HyperionDev

Datasets and DataFrames

Task

Visit our website

Introduction

In this task, you will be using pandas DataFrames to manipulate data. In the context of this task, when we refer to a dataset, we are referring to a collection of related data. This data can be programmatically manipulated in various ways.

Pandas

Pandas is a Python library that contains high-level data structures and tools designed for fast and easy data analysis. Pandas also provides for explicit data alignment, which prevents common errors that result from misaligned data coming in from different sources. All this makes pandas arguably the best tool for data wrangling.

Install pandas with `pip` as follows:

```
pip install -U pandas
```

The common convention for importing pandas is as follows:

```
import pandas as pd
```

Let's start with the three fundamental pandas data structures: Series, DataFrame, and Index.

Series

A Series is a one-dimensional labelled array that can hold any data type (integers, strings, floating-point numbers, Python objects, etc.).

On creating a Series by passing a list of values, pandas creates a default integer index (**`pandas.Index`**). We can create a Series in pandas as shown below:

```
import pandas as pd
import numpy as np

s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
```

Note the use of `np.nan` to create a NaN value that is used to represent missing values or other undefined entries.

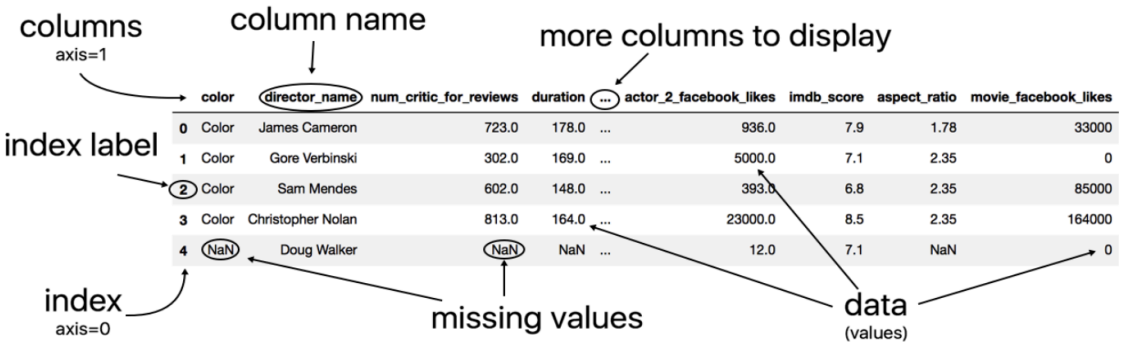
Output:

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

DataFrame

You can think of a DataFrame as a table of data with the following characteristics (Lynn, 2018):

- DataFrames can have multiple rows and columns.
- Each row corresponds to an individual data sample or observation.
- Each column represents a distinct variable or attribute that characterises the data samples.
- Data within a column is usually of the same type (e.g., numbers, text, dates).
- Unlike spreadsheets, DataFrames are typically free of missing values, ensuring data integrity and facilitating analysis.



	color	director_name	num_critic_for_reviews	duration	...	actor_2_facebook_likes	imdb_score	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	723.0	178.0	...	936.0	7.9	1.78	33000
1	Color	Gore Verbinski	302.0	169.0	...	5000.0	7.1	2.35	0
2	Color	Sam Mendes	602.0	148.0	...	393.0	6.8	2.35	85000
3	Color	Christopher Nolan	813.0	164.0	...	23000.0	8.5	2.35	164000
4	NaN	Doug Walker	NaN	NaN	...	12.0	7.1	NaN	0

The anatomy of a DataFrame (Petrou, 2017, Chapter 1)

You can create a Dataframe by passing a NumPy array. In the following example, a list of dates (datetime data type) is specified as the index, and the column labels are A, B, C, and D:

```
import pandas as pd
import numpy as np

# Generate dates for 6 different days
dates = pd.date_range('20130101', periods=6)

# Create DataFrame with dates at the index column
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
print(df)
```

Output:

```
                   A         B         C         D
2013-01-01  0.031025 -0.285793 -0.024709  0.400670
2013-01-02  0.713959 -1.356312 -0.581539 -0.591893
2013-01-03 -2.009914 -2.153678  0.092108  2.419320
2013-01-04 -1.521448 -1.475796 -0.695689 -0.282321
2013-01-05 -0.975965 -0.853425 -0.451579 -1.724022
2013-01-06 -0.241997 -0.433157  1.082101 -0.986914
```

Or, create a DataFrame by passing a dictionary of objects:

```
# Create DataFrame with a dictionary of objects
df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1,index=list(range(4)), dtype='float32'),
                    'D': np.array([3] * 4,dtype='int32'),
                    'E': pd.Categorical(["test", "train", "test", "train"]),
                    'F': 'foo'})

# Get the data types for each column
print(df2.dtypes)
```

Output:

```
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
```

Or you could load data from a .csv or txt file into a DataFrame using the `read_csv()` function, as shown below:

```
insurance_df = pd.read_csv("insurance.csv")
df = pd.read_csv('balance.txt', delim_whitespace=True)
```

Other functions can be used to read data from a variety of **different sources** into a pandas DataFrame. For example, `read_excel()` can be used to read data from an Excel spreadsheet file and `read_sql()` can be used to load data from an SQL database. Text files can also be loaded using `read_table()`. Sometimes it's easier to extract data from other sources into a .csv file and then read it into a DataFrame.

Index

The Index is a series of unique identifiers. Note how a default integer Index was created for the Series. However, you can also specify the values for the Index as we did for the DataFrame, where dates were used as an Index.

Pandas DataFrame techniques

Pandas DataFrames are the workhorses of data analysis in Python. As mentioned, these two-dimensional, table-like structures hold and organise your data. But their true power lies in the vast techniques available for manipulating that data. From wrangling messy datasets to summarising complex information, pandas offers a rich toolkit. This section explores methods for selecting specific data and performing calculations to unlock the potential of your data for further analysis and insights.

Using built-in DataFrame methods

When attempting to gain insight into your data, it's often helpful to leverage built-in methods to explore and process your data – for example, to view a sample of data or find the mean of a column.

Here is a list of common built-in methods and properties you can use to explore and process your data:

Indexing and selection

- `head()`: returns the first 5 rows, e.g., `df.head()`
- `tail()`: returns the last 5 rows, e.g., `df.tail()`
- `sample()`: returns a random sample of rows where the default is 1, e.g., `df.sample(3)`

Attributes

- `columns`: stores the column of the DataFrame, e.g., `df.columns`
- `values`: returns a NumPy array of the DataFrame, e.g., `df.values`
- `index`: stores the row labels (index) of the DataFrame, e.g., `df.index`

Computational and statistical methods

- `describe()`: returns a statistic summary for your data, e.g., `df.describe()`
- `mean()`: mean for each column
- `min()`: minimum for each column
- `max()`: maximum for each column
- `std()`: standard deviation for each column
- `var()`: variance for each column
- `nunique()`: number of unique values in each column
- `count()`: number of cells for each column or row that are not empty or undefined (e.g., NaN)
- `sum()`: sum of values for each column or row

Extra resource

Explore the **panda's documentation** for a list of all properties and methods associated with DataFrames.

Selecting columns in pandas

There are many ways to specify columns in pandas, but the simplest way is to use dictionary notation for specific columns. In essence, pandas Dataframes can be thought of as dictionaries, with the key being the column name and the value being the corresponding column values.

```
import pandas as pd
import seaborn as sns

iris_df = sns.load_dataset('iris')
print(iris_df.columns)
# Output:
# ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']

# Select only the species column
just_the_species = iris_df['species']
print(just_the_species.sample(5))
```

To select multiple columns, you simply need to specify a list of strings with each column name:

```
# Select columns with sepal and petal information
sepal_and_petal_info = iris_df[['sepal_length', 'sepal_width', 'petal_length',
                                'petal_width']]
print(sepal_and_petal_info.sample(5))
```

You can also choose specific values to be included in your search (i.e., omit certain rows from the results):

```
# Filter for specific values in a column
small_sepal_length = iris_df[iris_df['sepal_length'] < 4.8]
print(small_sepal_length.sample(5))
```

In the above example, we are filtering the dataset for all entries where the `sepal_length` is less than 4.8. Filtering or grouping data is a key part of wrangling data to prepare it for further analysis and utility.

Accessing specific records

Sometimes it's helpful to have a value that uniquely identifies a record as its index. For instance, in a database of students, the index of the observations could be a series of 1, 2, 3 … N, or it could be the student identification number. When a user searches for a record, they may input the unique identifier to get the student's records.

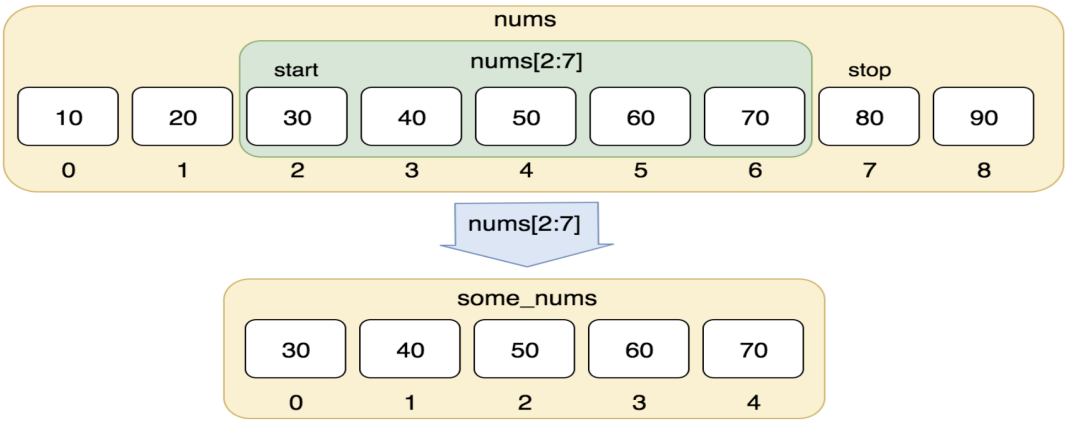
In the code example below, we ensure that the column named "Identifier" contains unique values. Next, we set this column as the DataFrame's index. Finally, the `.loc[]` property is used to access a specific row using the index value of that row.

```
# Check if all entries in "Identifier" are unique
df['Identifier'].is_unique

# Set "Identifier" column as the index
df = df.set_index('Identifier')

# Access a specific row
df.loc[76244]
```

Recall that it's possible to make a copy of a complete Python list or a subset of a list using the slice operator. For example, consider the array `nums` with nine numbers. You can get a subset of the array by indexing the range you want, e.g., `nums[2:7]`:



Slicing an array (Boiko, 2018)

This approach can also be used with a dataset:

```
# Slice records using loc
 subset_data = df.loc[1:80]
```

Extra resource

It's also possible to access columns by their label. Explore examples of how to **access DataFrame columns** using column labels or positions.

Grouping in pandas

Data analysis can sometimes get complicated, which is where more advanced functionality is needed.

Let's say you want to average the insurance charges of all people between the ages of 30 and 35. This can be done quite easily using the following:

```
# Get people in the 30-35 age group
between_30_and_35 = insurance_df[(insurance_df['age'] > 30) &
(insurance_df['age'] < 35 )]

# Print mean charges for all people in the 30-35 age group
print(between_30_and_35['charges'].mean())
```

Alternatively, you can also use the pandas **`DataFrame.query()`** method that takes boolean strings as an argument, as shown below:

```
# Use the query method to get people in the 30-35 age group
between_30_and_35 = insurance_df.query("age > 30 and age < 35")

# Print mean charges for all people in the 30-35 age group
print(between_30_and_35['charges'].mean())
```

Now let's say you want to average the insurance charges of every person in each age group. This can still be done with the syntax you know, but it will take a lot of lines of code. This is bad because we want to keep our code simple and concise. Thankfully, pandas provide us with something that allows us to do this with one line of code:

```
# Get the mean charges for each age
print(insurance_df.groupby('age')['charges'].mean())
```

This `groupby()` method tells the aggregation to work separately on each unique group specified.

Having explored the data manipulation capabilities of pandas, let's turn our attention to another tool commonly used in data science: Jupyter Notebook.

Jupyter Notebook

Jupyter Notebook is an open-source tool for creating interactive documents that combine code, visualisations, and text. It's widely used for data analysis, machine learning, and scientific computing. You will be using Jupyter Notebook in this bootcamp to execute some steps in the data science pipeline.

To use this tool, do the following:

1. **Create a virtual environment**

   A virtual environment enables you to manage Python project dependencies in isolation. Use the instructions provided in the **Additional Reading** to set up a virtual environment.

2. **Install Jupyter**

   First, check if you have installed Jupyter previously using this command:

   ```
   pip3 show jupyter
   ```

   If it's not installed, first ensure that you have the latest pip as older versions may have trouble with some dependencies:

   ```
   pip3 install --upgrade pip
   ```

   Then install Jupyter Notebook using:

   ```
   pip3 install jupyter
   ```

3. **Run Jupyter Notebook**

   Once you have installed Jupyter, you can start the Notebook server from the command line:

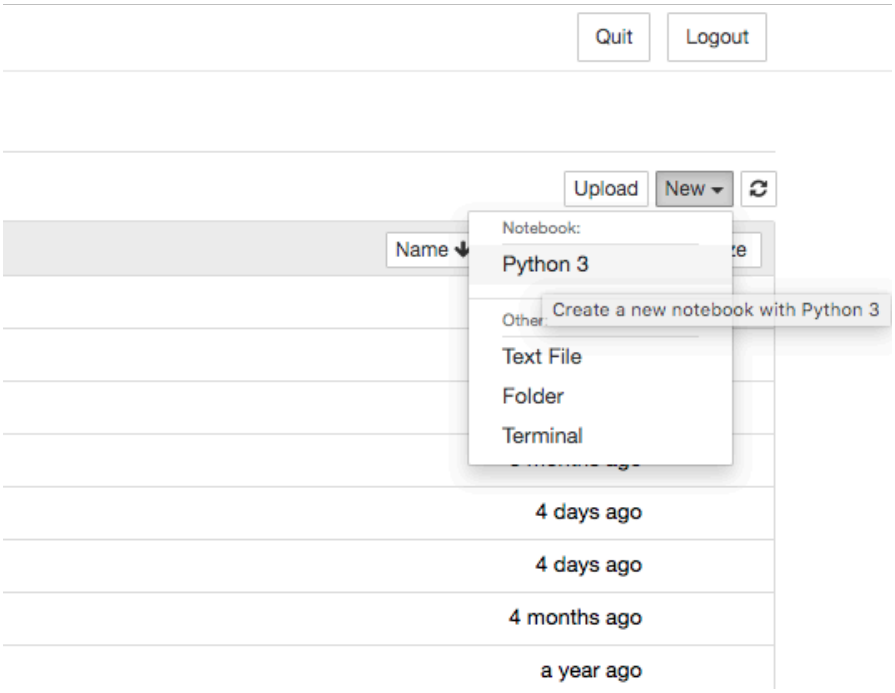
   ```
   jupyter notebook
   ```

   This will print some information about the Notebook server in your terminal, including the URL of the web application. The Notebook will then open in your browser.

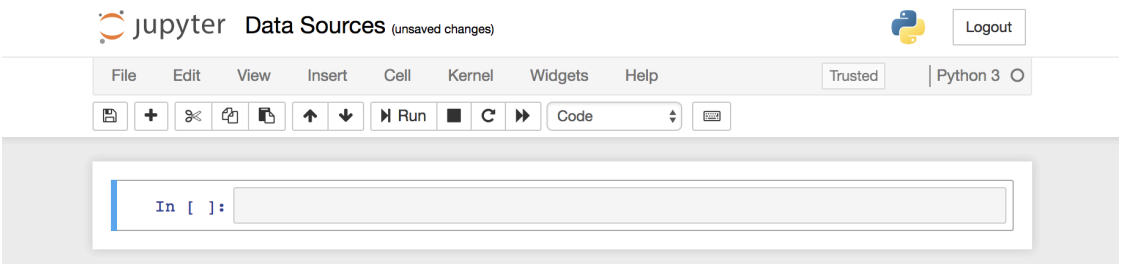
Once the Notebook has opened, you should see the dashboard showing the list of Notebooks, files, and subdirectories in the directory you've opened. You can see an example of a Jupyter Notebook below:

4. Create a new Notebook

Click the **New** dropdown menu and click on **Python 3**.

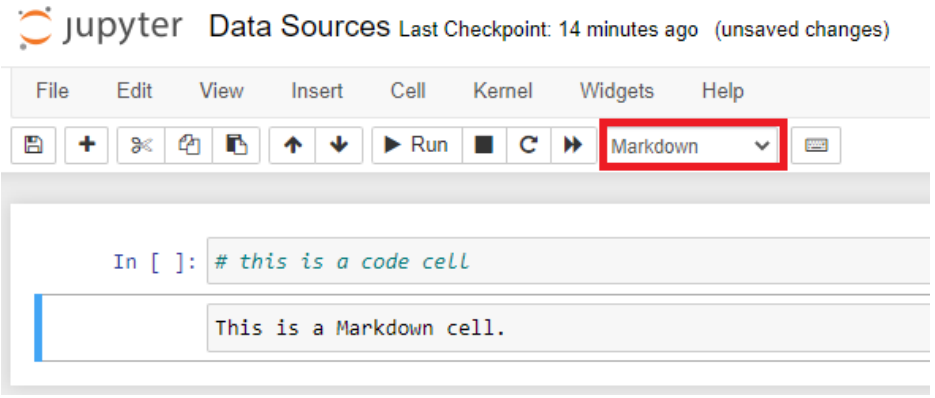


A new Jupyter Notebook will look like the screenshot below. Make sure to change the name of the Notebook. In this case, we have named the file “Data Sources”.



5. Add contextual information

In Jupyter Notebook you can specify whether a cell contains code or Markdown. **Markdown** is a lightweight Markup language that is used to embed documentation or other textual information between code cells.



Extra resource

For more information about working with Jupyter, please consult the first chapter, “**IPython: Beyond Normal Python**”, in *Python Data Science Handbook* (2016) by Jake VanderPlas.

Instructions

- Follow the instructions in this task to install Jupyter Notebook, if needed.
- In your command line interface, change the directory (cd) to the current folder.
- Open Jupyter Notebook by typing `jupyter notebook`.
- Within this task folder, you will find Jupyter Notebook examples. You can open and explore them by going to Jupyter's home screen and double-clicking on the Notebook.
    - **Dataset_Examples.ipynb** for task 1.
    - **Report_Example.ipynb** for task 2.
- You will also find a **pandas cheatsheet** that you can use as a quick guide for data-wrangling tasks in the future.

Take note

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select "Request Review", the task is automatically complete, and you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you've done that, feel free to progress to the next task.

Auto-graded task 1

Open the **Datasets_template.ipynb** file and save a copy of the notebook as **Datasets_task.ipynb**. Then, complete the following tasks in the notebook:

1. Write the code that performs the action described in the following statements:
    - Select the “Limit” and “Rating” columns of the first five observations.
    - Select the first five observations with 4 cards.
    - Sort the observations by “Education”. Show users with a high education value first.
2. Write a short explanation in the form of a comment for the following lines of code.
    - `df.iloc[:,:]`
    - `df.iloc[5:,5:]`
    - `df.iloc[:,0]`
    - `df.iloc[9,:]`

Auto-graded task 2

Follow these steps:

1. Create a copy of **Report.ipynb** and name it **Report_task.ipynb**.
2. Create a DataFrame that contains the data in **balance.txt**.
3. Write the code needed to produce a report that provides the following information:
    - Compare the average income based on ethnicity.
    - On average, do married or single people have a higher balance?
    - What is the highest income in our dataset?
    - What is the lowest income in our dataset?
    - How many cards do we have recorded in our dataset? (Hint: use **`sum()`**.)
    - How many females do we have information for vs how many males? (Hint: use **`count()`**.)

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Boiko, S. (2018, October 31). Python Indexing and Slicing for Lists, Tuples, Strings, other Sequential Types. Railsware.
https://railsware.com/blog/indexing-and-slicing-for-lists-tuples-strings-sequential-types/

Lynn, S. (2018). The pandas DataFrame – loading, editing, and viewing data in Python. *Shane Lynn*.
https://www.shanelynn.ie/using-pandas-dataframe-creating-editing-viewing-data-in-python/

Petrou, T. (2017, October 27). *Pandas Cookbook: Recipes for scientific computing, time series analysis and data visualization using Python*. Packt Publishing.