A2_DataExploration

February 11, 2019

1 COGS 108 - Assignment 2: Data Exploration

2 Important

You must submit this file (`A2_DataExploration.ipynb`) to TritonED to finish the homework.

This assignment has more than 5 times as many questions as A1! Get started as early as possible.

This assignment has hidden tests: tests that are not visible here, but that will be run on your submitted assignment.

- This means passing all the tests you can see in the notebook here does not guarantee you have the right answer!

Each coding question in this assignment only requires a small amount of code, about 1-3 lines.

- If you find yourself writing much more than that, you might want to reconsider your approach.
- Use the Tutorials notebooks as reference, as they often contain similar examples to those used in the assignment.

In [1]:
```
# Imports
#
# Note: these are all the imports you need for this assignment!
# Do not import any other functions / packages

# Display plots directly in the notebook instead of in a new window
%matplotlib inline

# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
# Configure libraries
# The seaborn library makes plots look nicer
sns.set()
```

```
sns.set_context('talk')

# Don't display too many rows/cols of DataFrames
pd.options.display.max_rows = 7
pd.options.display.max_columns = 8

# Round decimals when displaying DataFrames
pd.set_option('precision', 2)
```

2.1 Part 1 - Data Wrangling

For this assignment, you are provided with two data files that contain information on a sample of people. The two files and their columns are:

- `age_steps.csv`: Contains one row for each person.
  - `id`: Unique identifier for the person.
  - `age`: Age of the person.
  - `steps`: Number of steps the person took on average in January 2018.
- `incomes.json`: Contains one record for each person.
  - `id`: Unique identifier for the person. Two records with the same ID between `age_steps.csv` and `incomes.json` correspond to the same person.
  - `last_name`: Last name of the person.
  - `first_name`: First name of the person.
  - `income`: Income of the person in 2018.

For part 1 and 2 of the assignment, we recommend looking at the official 10 minutes to pandas guide: http://pandas.pydata.org/pandas-docs/stable/10min.html

Question 1a: Load the `age_steps.csv` file into a `pandas` DataFrame named `df_steps`. It should have 11850 rows and 3 columns.

```
In [3]: # YOUR CODE HERE
        df_steps = pd.read_csv("age_steps.csv")
        df_steps

Out[3]:           id  age  steps
        0      37475   46   5951
        1      51201   36  10139
        2      77330   50     -1
        ...      ...  ...    ...
        11847  39197   52   7580
        11848  62557   49   6273
        11849  77950   29   8888

        [11850 rows x 3 columns]
```

```
In [4]: # Tests for 1a

        assert isinstance(df_steps, pd.DataFrame)
        assert df_steps.shape == (11850, 3)
```

Question 1b: Load the `incomes.json` file into a pandas DataFrame called `df_income`. The DataFrame should have 13332 rows and 4 columns.

```
In [5]: # YOUR CODE HERE
        df_income = pd.read_json("incomes.json")
```

```
In [6]: # Tests for 1b

        assert isinstance(df_income, pd.DataFrame)
        assert df_income.shape == (13332, 4)
```

Question 1c: Drop the `first_name` and `last_name` columns from the `df_income` DataFrame. The resulting DataFrame should only have two columns.

```
In [7]: # YOUR CODE HERE
        df_income=df_income.drop(columns=["first_name","last_name"])
```

```
In [8]: # Tests for 1c

        assert 'first_name' not in df_income.columns
        assert 'last_name' not in df_income.columns
```

Question 1d: Merge the `df_steps` and `df_income` DataFrames into a single combined DataFrame called `df`. Use the `id` column to match rows together.

The final DataFrame should have 10664 rows and 4 columns: `id`, `income`, `age`, and `steps`.

Call an appropriate `pandas` method to perform this operation; don't write a `for` loop. (In general, writing a `for` loop for a DataFrame will produce poor results.)

```
In [9]: # YOUR CODE HERE
        df = pd.merge(df_steps,df_income[['id', 'income']],on='id')
        df
```

```
Out[9]:           id  age  steps    income
        0      37475   46   5951  48515.39
        1      51201   36  10139  37688.26
        2      77330   50     -1  37606.16
        ...      ...  ...    ...       ...
        10661  27343   34     -1       NaN
        10662  39197   52   7580  12469.22
        10663  77950   29   8888  69797.18

        [10664 rows x 4 columns]
```

```
In [10]: # Tests for 1d

         assert isinstance(df, pd.DataFrame)
         assert set(df.columns) == set(['id', 'income', 'age', 'steps'])
         assert df.shape == (10664, 4)
```

Question 1e: Reorder the columns of `df` so that they appear in the order: `id`, `age`, `steps`, then `income`.

```
In [11]: # YOUR CODE HERE
         df = df.loc[:, ['id', 'age', 'steps','income']]
```

```
In [12]: # Tests for 1e

         assert list(df.columns) == ['id', 'age', 'steps', 'income']
```

Question 1f: You may have noticed something strange: the merged `df` DataFrame has fewer rows than either of `df_steps` and `df_income`. Why did this happen?

Please select the **one** correct explanation below and save your answer in the variable `q1f_answer`. For example, if you believe choice number 4 explains why `df` has fewer rows, set `q1f_answer = 4`.

1. Some steps were recorded inaccurately in `df_steps`.
2. Some incomes were recorded inaccurately in `df_income`.
3. There are fewer rows in `df_steps` than in `df_income`.
4. There are fewer columns in `df_steps` than in `df_income`.
5. Some `id` values were repeated in `df_steps` and in `df_income`.
6. Some `id` values in either `df_steps` and `df_income` were missing in the other DataFrame.

You may use the cell below to run whatever code you want to check the statements above. Just make sure to set `q1f_answer` once you've selected a choice.

```
In [13]: # YOUR CODE HERE
         q1f_answer = 6
```

```
In [14]: # Tests for 1f

         assert isinstance(q1f_answer, int)
```

2.2 Part 2 - Data Cleaning

Before proceeding with analysis, we need to check our data for missing values.

There are many reasons data might contain missing values. Here are two common ones:

- **Nonresponse.** For example, people might have left a field blank when responding to a survey, or left the entire survey blank.
- **Lost in entry.** Data might have been lost after initial recording. For example, a disk cleanup might accidentally wipe older entries of a database.

In general, it is **not** appropriate to simply drop missing values from the dataset or pretend that if filled in they would not change your results. In 2016, many polls mistakenly predicted that Hillary Clinton would easily win the Presidential election by committing this error.

In this particular dataset, however, the **missing values occur completely at random**. This criteria allows us to drop missing values without significantly affecting our conclusions.

Question 2a: How values are missing in the `income` column of `df`? Save this number into a variable called `n_nan`.

In [15]:
```
# YOUR CODE HERE
n_nan = df["income"].isna().sum()
n_nan
```

Out[15]: 463

In [16]:
```
# Tests for 2a

assert(n_nan)
```

Question 2b: Remove all rows from `df` that have missing values.

In [17]:
```
# Remove all rows from df that have missing data. In other words, remove all rows wit

# YOUR CODE HERE
df = df.dropna(subset=['income'])
df.shape
```

Out[17]: (10201, 4)

In [18]:
```
# Tests for 2b

assert sum(np.isnan(df['income'])) == 0
assert df.shape == (10201, 4)
```

Question 2c: Note that we can now compute the average income. If your `df` variable contains the right values, `df['income'].mean()` should produce the value `25474.07`.

Suppose that we didn't drop the missing incomes. What will running `df['income'].mean()` output? Use the variable `q2c_answer` to record which of the below statements you think is true. As usual, you can use the cell below to run any code you'd like in order to help you answer this question as long as you set `q2c_answer` once you've finished.

1. No change; `df['income'].mean()` will ignore the missing values and output `25474.07`.
2. `df['income'].mean()` will produce an error.
3. `df['income'].mean()` will output `0`.
4. `df['income'].mean()` will output `nan` (not a number).
5. `df['income'].mean()` will fill in the missing values with the average income, then compute the average.
6. `df['income'].mean()` will fill in the missing values with `0`, then compute the average.

In [19]:
```
# YOUR CODE HERE
q2c_answer = 1
```

In [20]:
```
# Tests for 2c

assert isinstance(q2c_answer, int)
```

Question 2d: Suppose that missing incomes did not occur at random, and that individuals with incomes below $10000 a year are less likely to report their incomes. If so, one of the statements is true. Record your choice in the variable `q2d_answer`.

1. `df['income'].mean()` will likely output a value that is larger than the population's average income.
2. `df['income'].mean()` will likely output a value that is smaller than the population's average income.
3. `df['income'].mean()` will likely output a value that is the same as the population's average income
4. `df['income'].mean()` will raise an error.

In [21]:
```
# YOUR CODE HERE
q2d_answer = 1
```

In [22]:
```
# Tests for 2d

assert isinstance(q2d_answer, int)
```

2.3 Part 3: Data Visualization

Although pandas only displays a few rows of a DataFrame at a time, we can use data visualizations to quickly determine the **distributions** of values within our data.

pandas comes with some plotting capabilities built-in. We suggest taking a look at https://pandas.pydata.org/pandas-docs/stable/visualization.html for examples. Here's one example:

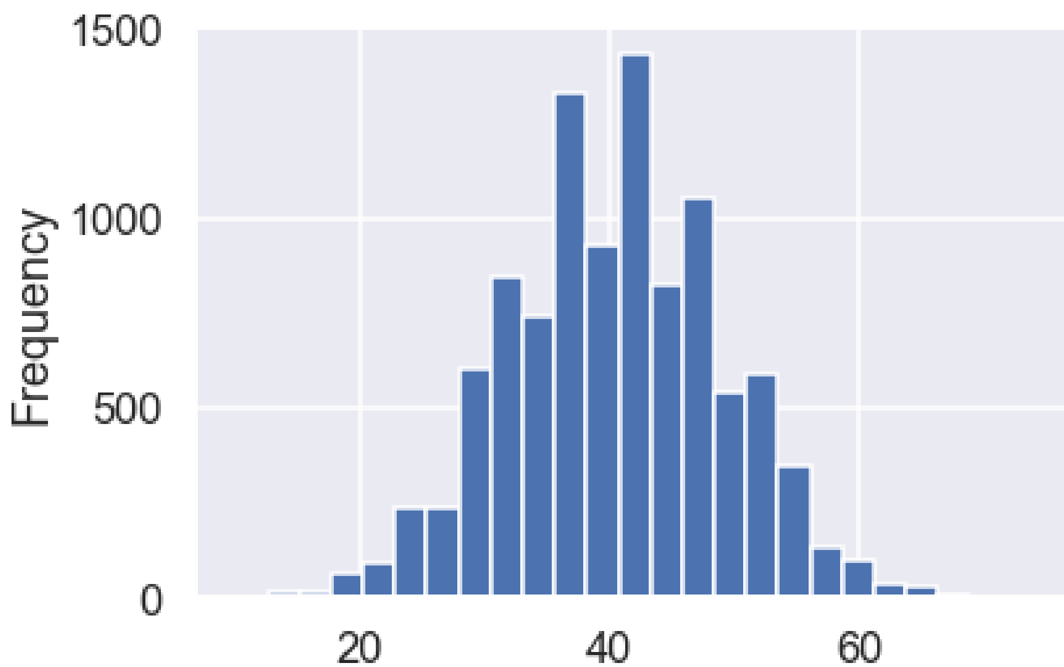
Most plotting libraries in Python are built on top of a library called Matplotlib, including the plotting methods used in pandas. Although you won't need to know Matplotlib for this assignment, you will likely have to use it in future assignments and your final project, so keep the library in mind.

Notes:

- Everywhere that we ask you to create a plot, make sure to leave the `plt.gcf()` line at the end of the cell. Otherwise, you will lose points in the autograder.
- For all your histograms, use **25 bins**.

Question 3a: Plot a histogram of the age column with 25 bins.

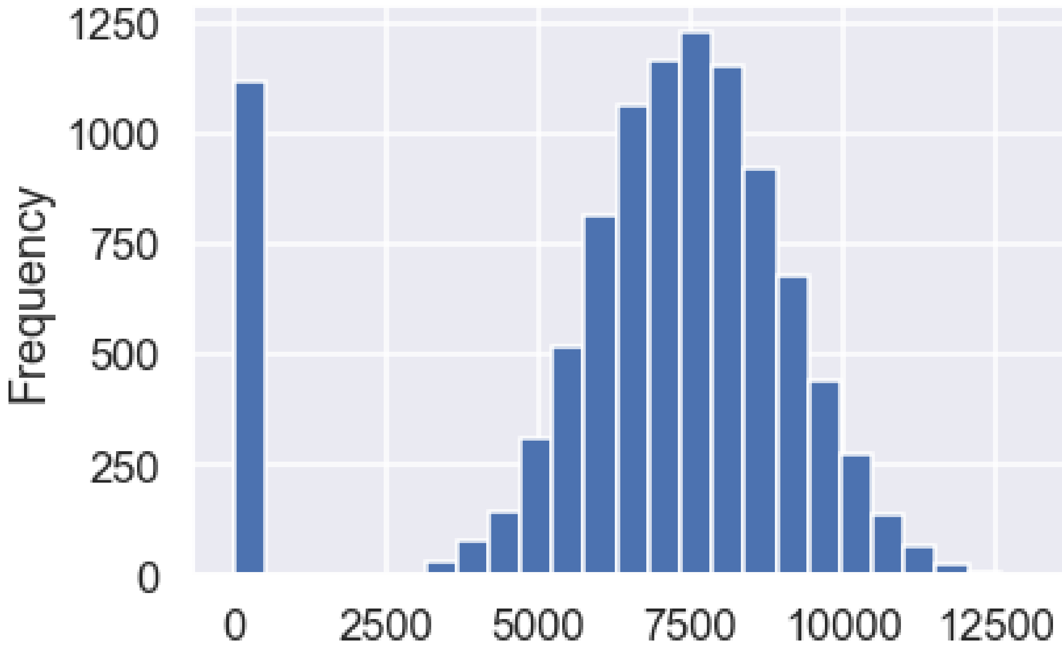
In [23]:
```
# YOUR CODE HERE
df['age'].plot.hist(stacked=True,bins=25)

f1 = plt.gcf()
```




```
In [24]: # Tests for 3a

         assert f1.gca().has_data()

         # If you fail this test, you didn't use 25 bins for your histogram.
         assert len(f1.gca().patches) == 25
```

Question 3b: Plot a histogram of the steps column with 25 bins.

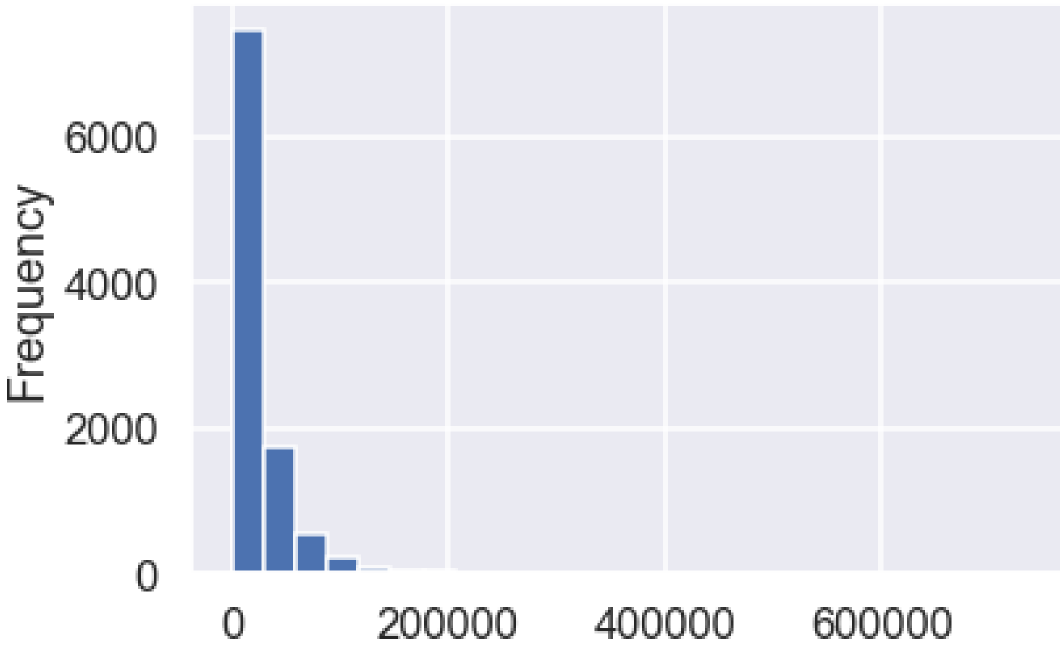
```
In [25]: # YOUR CODE HERE
         df['steps'].plot.hist(stacked=True,bins=25)

         f2 = plt.gcf()
```




```
In [26]: # Tests for 3b

         assert f2.gca().has_data()
```

Question 3c: Plot a histogram of the `income` column with 25 bins.

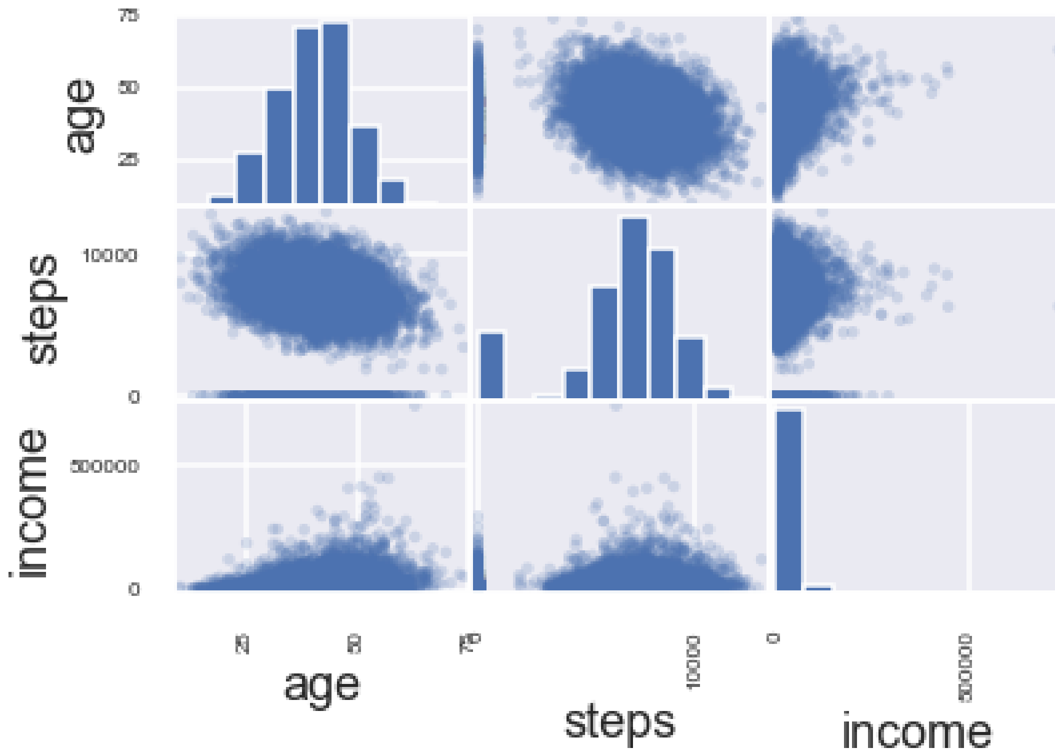
```
In [27]: # YOUR CODE HERE
         df['income'].plot.hist(stacked=True,bins=25)

         f3 = plt.gcf()
```




```
In [28]: # Tests for 3c

         assert f3.gca().has_data()
```

Question 3d: Plot the data using the pandas `scatter_matrix` function. Only plot the `age`, `steps`, and `income` columns.

```
In [29]: # YOUR CODE HERE
         from pandas.plotting import scatter_matrix
         scatter_matrix(df[['age','steps','income']],alpha=0.2)

         f4 = plt.gcf()
```




```
In [30]: # Tests for 3d

         assert f4.gca().has_data()
```

2.4 Part 4: Data Pre-Processing

In the above sections, we performed basic data cleaning and visualization.

In practice, these two components of an analysis pipeline are often combined into an iterative approach. We go back and forth between looking at the data, checking for issues, and cleaning the data.

Let's continue with an iterative procedure of data cleaning and visualization, addressing some issues that we notice after visualizing the data.

Question 4a: In the visualization of the `steps` column, we notice a large number of `-1` values. Count how many rows in `df` have `-1` in their `steps` column. Store the result in the variable `n_neg`.

```
In [32]: # YOUR CODE HERE
         n_neg = len(df[df['steps']<0])
         n_neg

Out[32]: 1116
```

```
In [33]: # Tests for 4a
```

```
assert(n_neg)
assert n_neg > 100
```

Question 4b: Since it's impossible to walk a negative number of steps, we will treat the negative values as missing data. Drop the rows with negative steps from `df`. Your answer should modify `df` itself.

```
In [35]: # YOUR CODE HERE
         df = df[df["steps"]>=0]
         df
```

```
Out[35]:            id  age  steps    income
         0       37475   46   5951  48515.39
         1       51201   36  10139  37688.26
         3       85906   35   6351  20277.05
         ...       ...  ...    ...       ...
         10660    4745   42   7455  33479.82
         10662   39197   52   7580  12469.22
         10663   77950   29   8888  69797.18

         [9085 rows x 4 columns]
```

```
In [36]: # Tests for 4b

         assert sum(df['steps'] == -1) == 0
```

You may have noticed that the values in `income` are not normally distributed which can hurt prediction ability in some scenarios. To address this, we will perform a log transformation on the `income` values.

First though, we will have to deal with any income values that are 0. Note that these values are not impossible values — they may, for example, represent people who are unemployed.

Question 4c: Add a new column to `df` called `income10`. It should contain the same values as `income` with all `0` values replaced with `1`.

```
In [38]: # YOUR CODE HERE
         df['income10'] = df['income'].replace(0,1)
         df.shape
```

```
Out[38]: (9085, 5)
```

```
In [39]: # Tests for 4c

         assert list(df.columns) == ['id', 'age', 'steps', 'income', 'income10']
         assert not any(df['income10'] == 0)
```

Question 4d: Now, transform the `income10` column using a log-base-10 transform. That is, replace each value in `income10` with the $ log_{10} $ of that value.

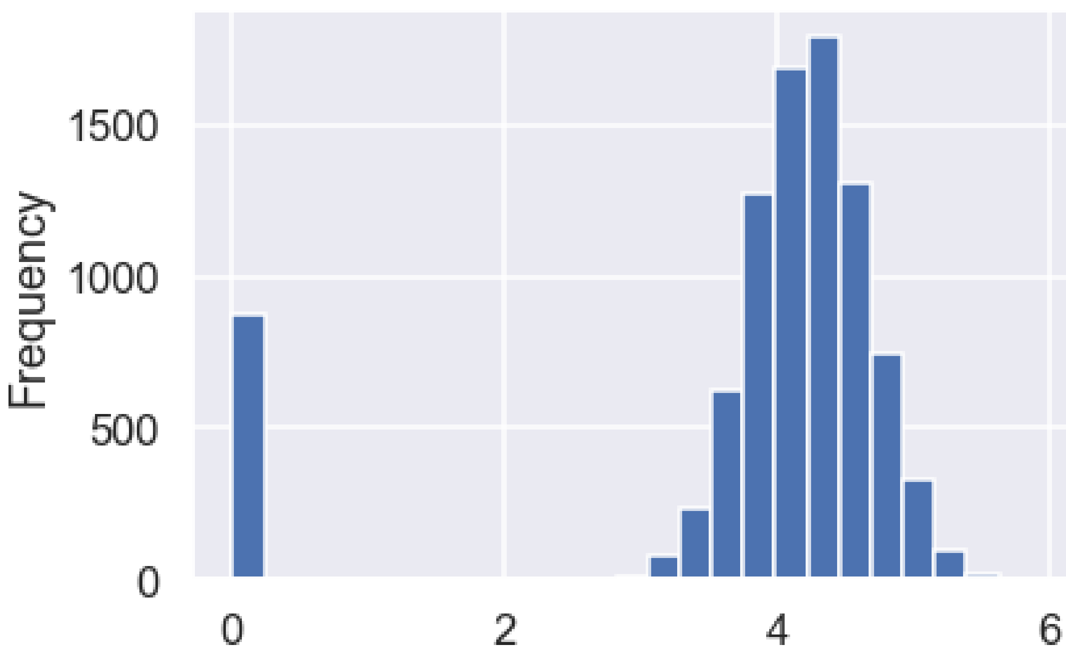
```
In [40]: # YOUR CODE HERE
         df['income10'] = df['income10'].apply(np.log10)
         df
```

```
Out[40]:           id  age  steps    income  income10
         0      37475   46   5951  48515.39      4.69
         1      51201   36  10139  37688.26      4.58
         3      85906   35   6351  20277.05      4.31
         ...      ...  ...    ...       ...       ...
         10660   4745   42   7455  33479.82      4.52
         10662  39197   52   7580  12469.22      4.10
         10663  77950   29   8888  69797.18      4.84

         [9085 rows x 5 columns]
```

```
In [41]: # Tests for 4d

         assert np.isclose(min(df['income10']), 0.0, 0.001)
         assert np.isclose(max(df['income10']), 5.867, 0.001)
```

Question 4e: Now, make a histogram for `income10` data after the data transformation. Again, use 25 bins.

```
In [42]: # YOUR CODE HERE
         df['income10'].plot.hist(stacked=True,bins=25)

         f4 = plt.gcf()
```




```
In [43]: # Tests for 4e
```

```
assert f4.gca().has_data()

# If you fail this test, you didn't use 25 bins for your histogram.
assert len(f4.gca().patches) == 25
```

Question 4f: We might also have certain regulations or restrictions that we need to follow about the data. Here, we will only analyze adults. Remove all rows from `df` where age is less than 18.

In [45]:
```
# YOUR CODE HERE
df = df[df["age"]>=18]
df
```

Out[45]:
```
          id  age  steps    income  income10
0      37475   46   5951  48515.39      4.69
1      51201   36  10139  37688.26      4.58
3      85906   35   6351  20277.05      4.31
...      ...  ...    ...       ...       ...
10660   4745   42   7455  33479.82      4.52
10662  39197   52   7580  12469.22      4.10
10663  77950   29   8888  69797.18      4.84

[9053 rows x 5 columns]
```

In [46]:
```
# Tests for 4f

assert min(df['age']) >= 18
```

2.5 Part 5 - Basic Analyses

Now that we have wrangled and cleaned our data, we can start doing some simple analyses.

Here we will explore some basic descriptive summaries of our data, look into the inter-relations (correlations) between variables, and ask some simple questions about potentially interesting subsets of our data.

Question 5a: Use the `describe` `pandas` method to check a descriptive summary of the data. Save the DataFrame generated by `describe` to a new variable called `desc`.

In [47]:
```
# YOUR CODE HERE
desc = df.describe()
```

In [48]:
```
# Tests for 5a

assert isinstance(desc, pd.DataFrame)
```

Question 5b: Calculate the pairwise correlations between all variables.

Note: do this with a `pandas` method. Keep all columns (including ID). Assign the result (which should be a DataFrame) to a variable called `corrs`.

In [49]:
```
# YOUR CODE HERE
corrs = df.corr()
corrs
```

```
Out[49]:                 id       age     steps  income  income10
         id        1.00e+00 -4.54e-03  1.17e-03   -0.02 -8.22e-03
         age      -4.54e-03  1.00e+00 -2.82e-01    0.27  1.03e-01
         steps     1.17e-03 -2.82e-01  1.00e+00    0.05  2.49e-02
         income   -2.44e-02  2.73e-01  4.74e-02    1.00  4.69e-01
         income10 -8.22e-03  1.03e-01  2.49e-02    0.47  1.00e+00
```

```
In [50]: # Tests for 5b

         assert isinstance(corrs, pd.DataFrame)
         assert corrs.shape == (5, 5)
```

Question 5c: Answer the following questions by setting your answer variables to either `'age'`, `'steps'`, or `'income'`.

- Which variable is most correlated with age (aside from `age` itself)? Record your answer in a variable called `age_corr`.
- Which variable is most correlated with income (aside from `income` and `income10`)? Record your answer in a variable called `inc_corr`.

```
In [1]: # YOUR CODE HERE
        age_corr = "steps"
        inc_corr = "age"
```

```
In [2]: # Tests for 5c

        assert age_corr
        assert inc_corr
        assert age_corr in {'steps', 'age', 'income'}
        assert inc_corr in {'steps', 'age', 'income'}
```

Question 5d: How many steps would you have to walk to be in the top 10% of walkers? Save your answer as a variable called `top_walker`.

Hint: check out the `quantile` method.

```
In [53]: # YOUR CODE HERE
         top_walker = df['steps'].quantile(.9)
         top_walker

Out[53]: 9478.0
```

```
In [54]: # Tests for 5d

         assert top_walker
         assert isinstance(top_walker, float)
```

Question 5e: What is the average income for people over the age of 45? Save your response in a variable called `old_income`.

Note: We're asking for the actual income, not the log-10 of income.

```
In [60]: # YOUR CODE HERE
         old_income = np.mean(df[df['age']>45]['income'])
         old_income

Out[60]: 37705.411710110595
```

```
In [61]: # Tests for 5e
         assert old_income
         assert old_income > 5
```

3 Part 6 - Predictions

A frequent goal of data analysis is to understand so that we can make predictions about future or unseen data points.

Here we will explore some basic predictions, looking into whether we might be able to predict income from our other variables.

Note: You will use the `np.polyfit` function from NumPy as we did in Tutorials/02-DataAnalysis.

Question 6a: Use polyfit to fit a 1-degree linear model, predicting income from age. Call the output parameters `a1` and `b1`.

```
In [62]: # YOUR CODE HERE
         a1, b1 = np.polyfit(df['age'], df['income'], 1)
         a1,b1

Out[62]: (1074.1595347586626, -18039.201842828246)
```

```
In [63]: # Tests for 6a

         assert(a1)
         assert(b1)

         # If you fail these tests, your parameter values are quite far from what they
         # should be.
         assert abs(a1) > 100
         assert abs(b1) > 100
```

Question 6b: Use the model parameters from 6a to predict the income of a 75-year-old. Call your prediction `pred_75`.

```
In [64]: # YOUR CODE HERE
         new_age = 75
         pred_75 = a1 * new_age + b1
         pred_75

Out[64]: 62522.76326407146
```

```
In [65]: # Tests for 6b

         assert(pred_75)
```

Question 6c: Use polyfit once more to fit a 1-degree linear model, predicting income from steps. Call the output parameters a2 and b2.

```
In [66]: # YOUR CODE HERE
         a2, b2 = np.polyfit(df['steps'], df['income'], 1)
         a2,b2
```

```
Out[66]: (1.0500308255474387, 17669.11148396506)
```

```
In [67]: # Tests for 6c

         assert(a2)
         assert(b2)

         # If you fail these tests, your parameter values are quite far from what they
         # should be.
         assert abs(a2) < 100
         assert abs(b2) > 100
```

Question 6d: Predict the income of someone who took 10,000 steps. Call your prediction pred_10k.

```
In [68]: # YOUR CODE HERE
         new_steps = 10000
         pred_10k = a2 * new_steps + b2
         pred_10k
```

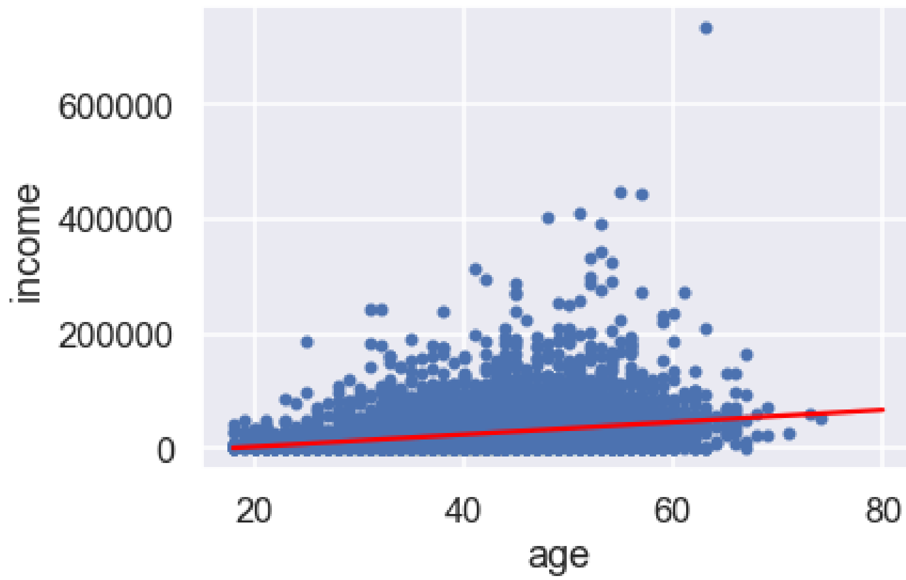
```
Out[68]: 28169.41973943945
```

```
In [69]: # Test for 6d

         assert(pred_10k)
```

Question 6e: To better understand a model, we can visualize its predictions. Use your first model to predict income from each integer age in between 18 and 80. Your predictions should be stored in a numpy array of floats called pred_age.

```
In [71]: # YOUR CODE HERE
         pred_age = a1*np.arange(18,81)+b1
         pred_age
```

```
Out[71]: array([ 1295.66978283,  2369.82931759,  3443.98885235,  4518.1483871 ,
                 5592.30792186,  6666.46745662,  7740.62699138,  8814.78652614,
                 9888.9460609 , 10963.10559566, 12037.26513041, 13111.42466517,
                14185.58419993, 15259.74373469, 16333.90326945, 17408.06280421,
                18482.22233897, 19556.38187372, 20630.54140848, 21704.70094324,
                22778.860478  , 23853.02001276, 24927.17954752, 26001.33908228,
                27075.49861704, 28149.65815179, 29223.81768655, 30297.97722131,
                31372.13675607, 32446.29629083, 33520.45582559, 34594.61536035,
```



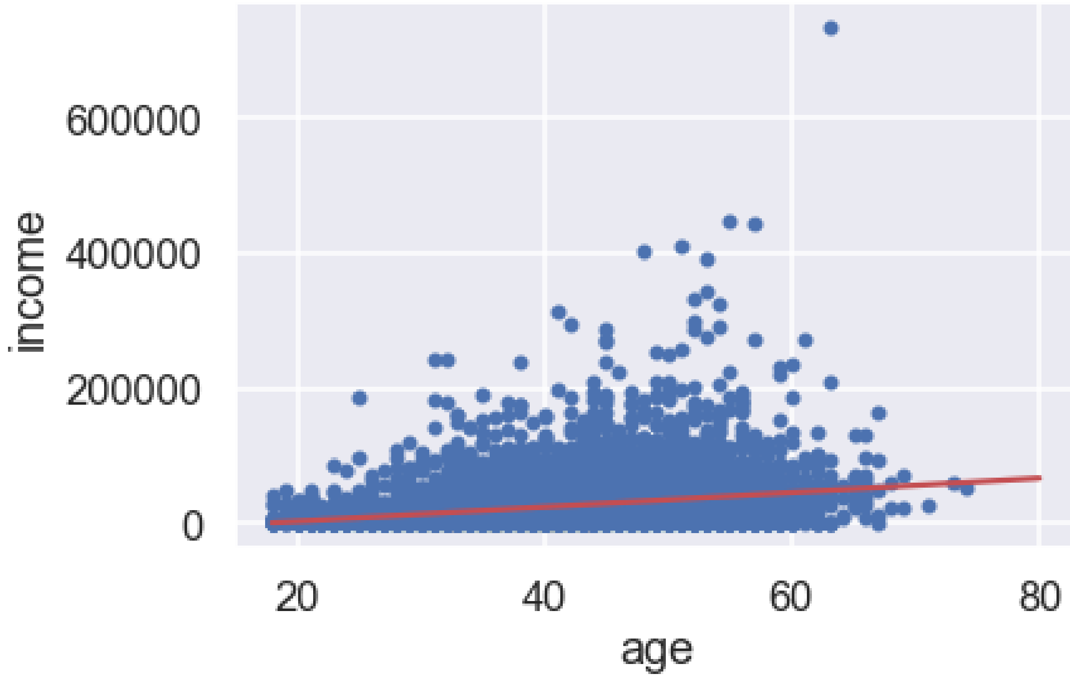

```
35668.7748951 , 36742.93442986, 37817.09396462, 38891.25349938,
39965.41303414, 41039.5725689 , 42113.73210366, 43187.89163842,
44262.05117317, 45336.21070793, 46410.37024269, 47484.52977745,
48558.68931221, 49632.84884697, 50707.00838173, 51781.16791648,
52855.32745124, 53929.486986  , 55003.64652076, 56077.80605552,
57151.96559028, 58226.12512504, 59300.2846598 , 60374.44419455,
61448.60372931, 62522.76326407, 63596.92279883, 64671.08233359,
65745.24186835, 66819.40140311, 67893.56093786])
```

```
In [72]: assert isinstance(pred_age, np.ndarray)
         assert len(pred_age) == 63

         # Your array should contain decimals, not integers
         assert isinstance(pred_age[0], float)
```

Question 6f: Make a scatter plot with income on the y-axis and age on the x-axis. Then, draw your predictions as a red line on top of the scatter plot. Your plot should look like this:

```
In [74]: # YOUR CODE HERE
         df.plot.scatter(x='age', y='income')
         plt.plot(np.arange(18,81), pred_age, 'r')

         f5 = plt.gcf()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as valu
```




```
In [75]: assert f5.gca().has_data()
```

Question 6g: Now, let's do the same for the model that uses steps.

Use your second model to predict income from each multiple of 100 steps in between 2000 and 13000. Your predictions should be stored in a numpy array called `pred_steps`.

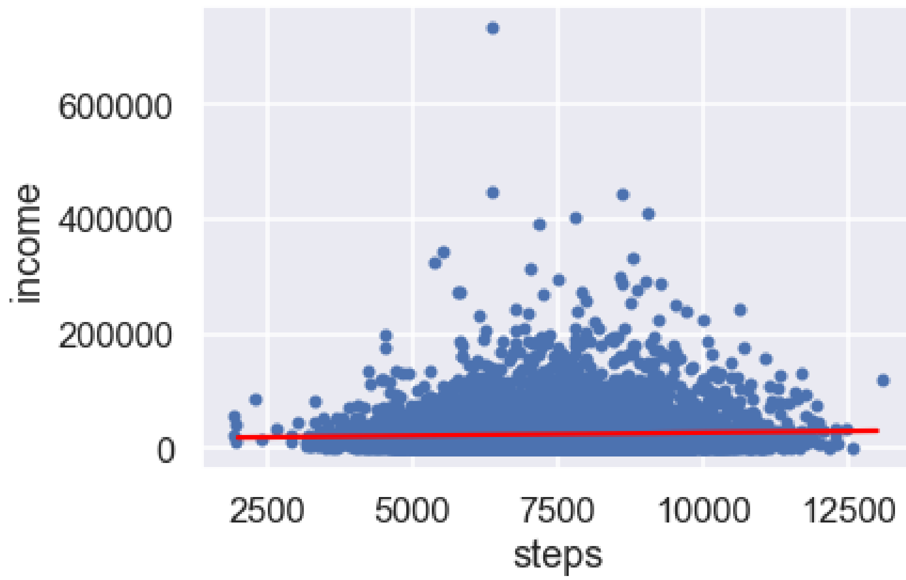
```
In [77]: # YOUR CODE HERE
         pred_steps=a2*np.arange(2000,13001,100)+b2
```

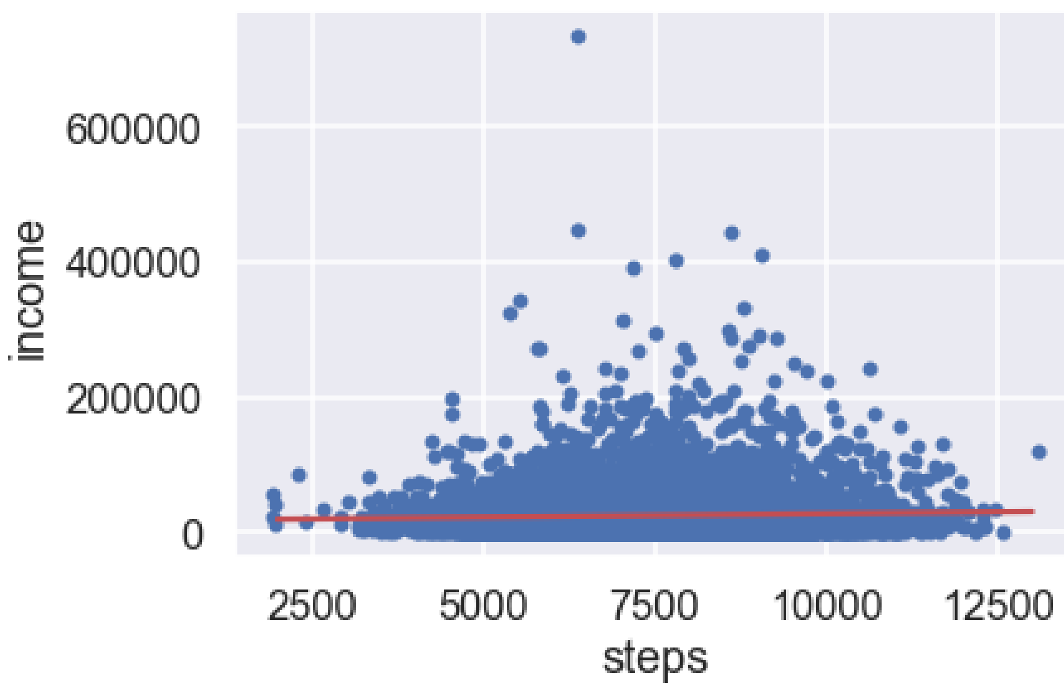
```
In [78]: assert isinstance(pred_steps, np.ndarray)
         assert len(pred_steps) == 111
         # Your array should contain decimals, not integers
         assert isinstance(pred_steps[0], float)
```

Question 6h: Make a scatter plot with income on the y-axis and steps on the x-axis. Then, draw your predictions as a red line on top of the scatter plot. Your plot should look like this:

```
In [79]: # YOUR CODE HERE
         df.plot.scatter(x='steps', y='income')
         plt.plot(np.arange(2000,13001,100), pred_steps, 'r')

         f6 = plt.gcf()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as valu
```






```
In [80]: assert f6.gca().has_data()
```

Question 6i: Notice that both these models perform poorly on this data. For this particular dataset, neither age nor steps seem to have a linear relationship with income. Nonetheless, fitting a linear model is simple and gives us a baseline to compare with more complex models in the future.

Suppose that you were forced to pick one of the above models. Between `'age'` and `'steps'`, which predictor has higher prediction power? Save your response in the variable `model_choice`.

```
In [81]: # YOUR CODE HERE
         model_choice = "age"
```

```
In [82]: assert model_choice
         assert model_choice in {'age', 'steps'}
```

3.1 Done! Upload this notebook to TritonED