Coursebook: Python for Data Analysts

- Part 1 of Data Analytics Specialization
- Course Length: 12 hours
- Last Updated: September 2021

- 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).

For the most part, experience in Python programming is good to have but not required. Familiarity with data manipulation and data structures in a different programming language a welcome addition but again, not required.

Training Objectives

This coursebook is intended for participants new to the world of data analysis and / or programming. No prior programming knowledge is assumed.

The coursebook focuses on: - Introduction to the `pandas` library. - Introduction to `DataFrame` - Data Types - Exploratory Data Analysis I - Indexing and Subsetting

The final part of this course is a Graded Asssignment, where you are expected to apply all that you've learned on a new dataset, and attempt the given questions.

Python for Data Analysts

Since you'll spend a great deal of your time working with data in Jupyter Notebook, I think it's important to get yourself familiar with notebok documents (or "notebooks").

> Notebook documents are documents produced by *Jupyter Notebook*, which contain both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc…). Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis.

Jupyter Notebook provides an easy-to-use, interactive data science environment that doesn't only work as an IDE, but also as a presentation tool (as it can be exported to other document formats, such as .HTML, .PDF, etc).

If you're a seasoned programmer, the **Ctrl + Shift + P** combination will bring up a shortcut reference guide that helps you use Jupyter Notebook more effectively.

Variables and Keywords

Later on, we will often use assignment statements (`=`) to creates new variables. A variable is a name that refers to a value.

```
activity = 'programming'

print(activity)
```

```
## programming
```

Thing to note here, like other programming languages, Python is **case-sensitive**, so `activity` and `Activity` are different symbols and will point to different variables.

```
"activity" == "Activity"
```

```
## False
```

```
activity == activity
```

```
## True
```

Our previous code returned `True` as the output. Try to create a new variable and use `True` as the variable name, then see what happen.

> SyntaxError: can't assign to keyword

```
## Your code below:
```

A couple of things to note here. `True`, along with its opposite, `False` are among a reserved list of vocabulary referred to as **Python Keywords**. We cannot use keyword as variable name, function name or assign values to them, essentially treating them as an identifier.

Interestingly, all python keywords except **True**, **False** and **None** are in lowercase and they must be written as it is. As of Python 3.7 (latest version of Python as of this writing), there are 33 keywords:

`True`, `False`, `None`, `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else` `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass` `raise`, `return`, `try`, `while`, `with`, `yield`

Introduction to pandas Library

Working with DataFrame

We will start off by learning about a powerful Python data analysis library by the name of `pandas`. Its official documentation introduces itself as the “fundamental high-level building block for doing practical, real world data analysis in Python”, and strive to do so by implementing many of the key data manipulation functionalities in R. This makes `pandas` a core member of many Python-based scientific computing environments.

From its official documentation:

> Python has long been great for data munging and preparation, but less so for data analysis and modeling. pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R.

To use `pandas`, we will use Python’s `import` function. Once imported, all `pandas` function can be accessed using the *pandas.function_name* notation.

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

```
## 1.2.4
```

```
rice = pd.read_csv("data_input/rice.csv", index_col=0)
rice.head()
```

```
##    receipt_id  receipts_item_id    purchase_time  ... discount quantity yearmonth
## 1     9622257          32369294  7/22/2018 21:19  ...        0        1   2018-07
## 2     9446359          31885876  7/15/2018 16:17  ...        0        1   2018-07
## 3     9470290          31930241  7/15/2018 12:12  ...        0        3   2018-07
## 4     9643416          32418582   7/24/2018 8:27  ...        0        1   2018-07
## 5     9692093          32561236  7/26/2018 11:28  ...        0        1   2018-07
##
## [5 rows x 10 columns]
```

In the code above, we used `.read_csv()` to read a csv file from a specified path. Notice that we set `index_col=0` so the first column in the csv is used as the index. By default, this function treats the first row as the header row. We can add `header=None` to the function call telling `pandas` to read in a CSV without headers.

You may find it curious that we use `0` to reference the first element of an axis; This is because Python uses 0-based indexing, a behavior that is different from other languages such as R and Matlab.

Knowledge Check: Error Referring to the previous **Python keywords** concept. Which of the following 4 lines of code will evaluate without raising an error?

- ☐ `pd.read_csv("data_input/rice.csv", index_col=false)`
- ☐ `Import pandas as pd`
- ☐ `print(100-2)`
- ☐ `None = 2`

```
## Your code below


## -- Solution code
```

You'll be tempted to go through all the keywords above and try to wrap your head around each one of them. If this proves to be a tad overwhelming, my recommendation is to move along the rest of the section; Most of us do not know the inner workings of every components of our car engine, but that shouldn't stop you from being an effective driver.

As stated in the beginning of this course book, we're choosing a top-down approach and concepts will be presented on a "need-to-know" basis. We'll no doubt come across many of the keywords again (since collectively they form the backbone of the language) but for now, there is no need to stress about them if that only serve to discourage you from learning to code.

What you need to know: - Python, as with `R`, `Swift`, `C`, and many other languages, are case-sensitive. `Sales` and `sales` refer to different objects.
- You cannot use any Python keywords as identifers. - When naming your variables, start with a letter and use underscore (`_`) to join multiple words. - Wrong: `2019`, `2019sales`, `sales-2019`, `sales.2019` - Correct: `sales_2019`, `profit_after_tax`

Dive Deeper Let's dive deeper into understanding the `index_col` parameter. From the documentation:

> `index_col` : int or sequence or `False`
> Column to use as the row labels of the DataFrame.

1. How would you change the `read.csv()` code such that the DataFrame uses `receipt_id` as the row label (index)?
2. `pandas.DataFrame.head()` accepts an additional parameter, `n`, and returns the first `n` rows of the DataFrame; Set `n=8` to see the first 8 rows of your `rice` DataFrame
3. The opposite of `.head()` is `.tail()`. It returns the last `n` row of your DataFrame. Create a new cell below and print the last 4 rows of our DataFrame

Reminder: Python uses 0-based indexing, and `receipt_id` is the second column in the csv

```
## Your code below


## -- Solution code
```

Data Types

`pandas` allow data analysts to create Series objects and DataFrame objects. Series is used to represent a one-dimensional array whereas DataFrame emulates the functionality of "Data Frames" in R and is useful for tabular data.

In practice, a large proportion of our data is tabular: when we import data from a relational database (MySQL, Postgre) or from a spreadsheet software (Google Sheets, Microsoft Excel) we can represent these data as a DataFrame object.

When we call pd.read_csv() earlier, pandas will try to infer data types from the values in each column. Sometimes, it get it right but more often that not, a data analyst's intervention is required. In the following sub-section, we'll learn about various techniques an analyst have at his/her disposal when it comes to the treatment of pandas data types.

```
print(rice.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
```

dtypes simply stands for "data types". Because rice is a pandas object, accessing the dtypes attribute will return a series with the data type of each column.

Knowledge check: .dtypes and pandas attributes Look at the following code - what is the expected output from the following code? Why?

```
x = [2019, 4, 'data science']
x.dtypes
```

Hint: Try type(x) and verify the type for object x.

```
## Your code below


## -- Solution code
```

Let's take a look at some examples of DataFrame.dtypes:

```
employees = pd.DataFrame({
    'name': ['Anita', 'Brian'],
    'age': [34, 29],
    'joined': [pd.Timestamp('20190410'), pd.Timestamp('20171128')],
    'degree': [True, False],
    'hourlyrate': [35.5, 29],
    'division': ['HR', 'Product']
})
employees.dtypes
```

```
## name                  object
## age                    int64
## joined        datetime64[ns]
## degree                  bool
## hourlyrate           float64
## division              object
## dtype: object
```

```
employees
```

```
##     name  age     joined  degree  hourlyrate division
## 0  Anita   34 2019-04-10    True        35.5       HR
## 1  Brian   29 2017-11-28   False        29.0  Product
```

Let's go through the columns and their data types from the above `DataFrame`:

- `name` [`object`]: store text values
- `age` [`int`]: integer values
- `joined` [`datetime`]: date and time values
- `degree` [`bool`]: True/False values
- `hourlyrate` [`float`]: floating point values
- `division` [`object`]: store text values

Among these columns, only `age` and `hourlyrate` are columns with numeric values. This is a simple, but important, observation to make as we make our way into the Exploratory Data Analysis phase. But before we do, let's do one more exercise. Take a closer look at the Data Frame we just created again.

Out of the 6 columns, one of them is of special interest to our next discussion, **categorical values**.

Categorical and Numerical Variables

When working with categories, it is recommended both from a business point of a view and a technical one to use `pandas` categorical data type. From a business perspective, this adds clarity to the analyst's mind about the type of data he/she is working with. This informs and guides the analysis, on questions such as which statistical methods or plot types to use.

From a technical viewpoint, the memory savings – and in turn, computation speed as well as computational resources – can be quite significant. Specifically, the docs remarked:

> The memory usage of a `Categorical` is proportional to the number of categories plus the length of the data. In contrast, an `object` dtype is a constant times the length of the data

One more important remakrs from the docs:

> Categoricals are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (categories; levels in R). Examples are gender, social class, blood type, country affiliation or rating via Likert scales.

Can you spot which of our column holds values that should be encoded in the `category` data type? Once you've spotted it, use the `astype('category')` method to perform the conversion. Remember to re-assign this new column so the original column (`object`) type is overwritten with the new `category` type column.

Examples:

```
# convert marital_status to category
employees['marital_status'] = employees['marital_status'].astype('category')

# convert experience to integer
employees['experience'] = employees['experience'].astype('int')
```

```
## Your code below


## -- Solution code
```

Use `employees.dtypes` to confirm that you've done the exercise above correctly:

```
## Your code below


## -- Solution code
```

In most real-world projects, your work as a data analyst will involve working with **categorical**, **numeric** and **datetime** values; either treating them as "features" or "target". In the case of machine learning:

- A **categorical** target represents a classification problem
- A **numeric** target represents a regression problem

Exploratory Data Analysis Tools

In simple words, exploratory data analysis (EDA) refers to the process of performing initial investigations on data, often with the objective of becoming familiar with certain characteristics of the data. This is usually done with the aid of summary statistics and simple graphical techniques that purposefully uncover the structure of our data.

We'll start off by using some of the most convenient EDA tools conveniently built into `pandas`. Particularly, this is a summary of what we'll cover in common EDA workflows:

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

```
employees.describe()
```

```
##              age  hourlyrate
## count   2.000000    2.000000
## mean   31.500000   32.250000
## std     3.535534    4.596194
## min    29.000000   29.000000
## 25%    30.250000   30.625000
## 50%    31.500000   32.250000
## 75%    32.750000   33.875000
## max    34.000000   35.500000
```

The `.describe()` method will generate descriptive statistics of our data, and by default include all numeric columns in our DataFrame. It lists 8 statistical properties of each attribute, for example on numerical values: - Count - Mean - Standard Deviation - Minimum Value - 25th Percentile (Q1) - 50th Percentile (Q2/Median) - 75th Percentile (Q3) - Maximum Value

The `describe()` method will generate descriptive statistics of our data, and by default include all numeric columns in our DataFrame. The code above calls `.describe()` on `employees`, from which there are two numeric columns. This method is an "instruction" to perform something (functions) associated with the object. We've seen earlier how to use `.head()` and `.tail()` on our DataFrame: these are also method calls!

We can add an `include` parameter in the `.describe()` method call, which takes a list-like of dtypes to be included or `all` for all columns of the dataframe.

Add a new cell below, calling `describe()` but only on columns of `object` and `datetime` types (`['object', 'datetime']`).

```
## Your code below


## -- Solution code
```

Very often, we also want to know the shape of our data - i.e. how many rows and columns are there in our DataFrame?

Our DataFrame has attributes that we can use to answer those questions. An attribute is a value stored within an object that describe an aspect of the object's characteristic. In the following call, we are asking for the `.shape` and the `.size` attribute of our `rice` DataFrame.

Tip: Unlike `describe()`, which is a method call; `shape` and `size` are **`attributes`** of our DataFrame - that means no function is evaluated; Only a value stored in the object's instance is looked up and returned.

```
print(rice.shape)
```

```
## (12000, 10)
```

```
print(rice.size)
```

```
## 120000
```

`size` returns the number of elements in the `rice` DataFrame. Because we have 12,000 rows and 10 columns, the total number of elements would be a total of 120,000.

Use `.shape` on the `employees` DataFrame. From the resulting output, could you tell what would be the result of calling `employees.size`?

```
## Your code below


## -- Solution code
```

One other attribute that is often useful is `.axes`, which return a list representing the axes of our DataFrame. Most likely, this would be a list of length 2, one for the row axis and one for the column axis, in that particular order.

Because it is ordered that way, calling `.axes[0]` would return the first item of that list, which would be the row axis (or row names if present) and calling `.axes[1]` would return the column axis, which would be equivalent to calling `rice.columns`:

```
rice.axes[1]
```

```
## Index(['receipt_id', 'receipts_item_id', 'purchase_time', 'category',
##        'sub_category', 'format', 'unit_price', 'discount', 'quantity',
##        'yearmonth'],
##       dtype='object')
```

We've covered `.dtypes` in earlier sections, so go ahead and practice inspecting the data types of `rice` DataFrame. Are the columns in the right data types? If they are not, formulate a mental checklist of type conversion you need to perform.

```
rice.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
```

Compare your mental checklist with the following code. We converted `purchase_time` to a `datetime` type, and perform the conversion for the categorical columns as well:

```
rice['purchase_time'] = rice['purchase_time'].astype('datetime64')
rice[['category', 'sub_category', 'format']] = rice[['category', 'sub_category', 'format']].astype('cate
rice.dtypes
```

```
## 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                   object
## dtype: object
```

Knowledge Check: Data types Supposed we have a pandas DataFrame named `inventory`.

1. We called `inventory.dtypes` and got the following output. Which of the column likely require type conversion because it seems to have the wrong data type? Choose all that apply.

   □ `units_instock`: int64
   □ `discount_price`: float64

□ `item_name`: object
□ `units_sold`: object

2. We would like to know the number of columns in `inventory`. Which of the following code would print the number of columns in `inventory`? Choose all that apply.

□ `print(len(inventory.columns))`
□ `print(inventory.shape[1])`
□ `print(len(rice.axes[1]))`

Indexing and Subsetting with Pandas

Using indexing operators to select, summarize or transform only a subset of data is a critical part of any data analysis workflow. Consider the following use-cases:

- Compare the sales in Year 2018 vs Year 2019

- Identify missed opportunities in a specific market segment (e.g. Retail vs Wholesale)
- Best quarter of the year to execute cross-selling promos / discounts
- Study profitability of goods in the higher price range (e.g. IDR45000000+) and how competitors positioning affect sales in that price range

Notice that in all of these use-cases, data analysts will want to use some combination of indexing and then perform the necessary computations on that specific slice or slices of data. Unsurprisingly, `pandas` come with a number of methods to help you accomplish this task.

In the following section, we'll take a closer look at some of the most common slicing and subsetting operations in `pandas`: - `head()` and `tail()`
- `select_dtypes()`
- Using `.drop()` - The `[]` operator - `.loc`
- `.iloc` - Conditional subsetting

Say we're only really interested in the numeric columns of our data, we can use `select_dtypes` to selectively include or exclude only particular data types.

In the following example, I use `select_dtypes` to *include* only textual columns (`objects`) and then proceed to pass the output of this function call into `.head()`. Notice that when we chain two methods this way, the output of the first function call will be "passed" into the second function call:

```
rice.select_dtypes(include='object').head()
```

```
##   yearmonth
## 1   2018-07
## 2   2018-07
## 3   2018-07
## 4   2018-07
## 5   2018-07
```

Change the following code from `include` to `exclude` and observe the difference in the output from our `.describe()` call:

```
rice.select_dtypes(include=['category']).describe()
```

```
##        category sub_category      format
## count     12000        12000       12000
## unique        1            1           3
## top        Rice         Rice  minimarket
## freq      12000        12000        7088
```

You can also use `include` or `exclude` with a list of data types instead of a singular value. To include all columns of data types integer and float, we can do either of these: - `include='number'`
- `include=['int', 'float']`

Try and do that now; Chain the `select_dtypes()` command with `.head()` to limit the output to only the first 5 rows:

```
## Your code below


## -- Solution code
```

Apart from using `select_dtypes` to exclude columns, we can also use `.drop()` to remove rows or columns by label names and the corresponding axis. By default, the `axis` is assumed to be 0, i.e. referring to the row. Hence the following code will drop the **row** with label `3`:

```
rice.drop(3).head()
```

```
##    receipt_id  receipts_item_id  ... quantity  yearmonth
## 1     9622257          32369294  ...        1    2018-07
## 2     9446359          31885876  ...        1    2018-07
## 4     9643416          32418582  ...        1    2018-07
## 5     9692093          32561236  ...        1    2018-07
## 6     9504155          32030785  ...        1    2018-07
##
## [5 rows x 10 columns]
```

We can drop multiple rows or columns by passing in a list. In the following code, we override the default `axis` value by passing `axis=1`; As a result `pandas` will drop the specified columns, while preserving all rows:

```
rice.drop(['unit_price', 'purchase_time', 'receipt_id'], axis=1).head()
```

```
##    receipts_item_id category sub_category  ... discount  quantity  yearmonth
## 1          32369294     Rice         Rice  ...        0         1    2018-07
## 2          31885876     Rice         Rice  ...        0         1    2018-07
## 3          31930241     Rice         Rice  ...        0         3    2018-07
## 4          32418582     Rice         Rice  ...        0         1    2018-07
## 5          32561236     Rice         Rice  ...        0         1    2018-07
##
## [5 rows x 7 columns]
```

Rather commonly, you may want to perform subsetting by slicing out a set of rows. This can be done using the `rice[start:end]` syntax, where `start` is inclusive.

The code follows slices out the first to fourth row, or equivalently, row with the index 0, 1, 2, and 3.

```
rice[0:4]
```

```
##    receipt_id  receipts_item_id  ... quantity yearmonth
## 1     9622257          32369294  ...        1   2018-07
## 2     9446359          31885876  ...        1   2018-07
## 3     9470290          31930241  ...        3   2018-07
## 4     9643416          32418582  ...        1   2018-07
##
## [4 rows x 10 columns]
```

Knowledge Check: Slicing Recalling that the `end` is not inclusive and Python's 0-based indexing behavior, if we have wanted to subset the **8th to 12th** row of our data, how would we have done it instead? Pick the right answer and try it in a new code cell below.

- □ `rice[7:12]`
- □ `rice[8:12]`
- □ `rice[7:13]`
- □ `rice[8:13]`

Using `.loc` and `.iloc`, we can perform slicing on both the row and column indices, offering us even greater flexibility and control over our subsetting operations.

`.iloc` requires us to pass an `integer` to either the row or/and column. We can also use `:` to indicate no subsetting in a certain direction. The following code slices out the first 5 rows but take all columns (pay attention to the use of the `:` operator):

```
rice.iloc[0:5, :]
```

```
##    receipt_id  receipts_item_id  ... quantity yearmonth
## 1     9622257          32369294  ...        1   2018-07
## 2     9446359          31885876  ...        1   2018-07
## 3     9470290          31930241  ...        3   2018-07
## 4     9643416          32418582  ...        1   2018-07
## 5     9692093          32561236  ...        1   2018-07
##
## [5 rows x 10 columns]
```

Putting together what you've learned so far, use `.iloc` to subset the **last 2 rows of our dataframe**, and the **first 4 columns**. If you perform this exercise correctly, the output `x` should be a `pandas` dataframe (`type(x)` is a pandas `DataFrame` object)

Bonus: If you want an extra challenge, try and perform this operation three times; 1. Use only `.iloc[ , ]`
2. Use `tail(2)` to get the last 2 rows then chain it with `.iloc`
3. Use `rice.shape[0]-2:` to get the last 2 rows in your `.iloc` operation

```
## Your code below


## -- Solution code
```

`.loc`, in contrast to `.iloc` does not subset based on *integer* but rather subset based on `label`. We can still use `integer` but our integers will be treated or interpreted as *labels*.

Let's read in the same `csv`, except this time we will set `receipt_id` as the row labels:

```
rice = pd.read_csv("data_input/rice.csv", index_col=1)
rice = rice.drop('Unnamed: 0', axis=1)
rice.head()
```

```
##             receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                     ...
## 9622257             32369294  7/22/2018 21:19  ...        1   2018-07
## 9446359             31885876  7/15/2018 16:17  ...        1   2018-07
## 9470290             31930241  7/15/2018 12:12  ...        3   2018-07
## 9643416             32418582   7/24/2018 8:27  ...        1   2018-07
## 9692093             32561236  7/26/2018 11:28  ...        1   2018-07
##
## [5 rows x 9 columns]
```

To subset for the row of transactions corresponding to receipt id 9643416 and 5735850, we can use label-based indexing (`.loc`) as such:

```
rice.loc[[9643416, 5735850], :]
```

```
##             receipts_item_id     purchase_time  ... quantity yearmonth
## receipt_id                                      ...
## 9643416             32418582    7/24/2018 8:27  ...        1   2018-07
## 5735850             17434503  12/13/2017 19:17  ...        1   2017-12
##
## [2 rows x 9 columns]
```

Dive Deeper: In the following code, we read in `companies.csv`. Take a peek at the data using `head` or `tail`.

Perform a *label*-based or *integer*-based indexing (whichever deemed more appropriate) by subsetting for the row corresponding to 'Li and Partners' and tries to answer the following questions:

1. Where is *Li and Partners* located?
2. How much `Returns` did PT. Algoritma Data Indonesia make?
3. What about the `Forecasted Growth` for Palembang Konsultansi?

```
clients = pd.read_csv("data_input/companies.csv", index_col=1)
clients.head()
```

```
##                                   ID Consulting Sales  ...  Location     Account
## Customer Name                                          ...
## New Media Group                30940       IDR7125000  ...   Jakarta  Enterprise
## Li and Partners                82391        IDR420000  ...   Jakarta     Startup
## PT. Kreasi Metrik Solusi       18374                0  ...  Surabaya  Enterprise
## PT. Algoritma Data Indonesia   57531        IDR850000  ...   Jakarta     Startup
## Palembang Konsultansi          19002       IDR2115000  ...   Bandung     Startup
##
## [5 rows x 10 columns]
## Your code below


## -- Solution code
```

Observe how in earlier exercises, you inspect the data and consciously decide whether `.loc` or `.iloc` is the more appropriate choice here. This is a helpful mental exercise to get into - in real life data analysis, very often you're faced with the dilemma of picking from a large box of tools, and knowing which method is the best fit is a critical ingredient for efficiency and fluency.

Conditional Subsetting

Along with `.iloc` and `.loc`, probably the most helpful type of subsetting would have to be conditional subsetting.

With conditional subsetting, we select data based on criteria we specified: - `.format == 'supermarket'` to select all transactions where format is supermarket
- `.unit_price >= 400000` to select all transactions with unit price being equal to or greater than 400000.
- `.quantity != 0` to select all transactions where quantity of purchase **is not** 0

We can also use the `&` and `|` operators to join conditions:

`sales[(sales.salesperson == 'Moana') & (sales.amount > 5000)]` subset any rows where Moana has sold more than $5000 worth of items

```
rice = pd.read_csv("data_input/rice.csv", index_col=1)
rice = rice.drop('Unnamed: 0', axis=1)
rice[rice.discount != 0].head()
```

```
##             receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                     ...
## 9767244             32763935   7/29/2018 8:25  ...        1   2018-07
## 9483559             31978405  7/16/2018 16:24  ...        1   2018-07
## 9841041             33010608   7/29/2018 8:21  ...        1   2018-07
## 9284943             31450853   7/6/2018 12:05  ...        1   2018-07
## 9853312             33070898  7/30/2018 13:54  ...        1   2018-07
##
## [5 rows x 9 columns]
```

Dive Deeper: In the cell below, write code using conditional subsetting and answer the following questions:

1. Say as a company, we define "bulk purchase transaction" to be any transaction containing at least 8 units of the same item (`quantity` column). In our dataset, how many "bulk purchases" do we have?
2. How many transactions in our dataset took place in a `minimarket`?
3. How many transactions in our dataset took place in a `hypermarket` and has at least 10 units (`quantity` column)?

Tip: You may find the `.shape` attribute convenient in extracting the number of rows / columns from a dataframe

```
## Your code below


## -- Solution code
```

Referencing and Copying

Another thing that you may need to note is that in Python, using `=` operator is also referencing to the initial DataFrame. For example, let's take a look back to our `rice` data:

```
rice = pd.read_csv("data_input/rice.csv", index_col=1)
rice = rice.drop('Unnamed: 0', axis=1)

rice.head()
```

```
##             receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                     ...
## 9622257             32369294  7/22/2018 21:19  ...        1   2018-07
## 9446359             31885876  7/15/2018 16:17  ...        1   2018-07
## 9470290             31930241  7/15/2018 12:12  ...        3   2018-07
## 9643416             32418582   7/24/2018 8:27  ...        1   2018-07
## 9692093             32561236  7/26/2018 11:28  ...        1   2018-07
##
## [5 rows x 9 columns]
```

Then, let's create a new dataframe, `rice_july`, which stores our rice sales data particularly in July 2018. Suppose that during that period, we're giving a 15 percent discount in all of our products, so we're also replacing all the values in `rice_july.discount` to 15.

```
rice_july = rice
rice_july['discount'] = 15
rice_july.head()
```

```
##             receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                     ...
## 9622257             32369294  7/22/2018 21:19  ...        1   2018-07
## 9446359             31885876  7/15/2018 16:17  ...        1   2018-07
## 9470290             31930241  7/15/2018 12:12  ...        3   2018-07
## 9643416             32418582   7/24/2018 8:27  ...        1   2018-07
## 9692093             32561236  7/26/2018 11:28  ...        1   2018-07
##
## [5 rows x 9 columns]
```

So far, so reasonable. However, if we were to now check the values of that Python list as referenced by `rice`, we see that the list has also been updated.

The explanation is that when we execute the line `rice_july = rice`, a new Python list **is not being created**. We truly have only one object, and that line only creates a new variable named `rice_july` that references that very same object.

The behavior with using the `=` operator in Python differs from other programming languages and can be confusing to seasoned developers new to Python.

The appropriate method instead is `.copy()`, which creates an actual copy of the python list. Notice that in the following code, there are two distinct Python objects, and changing the values in one do not affect the other:

```
rice = pd.read_csv("data_input/rice.csv", index_col=1)
rice = rice.drop('Unnamed: 0', axis=1)

rice_july = rice.copy()
rice_july['discount'] = 15
rice_july.head()
```

```
##              receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                      ...
## 9622257              32369294  7/22/2018 21:19  ...        1   2018-07
## 9446359              31885876  7/15/2018 16:17  ...        1   2018-07
## 9470290              31930241  7/15/2018 12:12  ...        3   2018-07
## 9643416              32418582   7/24/2018 8:27  ...        1   2018-07
## 9692093              32561236  7/26/2018 11:28  ...        1   2018-07
##
## [5 rows x 9 columns]
```

```
rice.head()
```

```
##              receipts_item_id    purchase_time  ... quantity yearmonth
## receipt_id                                      ...
## 9622257              32369294  7/22/2018 21:19  ...        1   2018-07
## 9446359              31885876  7/15/2018 16:17  ...        1   2018-07
## 9470290              31930241  7/15/2018 12:12  ...        3   2018-07
## 9643416              32418582   7/24/2018 8:27  ...        1   2018-07
## 9692093              32561236  7/26/2018 11:28  ...        1   2018-07
##
## [5 rows x 9 columns]
```