DataAnalysisWithPandas

October 7, 2024

1 Data Analysis with Pandas

Michael Fudge 2024 Edition

1.0.1 Table of Contents:

1.1 What is Pandas?

Pandas is a Python library for data analysis. The homepage of Pandas is http://pandas.pydata.org/.

Pandas takes the pain and suffering out of data analysis by doing a lot of the work for you. Pandas is so elegant at times, people feel it's "indistinguishable from magic." In this lesson we will try to our best to explain what Pandas is doing "behind the curtain" and expose the magic behind Pandas.

The Pandas library gets its name not from the furry Asian animal, but from the words "Panel Data".

1.2 What is data analysis?

Data Analysis is the process of systematically applying statistical and/or logical techniques to describe and illustrate, condense and recap, and evaluate data https://ori.hhs.gov/education/products/n_illinois_u/datamanagement/datopic.html.

The goals of data analysis are to:

1. discover useful information,
2. provide insights,
3. suggest conclusions, and
4. support decision-making.

1.2.1 Some examples

Most data analysis problems start with a fundamental question (but they don't always have to). Examples:

- Do students who study in groups perform better on examinations that those who study alone?
- What role (if any at all) does weather play in consumer shopping habits?
- What types of passengers were most likely to survive the Titantic?
- Among American Football teams who "go for it" on 4th downs, what is their win percentage?
- Are we stocking products in warehouses closest to the customers who purchase them?

1.2.2 Data analysis is "information storytelling"

Don't think of data analysis as crunching numbers and churning out graphs.

I like to think of data analysis as "information storytelling." Unlike slapping a chart on a powerpoint slide, which might be the *result* of a data analysis, it's a full disclosure of the process:

1. helping the reader understand your methodology (how you acquired and prepared the data)
2. sharing your complete analysis (including things that didn't work)
3. provding a narrative as to what the results mean, and most importantly
4. providing an honest and accurate analysis.

1.2.3 Why perform a data analysis with Code?

You **could** do this in Excel, but Code (Python) and Jupyter Notebook offer several advantages:

- It is easier to automate and update the process later on, since its code.
- You can intermix code witha narrative / explanations in the Jupyter notebook
- It can integrate with a variety of services and systems (because its code!).
- What you create is reproduceable. Someone else can run your code and see your results. This provdes stronger evidence to support your analysis as opposed to just a chart on a powerpoint slide!

Data analysis is a skill every information professional should have, and this is a primary reason we teach programming in the iSchool. When you can code it makes performing a data analysis that much better!

1.3 Part One

1.3.1 1.A Import pandas

Before you can use `pandas` you must import it into Python. When we import Pandas it is customary to use the alias `pd` so that we don't have to type `pandas` in our code. For example instead of typing `pandas.DataFrame()` we can save ourselves a few keystrokes with `pd.DataFrame()`

np.nan In most cases we will need `np.nan` from the `numpy` module, too. This is a special value which indicates the absense of a value. (nan means "Not a Number") Since these values are absent they are not included in numerical calculations. This is important for getting the "math" right, as we will see later on.

[1]:

```
import pandas as pd
import numpy as np

print(np.nan)
```

```
nan
```

1.3.2 1.B The Basic Elements of Pandas: Series

Series The **series** is the most basic structure in Pandas. It is simply a named-list of values. Every series has a data type, expressed as a numpy data type. Common data types you will see in this course are `int` or `float64` for numbers, and `Object` for string types.

The following example creates a series called `grades` we use Python's named arguments feature to set the data, name and data type (dtype) arguments. Some of the grades are missing so we use `np.nan` in those cases.

[2]:

```
grades = pd.Series(data = [100,80,100,90,np.nan,70], name = "Grade")
grades
```

[2]:

```
0    100.0
1     80.0
2    100.0
3     90.0
4      NaN
5     70.0
Name: Grade, dtype: float64
```

Index What's with those numbers to the left of the grades? That's the **index**, and every series has one. An index is an ordered list of values. It's how Pandas accesses specific values in a Series (or as we will see in a bit… a DataFrame)

The index of the series works a lot like the index in a list or a string. For example, this code prints the first and last grades in the series.

[3]:

```
print("first grade:", grades[0])
print("last grade:", grades[5])
```

```
first grade: 100.0
last grade: 70.0
```

Series Aggregate Functions The beauty of Pandas is that for most operations, you won't even need a loop! For example, we derive the `min()`, `max()`, `mean()`, `sum()` and `count()` of non `np.nan` values in a series without a for loop!!!

We do this by calling those method functions on the series itself, for example:

[4]:

```
print("Highest grade:", grades.max())
print("Average grade:", grades.mean())
print("lowest grade:", grades.min())
print("Sum of grades:", grades.sum())
print("Count of grades", grades.count())
```

```
Highest grade: 100.0
Average grade: 88.0
lowest grade: 70.0
Sum of grades: 440.0
Count of grades 5
```

Other Series Functions You can also get value counts for a series and also deduplicate values in a series. For example consider the following series:

[5]:

```
votes = pd.Series(data=[ 'y','y','y','n','y',np.nan,'n','n','y'], name="Vote")
votes
```

[5]:

```
0      y
1      y
2      y
3      n
4      y
5    NaN
6      n
7      n
8      y
Name: Vote, dtype: object
```

We use the `unique()` method function to return only the non-duplicate values from the series.

The `value_counts()` method function adds up values, creating a new series where the index is the value and the value is the count.

[6]:

```
print("deduplicate the votes:", votes.unique())
print("counts by value:", votes.value_counts())
```

```
deduplicate the votes: ['y' 'n' nan]
counts by value: Vote
y    5
n    3
Name: count, dtype: int64
```

Where did all these functions come from? They're methods (object functions) associated with the Series object. If you dir(grades) you can see them!

If you want to learn more, here's the official documentation: https://pandas.pydata.org/pandas-docs/stable/reference/series.html

1.3.3 1.C Basic Elements of Pandas: DataFrame

The pandas **DataFrame** is a table representation of data. It is the primary use case for pandas itself. A dataframe is simply a collection of **Series** that share a common **Index**. I like to think of the DataFrame as a programmable spreadsheet. It has rows and columns which can be accessed and manipulated with Python.

`DataFrame` is the most common Pandas data structure. As you'll see its expressive and versitile, making it an essential tool in data analysis.

To make a `DataFrame` we must create a Python `dict` of Series, or `list` . We use the series name as the key and the series itself as the value.

This example creates a DataFrame from two series of Student names and Grade-Point-Averages (GPA's), using Series:

[7]:
```
names = pd.Series( data = ['Allen','Bob','Chris','Dave','Ed','Frank','Gus'])
gpas = pd.Series( data = [4.0, np.nan, 3.4, 2.8, 2.5, 3.8, 3.0])
years = pd.Series( data = ['So', 'Fr', 'Fr', 'Jr', 'Sr', 'Sr', 'Fr'])
series_dict = { 'Name':  names, 'GPA': gpas, 'Year' : years }  # dict of␣
  ↪Series, keys are the series names
students = pd.DataFrame( series_dict )
students
```

[7]:
```
    Name  GPA Year
0  Allen  4.0   So
1    Bob  NaN   Fr
2  Chris  3.4   Fr
3   Dave  2.8   Jr
4     Ed  2.5   Sr
5  Frank  3.8   Sr
6    Gus  3.0   Fr
```

Here's the same code but we make the dataframe from simple Python lists. The end result is the same, as the DataFrame is always constructed from Series. In this case the `pf.DataFrame()` method created the three `Series` for us.

[8]:
```
data_dict = {
    'Name':  ['Allen','Bob','Chris','Dave','Ed','Frank','Gus'],
    'GPA': [4.0, np.nan, 3.4, 2.8, 2.5, 3.8, 3.0],
    'Year' : ['So', 'Fr', 'Fr', 'Jr', 'Sr', 'Sr', 'Fr'] }
students = pd.DataFrame( data_dict )
students
```

[8]:
```
    Name  GPA Year
0  Allen  4.0   So
1    Bob  NaN   Fr
2  Chris  3.4   Fr
3   Dave  2.8   Jr
4     Ed  2.5   Sr
```

```
5  Frank  3.8   Sr
6    Gus  3.0   Fr
```

Getting a series from the dataframe You can access a series from the dataframe using the series name. For example this gets the lowest GPA by using the `min()` function on the `students['GPA']` series:

```
[9]: print("Lowest GPA:", students['GPA'].min())
```

```
Lowest GPA: 2.5
```

Getting the column names You can use the `columns` property to return an iterable of the columns in the dataframe.

```
[10]: print("Columns in the students data frame are: ", students.columns)
```

```
Columns in the students data frame are:  Index(['Name', 'GPA', 'Year'],
dtype='object')
```

1.3.4 1.D Accessing elements with loc and iloc

The `loc` and `iloc` properties allow you to slice the dataframe. `loc` uses the index and column names, while `iloc` uses ordinal positions starting at zero.

Here are some examples:

```
[11]: # Examples using loc
print("loc: Get the name of the student at index 3: ", students.loc[3, 'Name'])
print("loc: Get the Year of the last student: ", students.loc[6, 'Year'])

# Same examples using iloc
print("iloc: Get the name of the student at index 3: ", students.iloc[3, 0])
print("iloc: Get the Year of the last student: ", students.iloc[-1, 2])
```

```
loc: Get the name of the student at index 3:  Dave
loc: Get the Year of the last student:  Fr
iloc: Get the name of the student at index 3:  Dave
iloc: Get the Year of the last student:  Fr
```

You can also use `loc` and `iloc` to retrieve a subset dataframe. For example, here are the names and GPA's of the first 4 students:

```
[12]: students.loc[0:3, 'Name':'GPA'] #uses the index and column names
```

```
[12]:     Name  GPA
      0  Allen  4.0
      1    Bob  NaN
      2  Chris  3.4
      3   Dave  2.8
```

[13]:
```
students.iloc[0:4, 0:2] # uses relative places, zero-based.
```

[13]:
```
    Name  GPA
0  Allen  4.0
1    Bob  NaN
2  Chris  3.4
3   Dave  2.8
```

[16]:
```
students[ students.GPA.isna()]
```

[16]:
```
  Name  GPA Year
1  Bob  NaN   Fr
```

1.3.5 1.E Basic DataFrame operations

In most cases, you will not construct a DataFrame as we have done in the examples so far. Instead you will load data from a file into a DataFrame. pandas supports a wide variety of file formats, which we will demonstrate in a later section, for now let's load a dataset of customers from the internet and use dataframe operations to explore the data.

This example read in a text file in CSV format into the DataFrame. There are 30 customers.

[14]:
```
customers = pd.read_csv('https://raw.githubusercontent.com/mafudge/datasets/
 ↪master/customers/customers.csv')
len(customers)
```

[14]:
```
30
```

Getting information about your DataFrame First thing you are likely to do with a loaded dataframe it get some basic information about the data. These methods will help:

- info() provide names of columns, counts of non-null values in each columns, and data types.
- describe() for each numerical column provide some basic statistics (min, max, mean, and quartiles).

[15]:
```
customers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   First            30 non-null     object
 1   Last             30 non-null     object
 2   Email            30 non-null     object
 3   Gender           30 non-null     object
 4   Last IP Address  30 non-null     object
 5   City             30 non-null     object
 6   State           30 non-null     object
 7   Total Orders    30 non-null     int64
 8   Total Purchased  30 non-null     int64
 9   Months Customer  30 non-null     int64
dtypes: int64(3), object(7)
memory usage: 2.5+ KB
```

[16]:
```
customers.describe()
```

[16]:
```
       Total Orders  Total Purchased  Months Customer
count     30.000000        30.000000        30.000000
mean       4.633333      1193.166667        15.133333
std        4.295012      1685.396507        15.734288
min        0.000000         0.000000         0.000000
25%        1.000000        68.750000         2.000000
50%        3.500000       500.000000         8.000000
75%        7.750000      1168.750000        27.750000
max       14.000000      6090.000000        42.000000
```

Viewing Data in your DataFrame Most likely the data in your DataFrame will be too large to view on the screen. There are methods to help with this:

- `head(n=5)` view the FIRST `n` rows in the dataframe (defaults to 5)
- `tail(n=5)` view the LAST `n` rows in the dataframe (defaults to 5)
- `sample(n=1)` view a random `n` rows from the dataframe (defautls to 1)

[17]:
```
customers.sample(n=3)
```

[17]:
```
    First    Last                Email Gender Last IP Address       City  \
24    Sal    Ladd  sladd@superrito.com      M   23.112.202.16  Rochester
7   Candi   Cayne      ccayne@rhyta.com      F     24.39.14.15   Portland
18   Mike  Rofone    mrofone@dayrep.com      M    23.224.160.4   Cheyenne

   State  Total Orders  Total Purchased  Months Customer
24    NY            14              594               10
7     ME             1              620                2
18    WY             0                0                0
```

[18]:
```
customers.head(n=3)
```

[18]:
```
   First    Last               Email Gender Last IP Address      City State  \
0     Al  Fresco  afresco@dayrep.com      M   74.111.18.161  Syracuse    NY
1   Abby    Kuss     akuss@rhyta.com      F   23.80.125.101   Phoenix    AZ
2  Arial   Photo   aphoto@dayrep.com      F      24.0.14.56    Newark    NJ

   Total Orders  Total Purchased  Months Customer
0             1               45                1
1             1               25                2
```

```
2             1              680                 1
```

[19]:

```
customers.tail(n=3)
```

[19]:

```
     First   Last                  Email Gender Last IP Address       City  \
27   Tally  Itupp  titupp@superrito.com      F    24.38.114.105  Sea Cliff
28     Tim   Pani   tpani@superrito.com      M    23.84.132.226    Buffalo
29  Victor   Rhee      vrhee@einrot.com      M   23.112.232.160  Green Bay

   State  Total Orders  Total Purchased  Months Customer
27    NY            11              380               42
28    NY             0                0                1
29    WI             0                0                2
```

1.3.6 1.F Selecting Rows and Columns: selectors and boolean indexes

In this final section of part one, we lean how to select rows and columns from the DataFrame using the selector operator `[]`

[20]:

```
students
```

[20]:

```
    Name  GPA Year
0  Allen  4.0   So
1    Bob  NaN   Fr
2  Chris  3.4   Fr
3   Dave  2.8   Jr
4     Ed  2.5   Sr
5  Frank  3.8   Sr
6    Gus  3.0   Fr
```

Selecting Columns To select columns from the dataframe, include a list of columns you would like to include inside the selector.

For example this includes the `Year` and `Name` columns from the `students` DataFrame.

[21]:

```
students[['Year', 'Name']]
```

[21]:

```
  Year   Name
0   So  Allen
1   Fr    Bob
2   Fr  Chris
3   Jr   Dave
4   Sr     Ed
5   Sr  Frank
6   Fr    Gus
```

Here another example where we include the name, email city and state from the `customers` dataframe. Since there are a lot of customers we use `head()`to show the first 5.

```
[22]: customers.columns
```

```
[22]: Index(['First', 'Last', 'Email', 'Gender', 'Last IP Address', 'City', 'State',
             'Total Orders', 'Total Purchased', 'Months Customer'],
            dtype='object')
```

```
[23]: customers[['First', 'Last', 'Email', 'City', 'State']].head(n=5)
```

```
[23]:    First    Last                  Email      City State
      0     Al  Fresco     afresco@dayrep.com  Syracuse    NY
      1   Abby    Kuss        akuss@rhyta.com   Phoenix    AZ
      2  Arial   Photo      aphoto@dayrep.com    Newark    NJ
      3  Bette   Alott       balott@rhyta.com   Raleigh    NC
      4   Barb  Barion  bbarion@superrito.com    Dallas    TX
```

Selecting A single column: DataFrame or Series? **When selecting a single column**, you have a decision to make: do you want a `Series` or a `DataFrame`?

- For a `Series` just include the column name in the selector: `'Email'`
- For a `DataFrame` include the column name as a list in the selector: `['Email']`

Again, in this example, `sample()` will be used to keep the size down!

```
[24]: # Series
      customers.sample(5)['Email']
```

```
[24]: 6       bmelator@einrot.com
      14        jpoole@dayrep.com
      9           crha@einrot.com
      18       mrofone@dayrep.com
      15     lhvmeehom@einrot.com
      Name: Email, dtype: object
```

```
[25]: # DataFrame
      customers.sample(5)[['Email']]
```

```
[25]:                     Email
      26       tanott@rhyta.com
      10     ddelyons@dayrep.com
      17      mmelator@rhyta.com
      4   bbarion@superrito.com
      0      afresco@dayrep.com
```

Selecting Rows: Boolean Index In pandas, a **boolean index** is a `Series` of type `bool` based on the result of some boolean expression.

We can then use the `pandas` selector `[]` to apply the boolean index to the DataFrame returning only rows where the boolean index is `True`

```
[26]: students
```

```
[26]:     Name  GPA Year
      0  Allen  4.0   So
      1    Bob  NaN   Fr
      2  Chris  3.4   Fr
      3   Dave  2.8   Jr
      4     Ed  2.5   Sr
      5  Frank  3.8   Sr
      6    Gus  3.0   Fr
```

For example this boolean index evaluates to `True` when the value in the `'Year'` series is equal to `'Fr'`.

```
[27]: students['Year'] == 'Fr'
```

```
[27]: 0    False
      1     True
      2     True
      3    False
      4    False
      5    False
      6     True
      Name: Year, dtype: bool
```

To filter the dataframe based on the boolean index, we include it in the dataframe selector, for example, this only shows the Freshmen `Year == 'Fr'`

```
[28]: # show Freshmen
      students[students['Year'] == 'Fr']
```

```
[28]:     Name  GPA Year
      1    Bob  NaN   Fr
      2  Chris  3.4   Fr
      6    Gus  3.0   Fr
```

1.3.7 Part One Summary

In this part we learned the basics of `pandas`. We learned how to aggregate series, build dataframes, and use the dataframe selector.

Let's put it all together to perform a data analysis.

Challenge: Get the name and gpa of the Senior student with the lowest gpa.

Algorithm:

1. filter students to seniors only
2. find min() GPA from seniors
3. filter the seniors to only display the GPA that equals the min GPA.

[29]:
```
seniors = students[ students['Year']=='Sr']
seniors
```

[29]:
```
    Name  GPA Year
4     Ed  2.5   Sr
5  Frank  3.8   Sr
```

[30]:
```
lowest_gpa_sr = seniors['GPA'].min()
lowest_gpa_sr
```

[30]:
```
2.5
```

[31]:
```
person = seniors[['Name','GPA']][seniors['GPA']==lowest_gpa_sr]
person
```

[31]:
```
  Name  GPA
4   Ed  2.5
```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

1.4 Part Two

1.4.1 2.A Iterating over a DataFrame Or Series

Sometimes its necessary to loop over your DataFrame. For example when plotting points on on a map, you will need to build the points by iterating over the values in the dataframe.

Looping over the DataFrame To loop over a dataframe use the `iterrows()` method function. This provides an `int` index and a `dict` for the values in each column. The key in the `dict` is the column name and the value is the row value at that index.

[32]:
```
students
```

[32]:
```
    Name  GPA Year
0  Allen  4.0   So
1    Bob  NaN   Fr
2  Chris  3.4   Fr
3   Dave  2.8   Jr
4     Ed  2.5   Sr
5  Frank  3.8   Sr
6    Gus  3.0   Fr
```

[33]:

```
for index, row in students.iterrows():
    print(f"{index}: {row['Name']} {row['GPA']} {row['Year']}")
```

```
0: Allen 4.0 So
1: Bob nan Fr
2: Chris 3.4 Fr
3: Dave 2.8 Jr
4: Ed 2.5 Sr
5: Frank 3.8 Sr
6: Gus 3.0 Fr
```

Likewise iterating over a series is similar. The `Series` itself is iterable so there is no need to use a method function.

[34]:

```
for name in students['Name']:
    print(name)
```

```
Allen
Bob
Chris
Dave
Ed
Frank
Gus
```

1.4.2 2.B Loading JSON with Pandas

The Pandas `read_json()` method can easily read in list-oriented JSON-formatted text into a DataFrame. Each item in the list becomes a row in the DataFrame.

For example, here's the `students.json` file we read in with Pandas. Notice the JSON starts with a list `[`

```
[
    {"Name":"Abby", "GPA": 4.0, "Year": "So"},
    {"Name":"Bette", "GPA": 3.7, "Year": "Jr"},
    {"Name":"Chris", "Year": "Fr"},
    {"Name":"Dee", "GPA": 3.4, "Year": "Fr"}
]
```

[35]:

```
students = pd.read_json("https://raw.githubusercontent.com/mafudge/datasets/
  ↪master/json-samples/students.json")
students
```

[35]:

```
    Name  GPA Year
0   Abby  4.0   So
1  Bette  3.7   Jr
2  Chris  NaN   Fr
3    Dee  3.4   Fr
```

json_normalize() for nested JSON The `read_json()` method does not perform well on nested JSON structures. For example consider the following JSON file of customer orders:

The file `orders.json`:

```
[
    {
        "Customer" : { "FirstName" : "Abby", "LastName" : "Kuss"},
        "Items" : [
            { "Name" : "T-Shirt", "Price" : 10.0, "Quantity" : 3},
            { "Name" : "Jacket", "Price" : 20.0, "Quantity" : 1}
        ]
    },
    {
        "Customer" : { "FirstName" : "Bette", "LastName" : "Alott"},
        "Items" : [
            { "Name" : "Shoes", "Price" : 25.0, "Quantity" : 1},
            { "Name" : "Jacket", "Price" : 20.0, "Quantity" : 1}
        ]
    },
    {
        "Customer" : { "FirstName" : "Chris", "LastName" : "Peanugget"},
        "Items" : [
            { "Name" : "T-Shirt", "Price" : 10.0, "Quantity" : 1}
        ]
    }
]
```

When we read this with `read_json_()` we get the three orders but only two columns.

[36]:
```
orders = pd.read_json("https://raw.githubusercontent.com/mafudge/datasets/
 ↪master/json-samples/orders.json")
orders
```

[36]:
```
                                          Customer  \
0       {'FirstName': 'Abby', 'LastName': 'Kuss'}
1     {'FirstName': 'Bette', 'LastName': 'Alott'}
2  {'FirstName': 'Chris', 'LastName': 'Peanugget'}

                                               Items
0  [{'Name': 'T-Shirt', 'Price': 10.0, 'Quantity'…
1  [{'Name': 'Shoes', 'Price': 25.0, 'Quantity': …
2  [{'Name': 'T-Shirt', 'Price': 10.0, 'Quantity'…
```

What we want is one row per item on the the order and the customer name to be in separate columns. the `json_normalize()` method can help here.

It is important to note that `json_normalize()` does not take a file as input, but rather de-serialized json.

```
[37]: import json
      with open("orders.json") as f:
          json_data = json.load(f)  #de-serialize

      orders = pd.json_normalize(json_data)
      orders
```

```
[37]:                                            Items Customer.FirstName  \
      0  [{'Name': 'T-Shirt', 'Price': 10.0, 'Quantity'…               Abby
      1  [{'Name': 'Shoes', 'Price': 25.0, 'Quantity': …              Bette
      2  [{'Name': 'T-Shirt', 'Price': 10.0, 'Quantity'…              Chris

        Customer.LastName
      0              Kuss
      1             Alott
      2         Peanugget
```

Better but we still need to handle the list of `Items`. To accomplish this we

1. set the `record_path` to be the nested list `'Items'`. This tells `json_normalize()` to use that JSON key as the row level. So now we will have 5 rows (one for each item) instead of 3.
2. Then we set the `meta` named argument to a `list` of each of the other values we wish to include, in this instance last name and first name.

NOTE: The `meta` syntax is a bit weird. Its a `list` of JSON paths (also represented as lists) to each item in the JSON. For example:

```
The meta Argument         ==> Matches This in the JSON            ==> And Displays As This Pandas
["Customer","FirstName"] ==> { "Customer" : { "FirstName": ...} ==> Customer.Firstname
```

```
[38]: orders = pd.json_normalize(json_data, record_path="Items",␣
       ↪meta=[["Customer","FirstName"],["Customer","LastName"]])
      orders
```

```
[38]:       Name  Price  Quantity Customer.FirstName Customer.LastName
      0  T-Shirt   10.0         3               Abby              Kuss
      1   Jacket   20.0         1               Abby              Kuss
      2    Shoes   25.0         1              Bette             Alott
      3   Jacket   20.0         1              Bette             Alott
      4  T-Shirt   10.0         1              Chris         Peanugget
```

Yes it seems complicated, because it is a bit complicated, let’s try another example, with some abstract values.

In the following example we want to generate a normalized table with 3 rows and 4 columns.

- The rows are based on the “A” `record_path`. There are three: 101, 111 and 201
- The meta data are based on columns “B”, and “C1”

[39]:

```
json_data = [
    {
        "A": [
            {"A1": 101, "A2": 102},
            {"A1": 111, "A2": 112}
        ],
        "B": 103,
        "C": {"C1": 104}
    },
    {
        "A": [
            {"A1": 201, "A2": 202}
        ],
        "B": 203,
        "C": {"C1": 204}
    }
]
```

[40]:

```
df = pd.json_normalize(json_data, record_path="A", meta=["B", ["C", "C1"]])
df
```

[40]:

```
    A1   A2    B C.C1
0  101  102  103  104
1  111  112  103  104
2  201  202  203  204
```

1.4.3 2.C Simple Web scraping with Pandas

The Pandas `read_html()` method can be used to extact HTML tables from websites. For any given webpage url as input, `read_html()` will output a `list` of DataDrames. There will be one DataFrame for each table on the website.

In this example we read the following wikipedia page: https://en.wikipedia.org/wiki/National_Basketball_Associa

NOTE: We use wayback machine to snapshot the page to the point in time this paper was written.

[41]:

```
dfs = pd.read_html("https://web.archive.org/web/20240316160100/https://en.
 ↪wikipedia.org/wiki/National_Basketball_Association")
```

Upon inspection the NBA teams are the 4th table in the list.

[42]:

```
teams = dfs[3]
teams.head(n=5)
```

[42]:

```
           Conference  Division                Team  \
0  Eastern Conference  Atlantic      Boston Celtics
1  Eastern Conference  Atlantic       Brooklyn Nets
2  Eastern Conference  Atlantic     New York Knicks
3  Eastern Conference  Atlantic  Philadelphia 76ers
```

```
4  Eastern Conference  Atlantic     Toronto Raptors

                     Location                 Arena  Capacity  \
0       Boston, Massachusetts             TD Garden     19156
1          New York, New York       Barclays Center     17732
2          New York, New York  Madison Square Garden     19812
3  Philadelphia, Pennsylvania     Wells Fargo Center     20478
4            Toronto, Ontario       Scotiabank Arena     19800

                                           Coordinates Founded Joined
0  42°21 59 N 71°03 44 W / 42.366303°N 71.062228°W    1946   1946
1   40°40 58 N 73°58 29 W / 40.68265°N 73.974689°W   1967*   1976
2  40°45 02 N 73°59 37 W / 40.750556°N 73.993611°W    1946   1946
3  39°54 04 N 75°10 19 W / 39.901111°N 75.171944°W   1946*   1949
4  43°38 36 N 79°22 45 W / 43.643333°N 79.379167°W    1995   1995
```

A table of championships can be found as the 6th table.

```
[43]: championships = dfs[5]
      championships.head(n=5)
```

```
[43]:                                            Teams  Win  Loss  Total  \
      0                    Minneapolis/Los Angeles Lakers   17    15     32
      1                                   Boston Celtics   17     5     22
      2  Philadelphia/San Francisco/Golden State Warriors    7     5     12
      3                                     Chicago Bulls    6     0      6
      4                                 San Antonio Spurs    5     1      6

                                           Year(s) won  \
      0  1949, 1950, 1952, 1953, 1954, 1972, 1980, 1982…
      1  1957, 1959, 1960, 1961, 1962, 1963, 1964, 1965…
      2           1947, 1956, 1975, 2015, 2017, 2018, 2022
      3                 1991, 1992, 1993, 1996, 1997, 1998
      4                       1999, 2003, 2005, 2007, 2014

                                      Year(s) runner-up
      0  1959, 1962, 1963, 1965, 1966, 1968, 1969, 1970…
      1                       1958, 1985, 1987, 2010, 2022
      2                       1948, 1964, 1967, 2016, 2019
      3                                                  -
      4                                               2013
```

You'll have to investigate the output to find what you need There is no magic command you can type to find the table you want. Ultimately you will need to inspect each DataFrame returned from `read_html()` and figure out which DataFrame in the list contains the table of data you need.

Here's some sample code to list every table on the webpage along with its index. We simple use a `for` loop to iterate over each DataFrame. Code like this can help you to identify the table you want from the webpage.

[44]:
```
from IPython.display import display

webpage = "https://web.archive.org/web/20230606035218/https://ist256.com/"
dataframes = pd.read_html(webpage)
for index, df in enumerate(dataframes):
    print("INDEX:", index)
    display(df.head(5))
```

```
INDEX: 0

      0     1     2
0   Feb   JUN   Dec
1   NaN    06   NaN
2  2022  2023  2024

INDEX: 1

         Dates Topic (Click Link for Content and Assigned Readings)
0  1/18 - 1/22                        Wednesday is the first class.
1  1/23 - 1/29      Lesson 01: Introduction to Python Programming
2   1/30 - 2/5      Lesson 02: Input, Output, Variables and Types
3   2/6 - 2/12                           Lesson 03: Conditionals
4  2/13 - 2/19                             Lesson 04: Iterations

INDEX: 2

    Date Due  Time Due Gradebook  Points        Tool  \
0  1/24/2023  11:59 PM       L01       3  Jupyterhub
1  1/25/2023  11:59 PM       S01       3  Jupyterhub
2  1/27/2023  11:59 PM       H01       3  Jupyterhub
3  1/31/2023  11:59 PM       L02       3  Jupyterhub
4   2/1/2023  11:59 PM       S02       3  Jupyterhub

                               What is Due?
0                  01-Intro/LAB-Intro.ipynb
1           01-Intro/SmallGroup-Intro.ipynb
2                   01-Intro/HW-Intro.ipynb
3          02-Variables/LAB-Variables.ipynb
4  02-Variables/SmallGroup-Variables.ipynb
```

1.4.4 2.D Advanced Dataframe Operations

In this section we explore the various options for combining one or more dataframes together. We'll cover the two most common method functions:

- `concat()` - Row-oriented. Combine one or more DataFrames together. Appropriate for similar sets of data. Online students + Campus students.

- `merge()` - Column-oriented. Combine one or more DataFrames based on a matching column. Appropriate for different sets of data that share a business rule. Students *take* Classes so we merge students + classes so we know who is taking which class.

Concat() Example. The common use case for `concat()` is to combine two separate but similar data sets into a single data set.

[45]:
```
campus_students = pd.read_csv("https://raw.githubusercontent.com/mafudge/
↪datasets/master/delimited/campus-students.csv")
campus_students
```

[45]:
```
    Name  Grade       Year
0  Helen    NaN  Sophomore
1   Iris   10.0     Senior
2  Jimmy    8.0   Freshman
3  Karen    NaN   Freshman
4  Lynne   10.0  Sophomore
5   Mike   10.0  Sophomore
6   Nico    NaN     Junior
7   Pete    8.0   Freshman
```

[46]:
```
online_students = pd.read_csv("https://raw.githubusercontent.com/mafudge/
↪datasets/master/delimited/online-students.csv")
online_students
```

[46]:
```
    Name  Grade       Year Location
0   Abby    7.0   Freshman       NY
1    Bob    9.0  Sophomore       CA
2  Chris   10.0     Senior       CA
3   Dave    8.0   Freshman       NY
4  Ellen    7.0  Sophomore       TX
5   Fran   10.0     Senior       FL
6   Greg    8.0   Freshman       NY
```

To perform the `concat()` we provide a `list` of DataFrames to concatenate.

[47]:
```
students = pd.concat([campus_students, online_students])
students
```

[47]:
```
    Name  Grade       Year Location
0  Helen    NaN  Sophomore      NaN
1   Iris   10.0     Senior      NaN
2  Jimmy    8.0   Freshman      NaN
3  Karen    NaN   Freshman      NaN
4  Lynne   10.0  Sophomore      NaN
5   Mike   10.0  Sophomore      NaN
6   Nico    NaN     Junior      NaN
7   Pete    8.0   Freshman      NaN
```

```
0   Abby   7.0   Freshman       NY
1    Bob   9.0  Sophomore       CA
2  Chris  10.0     Senior       CA
3   Dave   8.0   Freshman       NY
4  Ellen   7.0  Sophomore       TX
5   Fran  10.0     Senior       FL
6   Greg   8.0   Freshman       NY
```

concat() - Ignoring the index As you can see from the code above the index from the original DataFrames was used. For example `Helen` and `Abby` both share the index `0`. While this is acceptable, there are situations where a new index based on combined values is desirable. To make this happen include the `ignore_index=True` named argument. This will create a new index from the output DataFrame.

[48]:

```
students = pd.concat([campus_students, online_students], ignore_index=True)
students
```

[48]:

```
     Name  Grade       Year Location
0   Helen    NaN  Sophomore      NaN
1    Iris   10.0     Senior      NaN
2   Jimmy    8.0   Freshman      NaN
3   Karen    NaN   Freshman      NaN
4   Lynne   10.0  Sophomore      NaN
5    Mike   10.0  Sophomore      NaN
6    Nico    NaN     Junior      NaN
7    Pete    8.0   Freshman      NaN
8    Abby    7.0   Freshman       NY
9     Bob    9.0  Sophomore       CA
10  Chris   10.0     Senior       CA
11   Dave    8.0   Freshman       NY
12  Ellen    7.0  Sophomore       TX
13   Fran   10.0     Senior       FL
14   Greg    8.0   Freshman       NY
```

merge() Example. The common use case for `merge()` is to combine two separate but dissimilar data sets into a single data set. These DataFrames **share a common column** so you can match the output of each DataFrame to the other on matching rows.

[49]:

```
customers = pd.read_csv("https://raw.githubusercontent.com/mafudge/datasets/
  ↪master/delimited/customers.csv")
customers
```

[49]:

```
   customer_id firstname   lastname
0           10      Abby       Kuss
1           20     Bette      Alott
2           30     Chris  Peanugget
```

```
3           40        Don     Atello
```

```
[50]: orders = pd.read_csv("https://raw.githubusercontent.com/mafudge/datasets/master/
       ↪delimited/orders.csv")
      orders
```

```
[50]:    order_id  order_customer_id     item  price  qty
      0      1001                 10  T-Shirt     10    3
      1      1002                 10   Jacket     20    1
      2      1003                 20    Shoes     25    1
      3      1004                 20   Jacket     20    1
      4      1005                 30  T-Shirt     10    1
```

For the `customers` and `orders` DataFrames we want to combine them together so that each customer is matched to their order. In this case the common column in both DataFrames is `customer_id` (from `customers`) and `order_customer_id` (from `orders`).

To perform the merge we need at least 4 named arguments are as follows:

- `left` the DataFrame on the left.
- `right` the DataFrame on the right.
- `left_on` - the column from the `left` dataframe to match
- `right_on` - the column from the `right` dataframe to match

```
[51]: combined = pd.merge(left=customers, right=orders, left_on="customer_id",␣
       ↪right_on="order_customer_id")
      combined
```

```
[51]:    customer_id  firstname   lastname  order_id  order_customer_id     item  \
      0           10       Abby       Kuss      1001                 10  T-Shirt
      1           10       Abby       Kuss      1002                 10   Jacket
      2           20      Bette      Alott      1003                 20    Shoes
      3           20      Bette      Alott      1004                 20   Jacket
      4           30      Chris  Peanugget      1005                 30  T-Shirt

         price  qty
      0     10    3
      1     20    1
      2     25    1
      3     20    1
      4     10    1
```

Left and Right are relative Of course left and right are relative, as we need to know which dataframe we are referring to as part of the merge. For example this is the same merge:

```
[52]: combined = pd.merge(left=orders, right=customers, left_on="order_customer_id",␣
       ↪right_on="customer_id")
      combined
```

```
[52]:    order_id  order_customer_id     item  price  qty  customer_id  firstname  \
      0      1001                 10  T-Shirt     10    3           10       Abby
      1      1002                 10   Jacket     20    1           10       Abby
      2      1003                 20    Shoes     25    1           20      Bette
      3      1004                 20   Jacket     20    1           20      Bette
      4      1005                 30  T-Shirt     10    1           30      Chris

          lastname
      0       Kuss
      1       Kuss
      2      Alott
      3      Alott
      4  Peanugget
```

1.4.5 2.E Generating Columns, Lambdas and the apply() function

In this section we demonstrate how to derive new columns in the DataFrame from the existing data.

```
[53]: data_dict = {
          'Name':  ['Allen','Bob','Chris','Dave','Ed','Frank','Gus'],
          'GPA': [4.0, np.nan, 3.4, 2.8, 2.5, 3.8, 3.0],
          'Year' : ['So', 'Fr', 'Fr', 'Jr', 'Sr', 'Sr', 'Fr'] }
      students = pd.DataFrame( data_dict )
      students
```

```
[53]:     Name  GPA Year
      0  Allen  4.0   So
      1    Bob  NaN   Fr
      2  Chris  3.4   Fr
      3   Dave  2.8   Jr
      4     Ed  2.5   Sr
      5  Frank  3.8   Sr
      6    Gus  3.0   Fr
```

Column generation Pandas allows us to create new columns from existing data. This is simple to do when the values are constants or simple formulas.

This adds a column "MaxGPA" with the set value of 4.0 for every value in the series.

```
[54]: students["MaxGPA"] = 4.0
      students
```

```
[54]:     Name  GPA Year  MaxGPA
      0  Allen  4.0   So     4.0
      1    Bob  NaN   Fr     4.0
      2  Chris  3.4   Fr     4.0
      3   Dave  2.8   Jr     4.0
```

```
4     Ed  2.5   Sr     4.0
5  Frank  3.8   Sr     4.0
6    Gus  3.0   Fr     4.0
```

This creates a calculated column "DiffGPA" which is the difference between two columns:

[55]:
```
students["DiffGPA"] = students["MaxGPA"] - students["GPA"]
students
```

[55]:
```
    Name  GPA Year  MaxGPA  DiffGPA
0  Allen  4.0   So     4.0      0.0
1    Bob  NaN   Fr     4.0      NaN
2  Chris  3.4   Fr     4.0      0.6
3   Dave  2.8   Jr     4.0      1.2
4     Ed  2.5   Sr     4.0      1.5
5  Frank  3.8   Sr     4.0      0.2
6    Gus  3.0   Fr     4.0      1.0
```

Lambda Functions For more complex logic, a user-defined function can be created and then executed for each value using the `apply()` method function.

For example, consider the following function code to calculate whether an input GPA is on the deans list >=3.4:

[56]:
```
def deans_list(gpa: float) -> str:
    if gpa >= 3.4:
        return "Yes"
    else:
        return "No"

deans_list(3.4)
```

[56]:
```
'Yes'
```

Here's how we use the function with the `apply()` method to create a new Pandas column, breaking it down:

- `apply()` operates on a DataFrame, in this case `students`.
- `apply()` returns a Series, which we add to the column `"DeansList"`
- `lambda row:` is a way of saying "do this for each row"
- `row` becomes an iterator of `dict` of values in each row. For example in the first row (index=0) `row['Name'] == 'Allen'` and `row['GPA'] == 4.0`
- `axis=1` tells `apply()` to loop over rows. This is almost always what we want.

[57]:
```
students['DeansList'] = students.apply(lambda row: deans_list(row['GPA']),␣
 ↪axis=1)
students
```

```
[57]:     Name  GPA Year  MaxGPA  DiffGPA DeansList
      0  Allen  4.0   So     4.0      0.0       Yes
      1    Bob  NaN   Fr     4.0      NaN        No
      2  Chris  3.4   Fr     4.0      0.6       Yes
      3   Dave  2.8   Jr     4.0      1.2        No
      4     Ed  2.5   Sr     4.0      1.5        No
      5  Frank  3.8   Sr     4.0      0.2       Yes
      6    Gus  3.0   Fr     4.0      1.0        No
```

1.4.6 2.F DataFrames as output

Throughout most of this tutorial we have been echoing the DataFrame content to the Jupyter cell. In this section we discuss your options for rendering the DataFrame as output when part of a larger program.

[58]:

```python
data_dict = {
    'Name':  ['Allen','Bob','Chris','Dave','Ed','Frank','Gus'],
    'GPA': [4.0, np.nan, 3.4, 2.8, 2.5, 3.8, 3.0],
    'Year' : ['So', 'Fr', 'Fr', 'Jr', 'Sr', 'Sr', 'Fr'] }
students = pd.DataFrame( data_dict )
```

Printing To output a dataframe in code, you can `print()` it. You'll get a text-based version of the dataframe. For example:

[59]:

```python
size = int(input("How many students to display at random? "))
random_students = students.sample(n=size)
print(random_students)
```

```
How many students to display at random?  4

    Name  GPA Year
1    Bob  NaN   Fr
5  Frank  3.8   Sr
2  Chris  3.4   Fr
3   Dave  2.8   Jr
```

IPython.display To get a well-formatted dataframe you can pass it in as an argument to `IPython.display`. The same example:

[60]:

```python
from IPython.display import display
from ipywidgets import interact_manual
max_students = len(students)
@interact_manual(size=(1,max_students,1))
def on_click(size):
    random_students = students.sample(n=size)
    display(random_students)
```

```
interactive(children=(IntSlider(value=4, description='size', max=7, min=1),␣
 ↪Button(description='Run Interact',…
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```