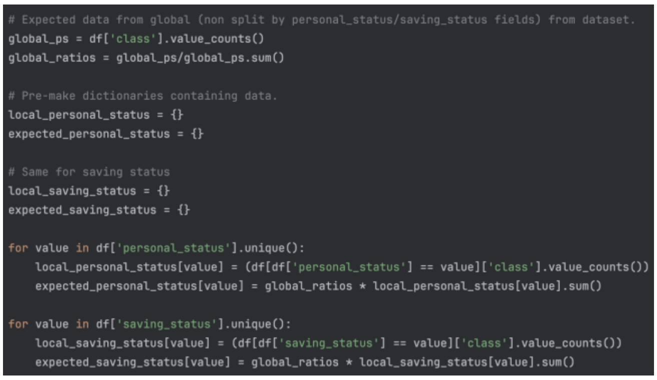
purpose	credit_amount	saving_status	personal_status	age	job	class
used car	2569	500<=X<1000	male single	1	skilled	good
furniture/ equipment	2124	<100	female div/dep/mar	1	skilled	bad
furniture/ equipment	983	>=1000	female div/dep/mar	19	unskilled resident	good
furniture/	1980	<100	female div/dep/mar	19	skilled	bad

Figure 3



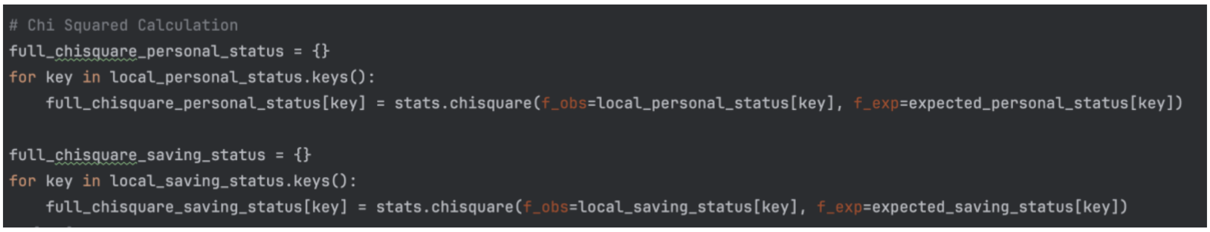
```
print(counts_bad_personal_status)
✓ [81] < 10 ms

personal_status
male single           146
female div/dep/mar    109
male mar/wid           25
male div/sep           20
Name: count, dtype: int64
```

Figure 4



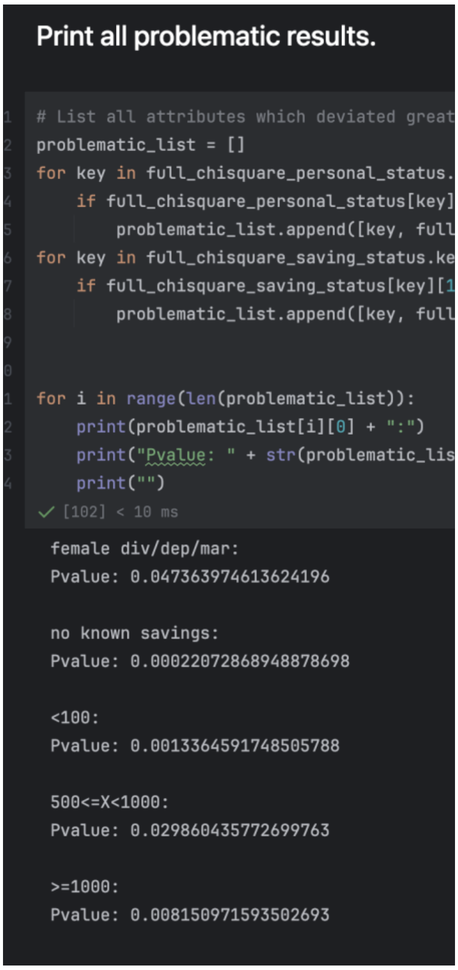
```
# Expected data from global (non split by personal_status/saving_status fields) from dataset.
global_ps = df['class'].value_counts()
global_ratios = global_ps/global_ps.sum()

# Pre-make dictionaries containing data.
local_personal_status = {}
expected_personal_status = {}

# Same for saving status
local_saving_status = {}
expected_saving_status = {}

for value in df['personal_status'].unique():
    local_personal_status[value] = (df[df['personal_status'] == value]['class'].value_counts())
    expected_personal_status[value] = global_ratios * local_personal_status[value].sum()

for value in df['saving_status'].unique():
    local_saving_status[value] = (df[df['saving_status'] == value]['class'].value_counts())
    expected_saving_status[value] = global_ratios * local_saving_status[value].sum()
```

Figure 5



```
# Chi Squared Calculation
full_chisquare_personal_status = {}
for key in local_personal_status.keys():
    full_chisquare_personal_status[key] = stats.chisquare(f_obs=local_personal_status[key], f_exp=expected_personal_status[key])

full_chisquare_saving_status = {}
for key in local_saving_status.keys():
    full_chisquare_saving_status[key] = stats.chisquare(f_obs=local_saving_status[key], f_exp=expected_saving_status[key])
```

Figure 6



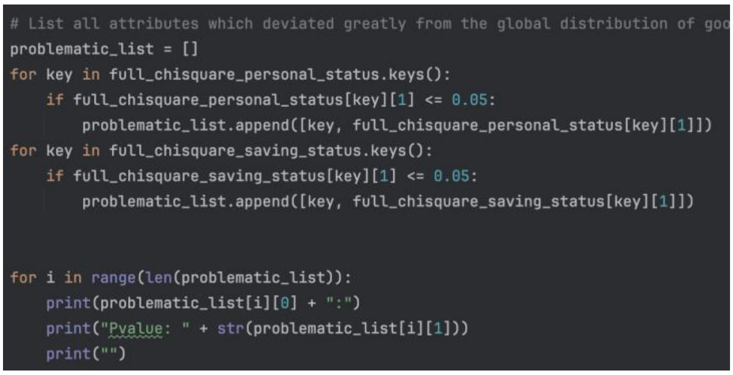
Print all problematic results.

```
# List all attributes which deviated great
problematic_list = []
for key in full_chisquare_personal_status.
    if full_chisquare_personal_status[key]
        problematic_list.append([key, full
for key in full_chisquare_saving_status.ke
    if full_chisquare_saving_status[key][1
        problematic_list.append([key, full


for i in range(len(problematic_list)):
    print(problematic_list[i][0] + ":")
    print("Pvalue: " + str(problematic_lis
    print("")
```

✓ [102] < 10 ms

```
female div/dep/mar:
Pvalue: 0.047363974613624196

no known savings:
Pvalue: 0.00022072868948878698

<100:
Pvalue: 0.0013364591748505788

500<=X<1000:
Pvalue: 0.029860435772699763

>=1000:
Pvalue: 0.008150971593502693
```

Figure 7



```
# List all attributes which deviated greatly from the global distribution of goo
problematic_list = []
for key in full_chisquare_personal_status.keys():
    if full_chisquare_personal_status[key][1] <= 0.05:
        problematic_list.append([key, full_chisquare_personal_status[key][1]])
for key in full_chisquare_saving_status.keys():
    if full_chisquare_saving_status[key][1] <= 0.05:
        problematic_list.append([key, full_chisquare_saving_status[key][1]])


for i in range(len(problematic_list)):
    print(problematic_list[i][0] + ":")
    print("Pvalue: " + str(problematic_list[i][1]))
    print("")
```

Figure 8