Coursebook: Exploratory Data Analysis

- Part 2 of Data Analytics Specialization
- Course Length: 12 hours
- Last Updated: July 2024

- Author: Samuel Chan
- Developed by Algoritma's product division and instructors team

Background

Top-Down Approach

The coursebook is part of the **Data Analytics Specialization** offered by Algoritma. It takes a more accessible approach compared to Algoritma's core educational products, by getting participants to overcome the "how" barrier first, rather than a detailed breakdown of the "why".

This translates to an overall easier learning curve, one where the reader is prompted to write short snippets of code in frequent intervals, before being offered an explanation on the underlying theoretical frameworks. Instead of mastering the syntactic design of the Python programming language, then moving into data structures, and then the `pandas` library, and then the mathematical details in an imputation algorithm, and its code implementation; we would do the opposite: Implement the imputation, then a succinct explanation of why it works and applicational considerations (what to look out for, what are assumptions it made, when *not* to use it etc).

Training Objectives

This coursebook is intended for participants who have completed the preceding courses offered in the **Data Analytics Developer** Specialization. This is the second course, **Exploratory Data Analysis**

The coursebook focuses on: - Why and What: Exploratory Data Analysis - Date Time objects - Categorical data types - Cross Tabulation and Pivot Table - Treating Duplicates and Missing Values

At the end of this course is a Learn-by-Building section, where you are expected to apply all that you've learned on a new dataset, and attempt the given questions.

Data Preparation and Exploration

About 60 years ago, John Tukey defined data analysis as the "procedures for analyzing data, techniques for interpreting the results of such procedures … and all the machinery of mathematical statistics which apply to analyzing data". His championing of EDA encouraged the development of statsitical computing packages, especially S at Bell Labs (which later inspired R).

He wrote a book titled *Exploratory Data Analysis* arguing that too much emphasis in statistics was placed on hypothesis testing (confirmatory data analysis) while not enough was placed on the discovery of the unexpected.

> Exploratory data analysis isolates patterns and features of the data and reveals these forcefully to the analyst.

This course aims to present a selection of EDA techniques – some developed by John Tukey himself – but with a special emphasis on its application to modern business analytics.

In the previous course, we've got our hands on a few common techniques:

- `.head()` and `.tail()`
- `.describe()`
- `.shape` and `.size`
- `.axes`
- `.dtypes`

In the following chapters, we'll expand our EDA toolset with the following additions:

- Tables
- Cross-Tables and Aggregates
- Using `aggfunc` for aggregate functions
- Pivot Tables

```
import pandas as pd
import numpy as np
print(pd.__version__)
```

```
## 2.2.2
```

```
# pandas output display setup
pd.set_option('display.float_format', lambda x: '%.2f' % x)
pd.options.display.float_format = '{:,}'.format
```

Working with Datetime

Given the program's special emphasis on business-driven analytics, one data type of particular interest to us is the `datetime`. In the first part of this coursebook, we've seen an example of `datetime` in the section introducing data types (`employees.joined`).

A large portion of data science work performed by business executives involve time series and/or dates (think about the kind of data science work done by computer vision researchers, and compare that to the work done by credit rating analysts or marketing executives and this special relationship between business and datetime data becomes apparent), so adding a level of familiarity with this format will serve you well in the long run.

As a start, let's read our data,`household.csv`:

```
household = pd.read_csv("data_input/household.csv")
household.dtypes
```

```
## receipt_id            int64
## receipts_item_id      int64
## purchase_time        object
## category             object
## sub_category         object
## format               object
## unit_price          float64
```

```
## discount              int64
## quantity              int64
## yearmonth            object
## dtype: object
```

Notice that all columns are in the right data types, except for `purchase_time`. The correct data type for this column would have to be a `datetime`. In previous module, you've learned how you can use `.astype()` to adjust a data type for a column. In fact, pandas has a function to work with datetime object in particular.

To convert a column `x` to a datetime, we would use:

`'x = pd.to_datetime(x)'`

```
household['purchase_time'] = pd.to_datetime(household['purchase_time'])
household.dtypes
```

```
## receipt_id                      int64
## receipts_item_id                int64
## purchase_time          datetime64[ns]
## category                       object
## sub_category                   object
## format                         object
## unit_price                    float64
## discount                        int64
## quantity                        int64
## yearmonth                      object
## dtype: object
```

Unlike using `astype()`, with `pd.to_datetime()` you are allowed to specify more arguments for the datetime conversion. Why it matters? Suppose we have a column which stores a daily sales data from end of January to the beginning of February:

```
date = pd.Series(['10-01-2020', '12-01-2020', '01-02-2020','02-02-2020'])
date
```

```
## 0    10-01-2020
## 1    12-01-2020
## 2    01-02-2020
## 3    02-02-2020
## dtype: object
```

The legal and cultural expectations for datetime format may vary between countries. In Indonesia for example, most people are used to storing dates in DMY order. Let's see what happen next when we convert our `date` to datetime object:

```
date.astype('datetime64[ns]')
```

```
## 0   2020-10-01
## 1   2020-12-01
## 2   2020-01-02
## 3   2020-02-02
## dtype: datetime64[ns]
```

Take a look on the third observation; rather than representing February 1st as it suppose, the data converted to January 2nd. Thing to note here, for dates with multiple representations, `pandas` will infer it as a month first order by default.

Using `pd.to_datetime`, you can specify your date formatting with parameters such as `format`* or `dayfirst`:

```
# pd.to_datetime(date, format='%d-%m-%Y')
pd.to_datetime(date, dayfirst=True)
```

```
## 0   2020-01-10
## 1   2020-01-12
## 2   2020-02-01
## 3   2020-02-02
## dtype: datetime64[ns]
```

**Using Python's `datetime` module, `pandas` pass the date string to `.strptime()` and follows by what's called Python's strptime directives. The full list of directives can be found in this Documentation.*

Other than `to_datetime`, `pandas` has a number of machineries to work with `datetime` objects. These are convenient for when we need to extract the `month`, or `year`, or `weekday_name` from `datetime`. Some common applications in business analysis include:

- `household['purchase_time'].dt.month`
- `household['purchase_time'].dt.month_name()`
- `household['purchase_time'].dt.year`
- `household['purchase_time'].dt.day`
- `household['purchase_time'].dt.dayofweek`
- `household['purchase_time'].dt.hour`
- `household['purchase_time'].dt.day_name()`

There are also other functions that can be helpful in certain situations. Supposed we want to transform the existing `datetime` column into values of periods we can use the `.to_period` method:

- `household['purchase_time'].dt.to_period('D')`
- `household['purchase_time'].dt.to_period('W')`
- `household['purchase_time'].dt.to_period('M')`
- `household['purchase_time'].dt.to_period('Q')`

Knowledge Check: Date time types
Est. Time required: 20 minutes

1. In the following cell, start again by reading in the `household.csv` dataset. Drop `receipt_id` and `sub_category` columns as we won't use the columns for our analysis.

2. Make sure the `purchase_time` column has converted as a datetime object.
3. Use `x.dt.weekday_name`/`x.dt.day_name()`, assuming `x` is a datetime object to get the day of week. Assign this to a new column in your `household` Data Frame, name it `weekday`
4. The `yearmonth` column stores the information of year and month of the `purchase_time`. Using `dt.to_period()`, how will you recreate the column if you needed the same information?
5. Print the first 5 rows of your data to verify that your preprocessing steps are correct

```
## Your code below


## -- Solution code
```

Tips: In the cell above, start from:

`household = pd.read_csv("data_input/household.csv")`

Inspect the first 5 rows of your data and pay close attention to the `weekday` column.

If you've managed the above exercises, well done! Run the following cell anyway to make sure we're at the same starting point as we go into the next chapter of working with categorical data (factors).

```
# Reference answer
household = pd.read_csv("data_input/household.csv", parse_dates=['purchase_time'])
household['weekday'] = household['purchase_time'].dt.day_name()
household['yearmonth'] = household['purchase_time'].dt.to_period('M')

household.head()
```

```
##    receipt_id  receipts_item_id  ... yearmonth   weekday
## 0     9622257          32369294  ...   2018-07    Sunday
## 1     9446359          31885876  ...   2018-07    Sunday
## 2     9470290          31930241  ...   2018-07    Sunday
## 3     9643416          32418582  ...   2018-07   Tuesday
## 4     9692093          32561236  ...   2018-07  Thursday
##
## [5 rows x 11 columns]
```

Working with Categories

From the output of `dtypes`, we see that there are three variables currently stored as `object` type where a `category` is more appropriate. This is a common diagnostic step, and one that you will employ in almost every data analysis project.

```
household.dtypes
```

```
## receipt_id                    int64
## receipts_item_id              int64
## purchase_time        datetime64[ns]
## category                     object
## sub_category                 object
## format                       object
## unit_price                  float64
## discount                      int64
## quantity                      int64
## yearmonth                 period[M]
## weekday                      object
## dtype: object
```

We'll convert the `weekday` column to a categorical type using `.astype()`. `astype('int64')` converts a Series to an integer type, and `.astype(category)` logically, converts a Series to a categorical.

By default, `.astype()` will raise an error if the conversion is not successful (we call them "exceptions"). In an analysis-driven environment, this is what we usually prefer. However, in certain production settings, you don't want the exception to be raised and rather return the original object (`errors='ignore'`).

```
household['weekday'] = household['weekday'].astype('category', errors='ignore')
household.dtypes
```

```
## receipt_id                    int64
## receipts_item_id              int64
## purchase_time        datetime64[ns]
## category                     object
## sub_category                 object
## format                       object
## unit_price                  float64
## discount                      int64
## quantity                      int64
## yearmonth                 period[M]
## weekday                    category
## dtype: object
```

Go ahead and perform the other conversions in the following cell. When you're done, use `dtypes` to check that you have the categorical columns stored as `category`.

```
## Your code here
```

Alternative Solutions (optional)

```
data1 = household.select_dtypes(exclude='object')
data2 = household.select_dtypes(include='object').apply(pd.Series.astype, dtype='category')

pd.concat([data1, data2], axis=1).dtypes
```

Solution 1:

```
## receipt_id                    int64
## receipts_item_id              int64
## purchase_time        datetime64[ns]
## unit_price                  float64
## discount                      int64
## quantity                      int64
## yearmonth                 period[M]
## weekday                    category
## category                   category
## sub_category               category
## format                     category
## dtype: object
```

```
objectcols = household.select_dtypes(include='object')
household[objectcols.columns] = objectcols.apply(lambda x: x.astype('category'))
household.dtypes
```

Solution 2

```
## receipt_id                    int64
## receipts_item_id              int64
## purchase_time        datetime64[ns]
## category                   category
## sub_category               category
## format                     category
## unit_price                  float64
## discount                      int64
## quantity                      int64
## yearmonth                 period[M]
## weekday                    category
## dtype: object
```

Contingency Tables

One of the simplest EDA toolkit is the frequency table (contingency tables) and cross-tabulation tables. It is highly familiar, convenient, and practical for a wide array of statistical tasks. The simplest form of a table is to display counts of a `categorical` column.

In `pandas`, each column of a `DataFrame` is a `Series`. To get the counts of each unique levels in a categorical column, we can use `.value_counts()`. The resulting object is a `Series` and in descending order so that the most frequent element is on top.

Try and perform `.value_counts()` on the `format` column, adding either:

- `sort=False` as a parameter to prevent any sorting of elements, or
- `ascending=True` as a parameter to sort in ascending order instead

```
household['sub_category'].value_counts(sort=False, ascending=True)
```

```
## sub_category
## Detergent    36000
## Rice         12000
## Sugar        24000
## Name: count, dtype: int64
```

```
## Your code below


## -- Solution code
```

`crosstab` is a very versatile solution to producing frequency tables on a `DataFrame` object. Its utility really goes further than that but we'll start with a simple use-case.

Consider the following code: we use `pd.crosstab()` passing in the values to group by in the rows (`index`) and columns (`columns`) respectively.

```python
pd.crosstab(index=household['sub_category'], columns="count")
```

```
## col_0         count
## sub_category
## Detergent     36000
## Rice          12000
## Sugar         24000
```

Realize that in the code above, we're setting the row (index) to be `sub_category` and the function will by default compute a frequency table.

```python
pd.crosstab(index=household['sub_category'], columns="count", normalize='all')
```

```
## col_0                        count
## sub_category
## Detergent                      0.5
## Rice          0.16666666666666666
## Sugar          0.3333333333333333
```

In the cell above, we set the values to be normalized over each columns, and this will divide each values in place over the sum of all values. This is equivalent to a manual calculation:

```python
catego = pd.crosstab(index=household['sub_category'], columns="count")
#catego / catego.sum()
```

We can also use the same `crosstab` method to compute a cross-tabulation of two factors. In the following cell, the `index` references the sub-category column while the `columns` references the format column:

```python
pd.crosstab(index=household['sub_category'], columns=household['format'])
```

```
## format        hypermarket  minimarket  supermarket
## sub_category
## Detergent            2611       24345         9044
## Rice                  999        7088         3913
## Sugar                1761       15370         6869
```

This is intuitive in a way: We use `crosstab()` which, we recall, computes the count and we pass in `index` and `columns` which correspond to the row and column respectively.

When we add `margins=True` to our method call, then an extra row and column of margins (subtotals) will be included in the output:

```python
pd.crosstab(index=household['sub_category'],
            columns=household['format'],
            normalize='columns',
            margins=True)*100
```

```
## format              hypermarket  ...                All
## sub_category                     ...
## Detergent     48.61292124371626  ...               50.0
## Rice         18.599888288959228  ... 16.666666666666664
## Sugar         32.78719046732452  ...  33.33333333333333
##
## [3 rows x 4 columns]
```

In the following cell, use `pd.crosstab()` with `yearmonth` as the row and `format` as the column. Set `margins=True` to get a total across the row and columns.

```
## Your code below


## -- Solution code
```

If you want an extra challenge, try and modify your code above to include a `normalize` parameter.

`normalize` accepts a boolean value, or one of `all`, `index` or `columns`. Since we want it to normalize across each row, we will set this parameter to the value of `index`.

Aggregation Table

In the following section, we will introduce another parameter to perform aggregation on our table. The `aggfunc` parameter when present, required the `values` parameter to be specified as well. `values` is the values to aggregate according to the factors in our index and columns:

```
pd.crosstab(index=household['sub_category'],
            columns='mean',
            values=household['unit_price'],
            aggfunc='mean')
```

```
## col_0                       mean
## sub_category
## Detergent     17,893.79321398722
## Rice             70,013.14631275
## Sugar        12,645.066023837499
```

Knowledge Check: Cross tabulation

Create a cross-tab using `sub_category` as the index (row) and `format` as the column. Fill the values with the median of `unit_price` across each row and column. Add a subtotal to both the row and column by setting `margins=True`.

1. On average, Sugar is cheapest at…?
2. On average, Detergent is most expensive at…?

Create a new cell for your code and answer the questions above.

```
## Your code below


## -- Solution code
```

Reference answer:

```
pd.crosstab(index=household['sub_category'],
            columns=household['format'],
            values=household['unit_price'],
            aggfunc='median', margins=True)
```

Higher-dimensional Tables

If we need to inspect our data in higher resolution, we can create cross-tabulation using more than one factor. This allows us to yield insights on a more granular level yet have our output remain relatively compact and structured:

```
pd.crosstab(index=household['yearmonth'],
            columns=[household['format'], household['sub_category']],
            values=household['unit_price'],
            aggfunc='median')
```

```
## format       hypermarket                     ... supermarket
## sub_category   Detergent      Rice     Sugar ...   Detergent      Rice     Sugar
## yearmonth                                    ...
## 2017-10         17,400.0 64,000.0 12,500.0  ...    16,925.0 64,000.0 12,500.0
## 2017-11         16,770.0 64,000.0 12,400.0  ...    16,500.0 64,000.0 12,400.0
## 2017-12         17,500.0 64,000.0 12,000.0  ...    16,600.0 64,000.0 12,400.0
## 2018-01         16,800.0 64,000.0 12,275.0  ...    16,700.0 64,000.0 12,400.0
## 2018-02         17,500.0 64,000.0 11,990.0  ...    16,200.0 64,000.0 12,290.0
## 2018-03         16,900.0 64,000.0 12,000.0  ...    15,680.0 64,000.0 12,400.0
## 2018-04         16,815.0 64,000.0 11,990.0  ...    15,700.0 64,000.0 12,400.0
## 2018-05         16,950.0 64,000.0 12,000.0  ...    16,700.0 64,000.0 12,400.0
## 2018-06         16,550.0 64,000.0 12,300.0  ...    16,700.0 64,000.0 12,400.0
## 2018-07         16,550.0 64,000.0 12,325.0  ...    16,600.0 64,000.0 12,300.0
## 2018-08         16,839.0 62,600.0 12,000.0  ...    17,000.0 64,000.0 12,300.0
## 2018-09         16,720.0 60,000.0 11,900.0  ...    16,990.0 62,550.0 12,300.0
##
## [12 rows x 9 columns]
```

In `pandas` we call a higher-dimensional tables as Multi-Index Dataframe. We are going to dive deeper into the structure of the object on the the next chapter.

Pivot Tables

If our data is already in a `DataFrame` format, using `pd.pivot_table` can sometimes be more convenient compared to a `pd.crosstab`.

Fortunately, much of the parameters in a `pivot_table()` function is the same as `pd.crosstab()`. The noticable difference is the use of an additional `data` parameter, which allow us to specify the `DataFrame` that is used to construct the pivot table.

We create a `pivot_table` by passing in the following: - `data`: our `DataFrame` - `index`: the column to be used as rows - `columns`: the column to be used as columns - `values`: the values used to fill in the table - `aggfunc`: the aggregation function

```
pd.pivot_table(
    data=household,
    index='yearmonth',
    columns=['format','sub_category'],
    values='unit_price',
    aggfunc='median'
)
```

```
## format       hypermarket                     ... supermarket
## sub_category   Detergent     Rice     Sugar  ...   Detergent     Rice    Sugar
## yearmonth                                    ...
## 2017-10         17,400.0 64,000.0 12,500.0  ...    16,925.0 64,000.0 12,500.0
## 2017-11         16,770.0 64,000.0 12,400.0  ...    16,500.0 64,000.0 12,400.0
## 2017-12         17,500.0 64,000.0 12,000.0  ...    16,600.0 64,000.0 12,400.0
## 2018-01         16,800.0 64,000.0 12,275.0  ...    16,700.0 64,000.0 12,400.0
## 2018-02         17,500.0 64,000.0 11,990.0  ...    16,200.0 64,000.0 12,290.0
## 2018-03         16,900.0 64,000.0 12,000.0  ...    15,680.0 64,000.0 12,400.0
## 2018-04         16,815.0 64,000.0 11,990.0  ...    15,700.0 64,000.0 12,400.0
## 2018-05         16,950.0 64,000.0 12,000.0  ...    16,700.0 64,000.0 12,400.0
## 2018-06         16,550.0 64,000.0 12,300.0  ...    16,700.0 64,000.0 12,400.0
## 2018-07         16,550.0 64,000.0 12,325.0  ...    16,600.0 64,000.0 12,300.0
## 2018-08         16,839.0 62,600.0 12,000.0  ...    17,000.0 64,000.0 12,300.0
## 2018-09         16,720.0 60,000.0 11,900.0  ...    16,990.0 62,550.0 12,300.0
##
## [12 rows x 9 columns]
##
## <string>:1: FutureWarning: The default value of observed=False is deprecated and will change to obse
```

A key difference between `crosstab` and `pivot_table` is that `crosstab` uses `len` (or `count`) as the default aggregation function while `pivot_table` using the mean. Copy the code from the cell above and make a change: use `sum` as the aggregation function instead:

```
## Your code below


## -- Solution code
```

Missing Values and Duplicates

During the data exploration and preparation phase, it is likely we come across some problematic details in our data. This could be the value of *-1* for the *age* column, a value of *blank* for the *customer segment* column, or a value of *None* for the *loan duration* column. All of these are examples of "untidy" data, which is rather common depending on the data collection and recording process in a company.

In `pandas`, we use `NaN` (not a number) to denote missing data; The equivalent for datetime is `NaT` but both are essentially compatible with each other. From the docs: > The choice of using `NaN` internally to denote missing data was largely for simplicity and performance reasons. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

consider the following data, there are few missing values. We will cover on how to treat the data.

```
household_missing = pd.read_csv('data_input/household-missing.csv', index_col=0, parse_dates=['purchase
household_missing.head()
```

```
##                        purchase_time category  ... quantity  weekday
## receipts_item_id                               ...
## 32000000                         NaT      NaN  ...      NaN      NaN
## 32000001                         NaT      NaN  ...      NaN      NaN
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0  Tuesday
## 32000002                         NaT      NaN  ...      NaN      NaN
## 32000003                         NaT      NaN  ...      NaN      NaN
```

```
##
## [5 rows x 7 columns]
```

Missing Values Treatment

In the beginning of this section, I used `reindex` to "inject" some rows where values don't exist (receipts item id 32000000 through 32000004) and also set `math.nan` on one of the values for `weekday`. Notice from the output that between row 3 to 8 there are at least a few rows with missing data. We can use `isna()` and `notna()` to detect missing values. An example code is as below:

```
household_missing['weekday'].isna()
```

```
## receipts_item_id
## 32000000     True
## 32000001     True
## 32030785    False
## 32000002     True
## 32000003     True
## 32000004     True
## 32369294    False
## 31885876     True
## 31930241    False
## 32418582    False
## 32561236    False
## 32030785    False
## 32935097    False
## 32593606    False
## 32573843    False
## 31913062    False
## 31125365    False
## 32856560    False
## 32552145    False
## 32369065    False
## Name: weekday, dtype: bool
```

A common way of using the `.isna()` method is to combine it with the subsetting methods we've learned in previous lessons:

```
household_missing[household_missing['weekday'].isna()]
```

```
##                        purchase_time category  ... quantity  weekday
## receipts_item_id                                ...
## 32000000                         NaT      NaN  ...      NaN      NaN
## 32000001                         NaT      NaN  ...      NaN      NaN
## 32000002                         NaT      NaN  ...      NaN      NaN
## 32000003                         NaT      NaN  ...      NaN      NaN
## 32000004                         NaT      NaN  ...      NaN      NaN
## 31885876         2018-07-15 16:17:00     Rice  ...      1.0      NaN
##
## [6 rows x 7 columns]
```

Go ahead and use `notna()` to extract all the rows where `weekday` column is not missing:

```
## Your code below


## -- Solution code
```

Another common use-case in missing values treatment is to count the number of `NAs` across each column:

```
household_missing.isna().sum()
```

```
## purchase_time    5
## category         5
## format           5
## unit_price       5
## discount         5
## quantity         5
## weekday          6
## dtype: int64
```

When we are certain that the rows with `NAs` can be safely dropped, we can use `dropna()`, optionally specifying a threshold. By default, this method drops the row if any NA value is present (`how='any'`), but it can be set to do this only when all values are NA in that row (`how='all'`).

```
# drops row if all values are NA
household2.dropna(how='all')

# drops row if it doesn't have at least 5 non-NA values
household2.dropna(thresh=5)
```

```
household_missing.dropna(thresh = 6) # NA in 6 columns
```

```
##                        purchase_time category  ... quantity    weekday
## receipts_item_id                               ...
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0    Tuesday
## 32369294         2018-07-22 21:19:00     Rice  ...      1.0     Sunday
## 31885876         2018-07-15 16:17:00     Rice  ...      1.0        NaN
## 31930241         2018-07-15 12:12:00     Rice  ...      3.0     Sunday
## 32418582         2018-07-24 08:27:00     Rice  ...      1.0    Tuesday
## 32561236         2018-07-26 11:28:00     Rice  ...      1.0   Thursday
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0    Tuesday
## 32935097         2018-07-29 18:18:00     Rice  ...      1.0     Sunday
## 32593606         2018-07-25 12:48:00     Rice  ...      1.0  Wednesday
## 32573843         2018-07-26 16:41:00     Rice  ...      1.0   Thursday
## 31913062         2018-07-14 21:17:00     Rice  ...      3.0   Saturday
## 31125365         2018-07-02 15:39:00     Rice  ...      1.0     Monday
## 32856560         2018-07-31 05:51:00     Rice  ...      1.0    Tuesday
## 32552145         2018-07-26 11:43:00     Rice  ...      1.0   Thursday
## 32369065         2018-07-23 14:22:00     Rice  ...      1.0     Monday
##
## [15 rows x 7 columns]
```

Some common methods when working with missing values are demonstrated in the following section. We make a copy of the NA-included DataFrame, and name it `household_clean`:

```
household_clean = household_missing.copy()
household_clean.head()
```

```
##                          purchase_time category  ... quantity  weekday
## receipts_item_id                                 ...
## 32000000                           NaT      NaN  ...      NaN      NaN
## 32000001                           NaT      NaN  ...      NaN      NaN
## 32030785           2018-07-17 18:05:00     Rice  ...      1.0  Tuesday
## 32000002                           NaT      NaN  ...      NaN      NaN
## 32000003                           NaT      NaN  ...      NaN      NaN
##
## [5 rows x 7 columns]
```

In the following cell, the technique is demonstrably repetitive or even verbose. This is done to give us an idea of all the different options we can pick from.

You may observe, for example that the two lines of code are functionally identical: - `.fillna(0)` - `.replace(np.nan, 0)`

```
household_clean.dtypes
```

```
## purchase_time    datetime64[ns]
## category                 object
## format                   object
## unit_price              float64
## discount                float64
## quantity                float64
## weekday                  object
## dtype: object
```

```
household_clean[['category', 'format','discount']] = household_clean[['category', 'format','discount']]
household_clean['unit_price'] = household_clean['unit_price'].fillna(0)
household_clean['purchase_time'] = household_clean['purchase_time'].bfill() # same likely fillna(method
household_clean['weekday'] = household_clean['purchase_time'].dt.day_name()
household_clean['quantity'] = household_clean['quantity'].replace(np.nan, -1)

household_clean.head()
```

```
##                          purchase_time category  ... quantity  weekday
## receipts_item_id                                 ...
## 32000000           2018-07-17 18:05:00  Missing  ...     -1.0  Tuesday
## 32000001           2018-07-17 18:05:00  Missing  ...     -1.0  Tuesday
## 32030785           2018-07-17 18:05:00     Rice  ...      1.0  Tuesday
## 32000002           2018-07-22 21:19:00  Missing  ...     -1.0   Sunday
## 32000003           2018-07-22 21:19:00  Missing  ...     -1.0   Sunday
##
## [5 rows x 7 columns]
```

Duplicated Data

To observe for duplicates in our data, we can use `duplicate()` and combine it with the subsetting method as below:

```
household_clean[household_clean.duplicated(keep=False)]
```

```
##                          purchase_time category  ... quantity  weekday
## receipts_item_id                                  ...
## 32000000         2018-07-17 18:05:00  Missing  ...     -1.0  Tuesday
## 32000001         2018-07-17 18:05:00  Missing  ...     -1.0  Tuesday
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0  Tuesday
## 32000002         2018-07-22 21:19:00  Missing  ...     -1.0   Sunday
## 32000003         2018-07-22 21:19:00  Missing  ...     -1.0   Sunday
## 32000004         2018-07-22 21:19:00  Missing  ...     -1.0   Sunday
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0  Tuesday
##
## [7 rows x 7 columns]
```

When we have data where duplicated observations are recorded, we can use `.drop_duplicates()` specifying whether the first occurence or the last should be kept. You can specify whether you want to keep the first or last occurence with `keep= 'first'/'last'`.

```
household_clean.drop_duplicates()
```

```
##                          purchase_time category  ... quantity    weekday
## receipts_item_id                                  ...
## 32000000         2018-07-17 18:05:00  Missing  ...     -1.0    Tuesday
## 32030785         2018-07-17 18:05:00     Rice  ...      1.0    Tuesday
## 32000002         2018-07-22 21:19:00  Missing  ...     -1.0     Sunday
## 32369294         2018-07-22 21:19:00     Rice  ...      1.0     Sunday
## 31885876         2018-07-15 16:17:00     Rice  ...      1.0     Sunday
## 31930241         2018-07-15 12:12:00     Rice  ...      3.0     Sunday
## 32418582         2018-07-24 08:27:00     Rice  ...      1.0    Tuesday
## 32561236         2018-07-26 11:28:00     Rice  ...      1.0   Thursday
## 32935097         2018-07-29 18:18:00     Rice  ...      1.0     Sunday
## 32593606         2018-07-25 12:48:00     Rice  ...      1.0  Wednesday
## 32573843         2018-07-26 16:41:00     Rice  ...      1.0   Thursday
## 31913062         2018-07-14 21:17:00     Rice  ...      3.0   Saturday
## 31125365         2018-07-02 15:39:00     Rice  ...      1.0     Monday
## 32856560         2018-07-31 05:51:00     Rice  ...      1.0    Tuesday
## 32552145         2018-07-26 11:43:00     Rice  ...      1.0   Thursday
## 32369065         2018-07-23 14:22:00     Rice  ...      1.0     Monday
##
## [16 rows x 7 columns]
```

```
print(household_clean.shape)
```

```
## (20, 7)
```

```
print(household_clean.drop_duplicates(keep="first").shape)
```

```
## (16, 7)
```

Knowledge Check: Duplicates and Missing Value
Est. Time required: 20 minutes

1. Duplicates may mean a different thing from a data point-of-view and a business analyst's point-of-view. You want to be extra careful about whether the duplicates is an intended characteristic of your data, or whether it poses a violation to the business logic.

   - a. A medical center collects anonymized heart rate monitoring data from patients. It has duplicate observations collected across a span of 3 months
   - b. An insurance company uses machine learning to deliver dynamic pricing to its customers. Each row contains the customer's name, occupation / profession and historical health data. It has duplicate observations collected across a span of 3 months
   - c. On our original `household` data, check for duplicate observations. Would you have drop the duplicated rows?

---

2. Once you've identified the missing values, there are 3 common ways to deal with it:

   - a. Use `dropna` with a reasonable threshold to remove any rows that contain too little values rendering it unhelpful to your analysis
   - b. Replace the missing values with a central value (mean or median)
   - c. Imputation through a predictive model
     - In a dataframe where `salary` is missing but the bank has data about the customer's occupation / profession, years of experience, years of education, seniority level, age, and industry, then a machine learning model such as regression or nearest neighbor can offer a viable alternative to the mean imputation approach

```
## Your Answer here
```