Pandas Review

Pandas is a Python package for organizing and analyzing data. In one sense, it is a generalization of NumPy, on which it is based.

NumPy is fantastic for working with numerical data that are "well behaved". For example, if you are analyzing data from a tightly controlled laboratory experiment, then NumPy might be perfect.

In the broader world of behavioral data science, however, data can be complicated. Variables can be of multiple types, values can be missing, etc. Pandas was developed to make it easier for us to work with data sets in general, not just numerical arrays.

If you have experience in R then, in a nutshell, pandas gives you an equivalent to R in Python (some data scientists use both, picking one or the other depending on the project, but most people prefer sticking with one language if they can).

Pandas Data

The main data object in pandas is the `DataFrame`. It is a table of data in which each column has a name, generally corresponding to a specific real-world variable.

Just as we can think about a NumPy array as a spatial layout of a Python list of lists, we can think of a pandas `DataFrame` as a spatial layout of a Python dictionary.

Consider the following Python dictionary:

In [1]:
```
dis_chars = {'name': ['Mickey', 'Minnie', 'Pluto'],
             'gender': ['m', 'f', 'n'],
             'age': [95, 95, 93],}
```

In [2]:
```
dis_chars
```

Out[2]:
```
{'name': ['Mickey', 'Minnie', 'Pluto'],
 'gender': ['m', 'f', 'n'],
 'age': [95, 95, 93]}
```

On the one hand, this is a nice organized *container* of data. But on the other hand, it is not much else. If we wanted to compute anything, like the mean age of all non-male characters, we'd have to start writing code from scratch.

Let's make our dictionary into a `DataFrame`. First, we'll import pandas.

In [3]:
```
import pandas as pd
```

Importing pandas as `pd` is conventional, like importing numpy as `np`, so there's no reason to do anything else.

Now we can convert our data to a `DataFrame` using `pd.DataFrame()`.

In [4]:
```
dis_df = pd.DataFrame(dis_chars)
```

And let's look at our new creation!

In [5]:
```
dis_df
```

Out[5]:

	name	gender	age
0	Mickey	m	95
1	Minnie	f	95
2	Pluto	n	93

Now we have a nice organized table of data, in which each column corresponds to a variable, and can be referred to by name.

In [6]:
```
dis_df['name']
```

Out[6]:
```
0    Mickey
1    Minnie
2     Pluto
Name: name, dtype: object
```

Further, it makes it relatively easy for us to do lots of analyses "out of the box". For example:

In [7]:
```
dis_df['age'].mean()
```

Out[7]:
```
94.33333333333333
```

Here, we just grabbed a column of data by name (`dis_df['age']`), and then computed its mean with the built-in `mean()` method.

The `DataFrame` isn't the only type of object in pandas, but it's the biggie. If you have experience in R, then you'll be in familiar territory, because the `DataFrame` in Python is modeled after the data frame (or tibble) in R.

In [8]:
```
type(dis_df)
```

Out[8]:
```
pandas.core.frame.DataFrame
```

Each column of a `DataFrame` is a pandas `Series`.

In [9]:
```
dis_age_s = dis_df['age']
dis_age_s
```

Out[9]:
```
0    95
1    95
2    93
Name: age, dtype: int64
```

In [10]:
```
type(dis_age_s)
```

Out[10]:
```
pandas.core.series.Series
```

And each series is a collection of more fundamental objects. So if we look at the last age in our series...

In [11]:
```
a = dis_age_s[2]
a
```

Out[11]:
```
93
```

And check the type...

In [12]:
```
type(a)
```

Out[12]:
```
numpy.int64
```

We see that it is a numpy integer; a hint that pandas is indeed built from NumPy!

If we check the type of one of the other values:

In [13]:
```
type(dis_df['gender'][2])
```

Out[13]:
```
str
```

We see that it is a Python string object. (Take a moment to dissect that line of code, and see how it is doing exactly the same thing as we did to get the type of an age value, just in one go.)

In the code cell below, get the very first name in our Disney `DataFrame`.

In [14]:
```
# At first, Mickey's name was going to be Mortimer Mouse. I know, right?
dis_df['name'][0]
```

Out[14]:
```
'Mickey'
```

One great thing about pandas is that, if we want to add a column, we just act like it already exists and assign values to it. Like this:

In [15]:
```
dis_df['wearsBow'] = [False, True, False]
dis_df
```

Out[15]:

	name	gender	age	wearsBow
0	Mickey	m	95	False
1	Minnie	f	95	True
2	Pluto	n	93	False

Notice that we are addressing a 'wearsBow' column just like we would an existing column such as 'name'. Pandas, rather than complain and be annoying, just creates the column for us!

Data i/o (Input and Output)

One of the really great things about pandas is that it makes reading, inspecting, and writing data files in common formats very easy.

Importing (input)

Following the pandas documentation, let's look at some data about the passengers on the RMS Titanic.

Download the titanic.csv and place in folder named 'data' that is in the same folder as you have this notebook.

Now, loading it is as easy as calling `pd.read_csv()`:

In [16]:
```
In [2]: titanic = pd.read_csv("data/titanic.csv")
```

There are lots of other formats that pandas can read, including excel and html.

It can even read data from the clipboard! Try it! Go to the Wikepedia page for Austin, scroll to the demographics section, and select the three columns (including the headers) down to 2020, and copy them to your clipboard.

Now run the code below.

In [20]:
```
atx_pop = pd.read_clipboard()
```

In [21]:
```
atx_pop
```

Out[21]:

	Census	Pop.	Note	%±
0	1850	629	NaN	—
1	1860	3,494	NaN	455.5%
2	1870	4,428	NaN	26.7%
3	1880	11,013	NaN	148.7%
4	1890	14,575	NaN	32.3%
5	1900	22,258	NaN	52.7%
6	1910	29,860	NaN	34.2%
7	1920	34,876	NaN	16.8%
8	1930	53,120	NaN	52.3%
9	1940	87,930	NaN	65.5%
10	1950	132,459	NaN	50.6%
11	1960	186,545	NaN	40.8%
12	1970	253,539	NaN	35.9%
13	1980	345,890	NaN	36.4%
14	1990	465,622	NaN	34.6%
15	2000	656,562	NaN	41.0%
16	2010	790,390	NaN	20.4%
17	2020	961,855	NaN	21.7%

Inspecting

It's important to peek at any imported data to make sure nothing looks funny (like we just did with the Austin population data). So let's peek at the RMS Titanic data.

In [19]:
```
titanic
```

Out[19]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450
...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376

891 rows × 12 columns

A nice thing about pandas `DataFrames` is that, by default, they show you their first and last 5 rows (their head and tail), and then tell you how big they are (891x12 in this case).

We can look at as much of the head or tail as we want with the `head()` and `tail()` methods.

In [22]:
```
titanic.tail(9)
```

Out[22]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticke
882	883	0	3	Dahlberg, Miss. Gerda Ulrika	female	22.0	0	0	755
883	884	0	2	Banfield, Mr. Frederick James	male	28.0	0	0	C.A./SOTO 3406
884	885	0	3	Sutehall, Mr. Henry Jr	male	25.0	0	0	SOTON/O 39207
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5	38265
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	21153
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	11205
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 660
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	11136
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	37037

Use the cell below to display the first 11 rows of the titanic data.

In [25]:
```
# but these rows go to 11...
titanic.head(11)
```

Out[25]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	237736
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1	1	PP 9549

We can also look at the data types:

In [26]:
```
titanic.dtypes
```

Out[26]:
```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

(the columns listed as "object" seem to be strings)

We can also get more detailed information using the `info()` method:

In [27]:
```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

This gives us a nice summary of the types of data in the columns and, in particular, how many valid (non-missing) values are in each. We can see that "Cabin", for example, has many missing values.

Exporting (output)

The `to_` methods, such as `to_csv()`, `to_excel()`, etc., allow us to export data in many ways. As an example, let's export the titanic data as a Microsoft Excel file.

In the cell below, use `titanic.to_excel(...)` to export the data to an Excel spreadsheet.

In [29]:
```
# exporting an Excel file!
titanic.to_excel("/Users/zzanescott/Desktop/titanic.xlsx")
```

Open the file in Excel to verify that the export worked.

Selecting Data

In numpy, we select data by primarily by row and column indexes. In pandas, we generally address columns (corresponding to real world variables) by *name* and rows by one or more *criteria*.

Getting columns

As we did above with our little toy Disney data, we can compute the mean age of the passengers by grabbing that column of data by name, and then computing the mean of it.

In [30]:
```
ages = titanic['Age']
ages.mean()
```

Out[30]:
```
29.69911764705882
```

In the cell below, compute the mean age in one line of code (i.e., not creating the temporary 'age' object).

In [31]:
```
# average age of passengers on the RMS Titanic
titanic['Age'].mean()
```

Out[31]:
```
29.69911764705882
```

We can get multiple columns by indexing our `DataFrame` with a Python list of column names. We can do this in two lines for readability.

In [32]:
```
wanted_cols = ['Fare', 'Survived']
fare_surv = titanic[wanted_cols]
```

In [33]:
```
fare_surv
```

Out[33]:

	Fare	Survived
0	7.2500	0
1	71.2833	1
2	7.9250	1
3	53.1000	1
4	8.0500	0
...	...	...
886	13.0000	0
887	30.0000	1
888	23.4500	0
889	30.0000	1
890	7.7500	0

891 rows × 2 columns

But more commonly we do it in a single line.

In [34]:
```
fare_surv = titanic[['Fare', 'Survived']]
```

Your initial reaction might be "Why the double brackets? Why not single brackets?", and the reason should be clear if we look back at the two line example: the `DataFrame` expects a Python list, not separate strings. So the outer set of brackets are indexing brackets, and the inner set defines a Python list.

Getting rows

We generally extract rows of interest by placing one or more criterea on a particular column.

In [35]:
```
my_critereon = fare_surv['Fare'] > 20
rich = fare_surv[my_critereon]
```

What is actually happening here is that the logical test `fare_surv['Fare'] > 20` is creating a pandas series that is `True` for the rows in which the fare paid was greater than 20 pounds sterling, and `False` otherwise.

Let's look at `my_critereon`:

In [36]:
```
my_critereon
```

Out[36]:
```
0      False
1       True
2      False
3       True
4      False
       ...
886    False
887     True
888     True
889     True
890    False
Name: Fare, Length: 891, dtype: bool
```

This series is then used to get all the rows of `fare_surv` that correspond to the `True` values, and these are placed in `rich`.

This is known as ***logical indexing***, and is widely used in data analysis!

As with fetching columns, we can do this one line instead of two.

In [37]:
```
rich = fare_surv[fare_surv['Fare'] > 20]
```

Whether you make a separate indexing series like `my_critereon` or put the test inside the indexing brackets is up to you. For simple tests, putting the test inside the brackets doesn't hurt the readability of the code at all. For more complicated tests – if you wanted all the cases of female passengers that paid between 20 and 50 lbs. for their fare, and had no siblings and two parents aboard, say – then you might want to make the test series first, and then do the indexing.

In the cell below, get the passenger class (`Pclass`) and survival status of passengers that paid more than 20 pounds for their voyage.

In [38]:
```
# passenger class and survival of high fares
titanic[titanic["Fare"] > 20][["Pclass", "Survived"]]
```

Out[38]:

	Pclass	Survived
1	1	1
3	1	1
6	1	0
7	3	0
9	2	1
...	...	...
880	2	1
885	3	0
887	1	1
888	3	0
889	1	1

376 rows × 2 columns

Now fetch the same for passengers that paid 20 pounds or less for their voyage.

In [39]:
```
# passenger class and survival of low fares
titanic[titanic["Fare"] <= 20][["Pclass", "Survived"]]
```

Out[39]:

	Pclass	Survived
0	3	0
2	3	1
4	3	0
5	3	0
8	3	1
...	...	...
882	3	0
883	2	0
884	3	0
886	2	0
890	3	0

515 rows × 2 columns

Finally, get the class and survival status for passengers that paid either less than 10 lbs. **or** more than 50 lbs. for their fare.

In [40]:
```
# ppl paying a little *or* a lot
titanic.loc[(titanic["Fare"] < 10) | (titanic["Fare"] > 50),["Pclass", "Surv
```

Out[40]:

	Pclass	Survived
0	3	0
1	1	1
2	3	1
3	1	1
4	3	0
...	...	...
878	3	0
879	1	1
881	3	0
884	3	0
890	3	0

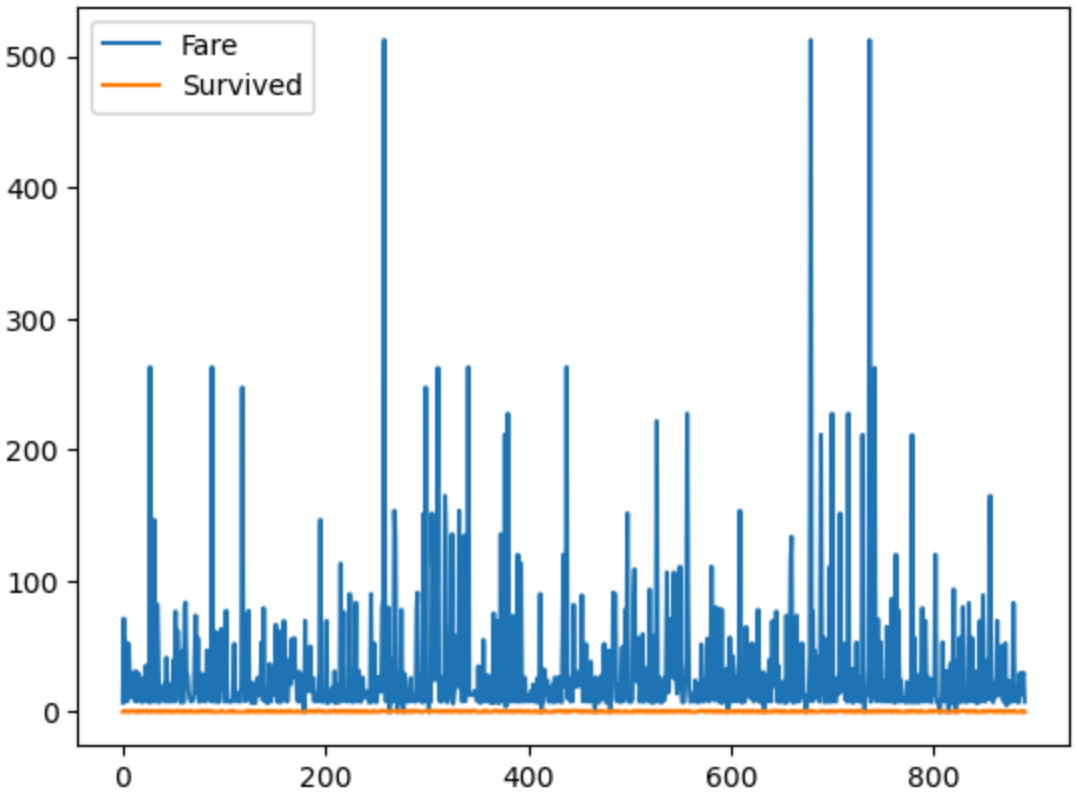
496 rows × 2 columns

If you did the above in two steps, see if you can do it in one go instead! There are hints just above.

Basic Plotting

`DataFrame` objects know how to plot themselves! Or, more precisely, `DataFrame` objects have methods for plotting. Let's try!

In [41]:
```
import matplotlib.pyplot as plt
fare_surv.plot();
```



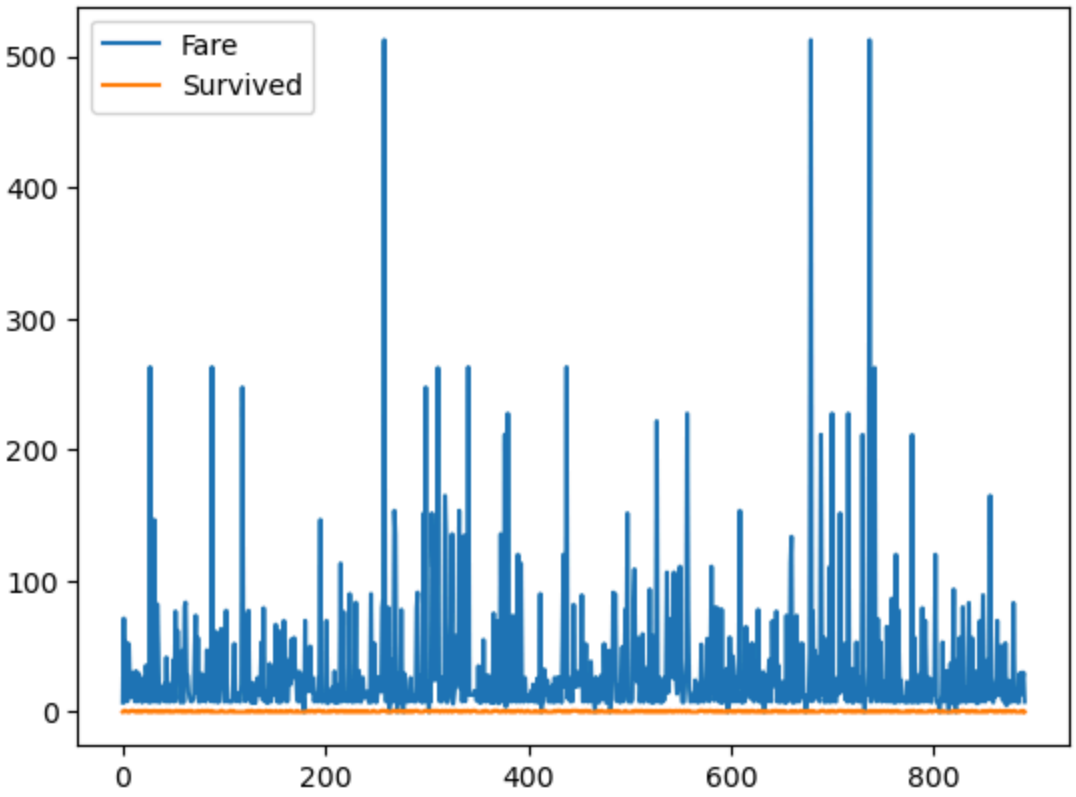
As a graph, this one isn't very informative, but it does show us what the default `DataFrame.plot()` method does: it plots (numerical) data by row index. This could be quite useful if a data frame were sorted on a particular variable...

Other type of plots are reached through plot, like `fare_surv.plot.scatter()` or similar. We can see what methods are available by hitting the <TAB> key after `DataFrame.plot.`

Do this below;

In [47]:
```
fare_surv.plot()
```

Out[47]:
```
<Axes: >
```

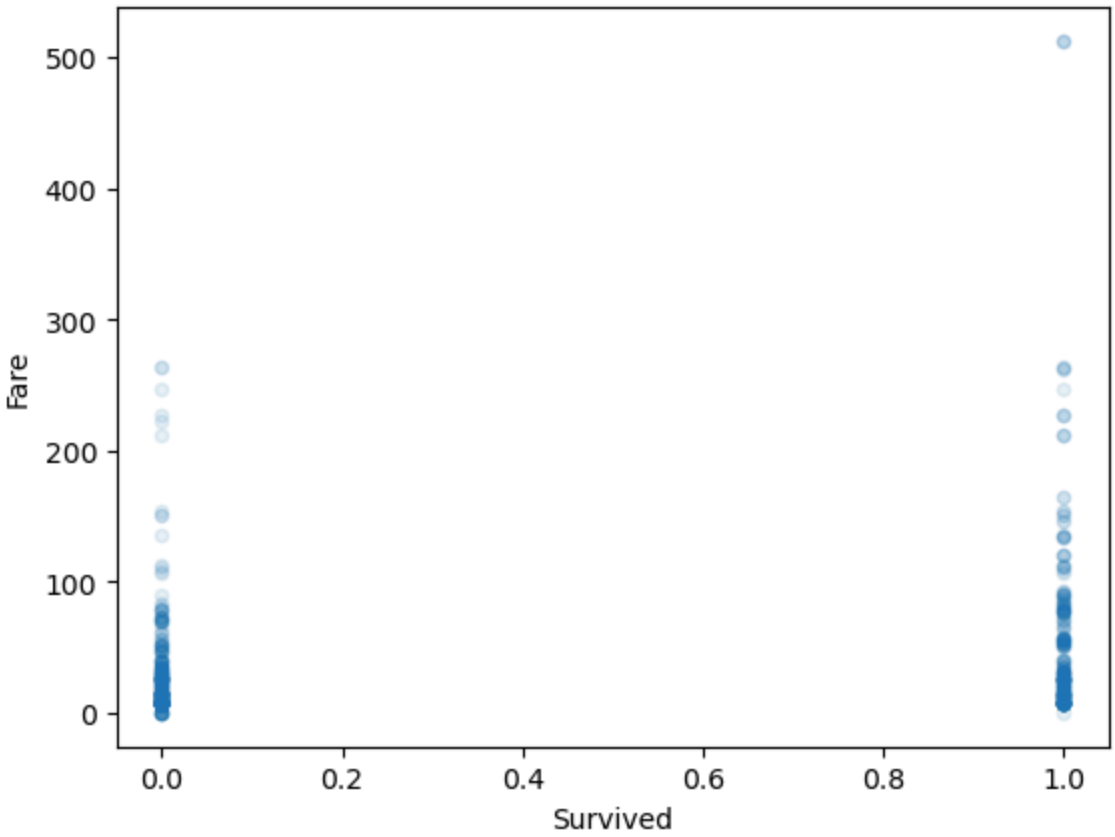


So there *is* a `scatter()` available, along with many of our other `matplotlib` friends.

Let's try a scatter plot Fare vs. Survival.

In [52]:
```
fare_surv.plot.scatter(x="Survived", y="Fare", alpha = 0.1)
```

Out[52]:
```
<Axes: xlabel='Survived', ylabel='Fare'>
```

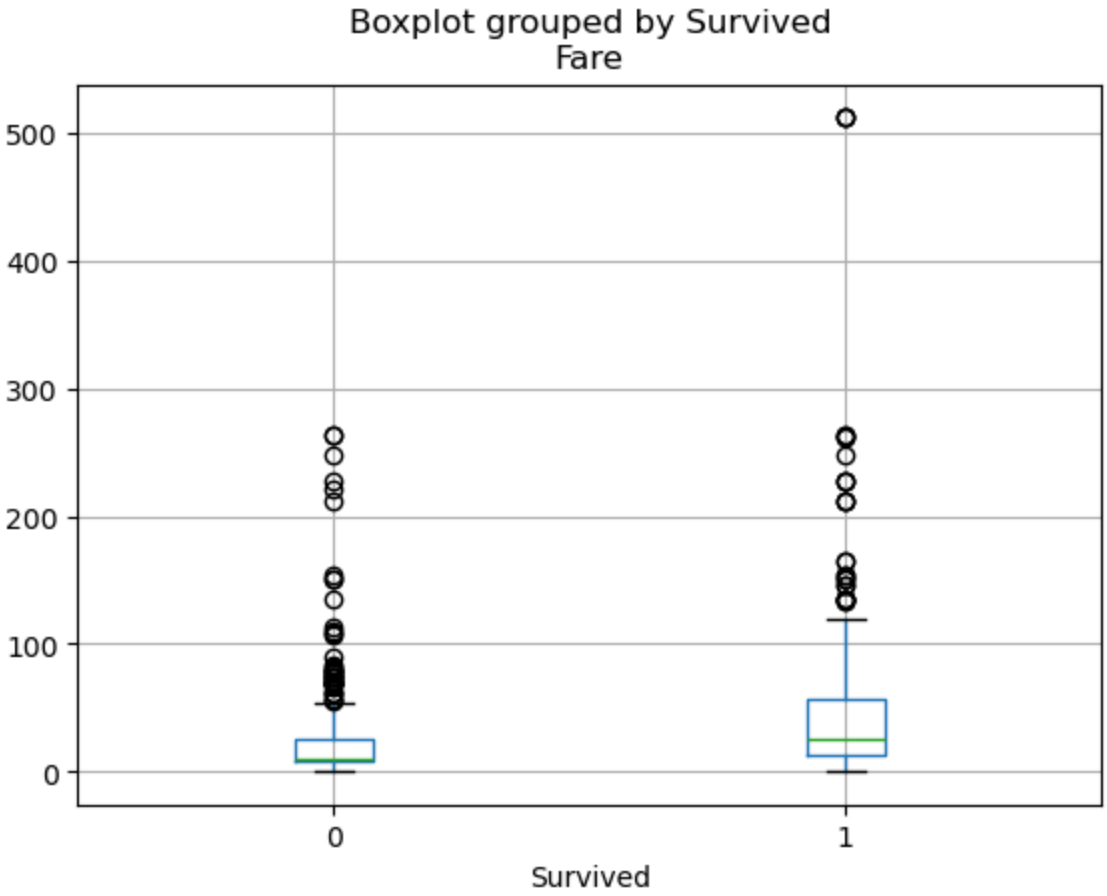


Looks like those 500 lb. fares were worth it.

Use the cell below to make a box plot of the ***column*** `Fare` ***by*** the variable `Survived`.

In [53]:
```
# boxplot of Fare paid by Survival status
titanic.boxplot(column="Fare", by="Survived")
```

Out[53]:
```
<Axes: title={'center': 'Fare'}, xlabel='Survived'>
```



Calculating New Columns

We often want to compute new columns based on existing ones. Pandas makes this really easy!

Let's use numpy to make a toy data set of annual wages and income-from-interest for 10 people.

In [54]:
```
import numpy as np
```

In following code, you should be able to understand the numpy bit up top. The pandas bit further down should sort of make sense, but don't worry if you don't fully understand it. You can come back and look at it again after you've finished this tutorial.

In [55]:
```
# make some incomes in thousands of US dollars
rng = np.random.default_rng(seed=42)
raw_dat = rng.integers(0,100,size=(10, 2))
raw_dat[:,0] = raw_dat[:,0] + 100
raw_dat[4,1] = raw_dat[4,1] + 200

# make initial column names
col_names = ['wage', 'interest']
```

```
# make the initial pandas data frame
incomes = pd.DataFrame(raw_dat, columns = col_names)

# add a gender column
gender = ['f', 'm', 'n', 'f', 'f', 'n', 'm', 'm', 'f', 'f']
incomes['gender'] = gender

# look at our new data frame
incomes
```

Out[55]:

	wage	interest	gender
0	108	77	f
1	165	43	m
2	143	85	n
3	108	69	f
4	120	209	f
5	152	97	n
6	173	76	m
7	171	78	m
8	151	12	f
9	183	45	f

One obvious thing to look at from a behavioral science perspective would be total income. After all, money is money...

So we'll make a new column for total income, and set it to the sum of the wage and interest columns. To do this, we address our desired column as though it already exists, and make it equal to what we want (the sum of wage and interest income, in this case).

In [56]:
```
incomes['total'] = incomes['wage'] + incomes['interest']
incomes
```

Out[56]:

	wage	interest	gender	total
0	108	77	f	185
1	165	43	m	208
2	143	85	n	228
3	108	69	f	177
4	120	209	f	329
5	152	97	n	249
6	173	76	m	249
7	171	78	m	249
8	151	12	f	163
9	183	45	f	228

*All of the arithmetic** and **logical** operators can be used to create new columns based on existing ones.*

We can also use scaler multipliers or addends, etc. (like we did when we created the raw data with numpy just above). The scaler will be "broadcast" to each element of the column.

For example, if we wanted to know the total income in Euros, we could do this:

In [57]:
```
dol2eu = 0.94 # 0.94 euros per US dollar (early 2023)
incomes['total_eu'] = dol2eu * incomes['total']
incomes
```

Out[57]:

	wage	interest	gender	total	total_eu
0	108	77	f	185	173.90
1	165	43	m	208	195.52
2	143	85	n	228	214.32
3	108	69	f	177	166.38
4	120	209	f	329	309.26
5	152	97	n	249	234.06
6	173	76	m	249	234.06
7	171	78	m	249	234.06
8	151	12	f	163	153.22
9	183	45	f	228	214.32

In the cell below, add a Boolean (True/False) column that shows if each person's wages exceeds their income from interest.

In [59]:
```
# adding a wages vs incomes comparison column
incomes["wage_exceeds_interest"] = incomes["wage"] > incomes["interest"]
incomes
```

Out[59]:

	wage	interest	gender	total	total_eu	wage_exceeds_interest
0	108	77	f	185	173.90	True
1	165	43	m	208	195.52	True
2	143	85	n	228	214.32	True
3	108	69	f	177	166.38	True
4	120	209	f	329	309.26	False
5	152	97	n	249	234.06	True
6	173	76	m	249	234.06	True
7	171	78	m	249	234.06	True
8	151	12	f	163	153.22	True
9	183	45	f	228	214.32	True

Summary Statistics

Getting summary statistics is also something that pandas makes really easy.

Simple descriptive statistics

We can get a quick look an entire `DataFrame` with its `describe()` method (similar to `summary()` in R).

In [60]:
```
incomes.describe()
```

Out[60]:

	wage	interest	total	total_eu
count	10.000000	10.000000	10.000000	10.000000
mean	147.400000	79.100000	226.500000	212.910000
std	27.281251	51.977452	47.815037	44.946135
min	108.000000	12.000000	163.000000	153.220000
25%	125.750000	51.000000	190.750000	179.305000
50%	151.500000	76.500000	228.000000	214.320000
75%	169.500000	83.250000	249.000000	234.060000
max	183.000000	209.000000	329.000000	309.260000

Notice that `describe()` handled the presence of a string column gracefully by ignoring it rather than producing an error.

If we hit the <TAB> key after `incomes.`, we'll see that `DataFrame` objects have a LOT of methods!

In [62]:
```
incomes.count()
```

Out[62]:
```
wage                    10
interest                10
gender                  10
total                   10
total_eu                10
wage_exceeds_interest   10
dtype: int64
```

If we browse around a little, we see that all the common summary statistics like mean, median, standard deviation, etc. are there, and they all have reasonable names. Let's compute the mean

In [63]:
```
incomes.mean()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[63], line 1
----> 1 incomes.mean()

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:11335,
in DataFrame.mean(self, axis, skipna, numeric_only, **kwargs)
  11327 @doc(make_doc("mean", ndim=2))
  11328 def mean(
  11329     self,
   (...)
  11333     **kwargs,
  11334 ):
> 11335     result = super().mean(axis, skipna, numeric_only, **kwargs)
  11336     if isinstance(result, Series):
  11337         result = result.__finalize__(self, method="mean")

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/generic.py:1199
2, in NDFrame.mean(self, axis, skipna, numeric_only, **kwargs)
  11985 def mean(
  11986     self,
  11987     axis: Axis | None = 0,
   (...)
  11990     **kwargs,
  11991 ) -> Series | float:
> 11992     return self._stat_function(
  11993         "mean", nanops.nanmean, axis, skipna, numeric_only, **kwargs
  11994     )

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/generic.py:1194
9, in NDFrame._stat_function(self, name, func, axis, skipna, numeric_only, *
*kwargs)
  11945 nv.validate_func(name, (), kwargs)
  11947 validate_bool_kwarg(skipna, "skipna", none_allowed=False)
> 11949 return self._reduce(
  11950     func, name=name, axis=axis, skipna=skipna, numeric_only=numeric_
only
  11951 )

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:11204,
in DataFrame._reduce(self, op, name, axis, skipna, numeric_only, filter_typ
e, **kwds)
  11200     df = df.T
  11202 # After possibly _get_data and transposing, we are now in the
  11203 #  simple case where we can use BlockManager.reduce
> 11204 res = df._mgr.reduce(blk_func)
  11205 out = df._constructor_from_mgr(res, axes=res.axes).iloc[0]
  11206 if out_dtype is not None and out.dtype != "boolean":

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/internals/manag
ers.py:1459, in BlockManager.reduce(self, func)
   1457 res_blocks: list[Block] = []
   1458 for blk in self.blocks:
-> 1459     nbs = blk.reduce(func)
   1460     res_blocks.extend(nbs)
   1462 index = Index([None])  # placeholder
```

```
File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/internals/block
s.py:377, in Block.reduce(self, func)
    371 @final
    372 def reduce(self, func) -> list[Block]:
    373     # We will apply the function and reshape the result into a singl
e-row
    374     #  Block with the same mgr_locs; squeezing will be done at a hig
her level
    375     assert self.ndim == 2
--> 377     result = func(self.values)
    379     if self.values.ndim == 1:
    380         res_values = result

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/frame.py:11136,
in DataFrame._reduce.<locals>.blk_func(values, axis)
  11134         return np.array([result])
  11135 else:
> 11136     return op(values, axis=axis, skipna=skipna, **kwds)

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/nanops.py:147,
in bottleneck_switch.__call__.<locals>.f(values, axis, skipna, **kwds)
    145         result = alt(values, axis=axis, skipna=skipna, **kwds)
    146 else:
--> 147     result = alt(values, axis=axis, skipna=skipna, **kwds)
    149 return result

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/nanops.py:404,
in _datetimelike_compat.<locals>.new_func(values, axis, skipna, mask, **kwar
gs)
    401 if datetimelike and mask is None:
    402     mask = isna(values)
--> 404 result = func(values, axis=axis, skipna=skipna, mask=mask, **kwargs)
    406 if datetimelike:
    407     result = _wrap_results(result, orig_values.dtype, fill_value=iNa
T)

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/nanops.py:720,
in nanmean(values, axis, skipna, mask)
    718 count = _get_counts(values.shape, mask, axis, dtype=dtype_count)
    719 the_sum = values.sum(axis, dtype=dtype_sum)
--> 720 the_sum = _ensure_numeric(the_sum)
    722 if axis is not None and getattr(the_sum, "ndim", False):
    723     count = cast(np.ndarray, count)

File /opt/anaconda3/lib/python3.11/site-packages/pandas/core/nanops.py:1678,
in _ensure_numeric(x)
   1675 inferred = lib.infer_dtype(x)
   1676 if inferred in ["string", "mixed"]:
   1677     # GH#44008, GH#36703 avoid casting e.g. strings to numeric
-> 1678     raise TypeError(f"Could not convert {x} to numeric")
   1679 try:
   1680     x = x.astype(np.complex128)

TypeError: Could not convert ['fmnffnmmff'] to numeric
```

That worked, but it complained (at least my version of pandas did). It wants us to pick only valid (numeric) columns over which to compute the mean. Okay.

In [64]:
```
incomes[['wage', 'interest']].mean()
```

Out[64]:
```
wage        147.4
interest     79.1
dtype: float64
```

Compute the standard deviation of total income (in Euros, if you prefer)

In [66]:
```
# deviation of total income
incomes['total_eu'].std()
```

Out[66]:
```
44.94613492219819
```

Pro tip: if you *do* want to compute a statistic on *all* the numeric columns on large data frame, you can save typing with `DataFrame.mean(numeric_only = True)`. Try it!

In [68]:
```
incomes.mean(numeric_only = True)
```

Out[68]:
```
wage                    147.40
interest                 79.10
total                   226.50
total_eu                212.91
wage_exceeds_interest     0.90
dtype: float64
```

Computing statistics by group

We can also easily compute statistics separately based on a grouping variable, like 'gender' for the incomes data.

Here's our grouping variable:

In [69]:
```
incomes['gender']
```

Out[69]:
```
0    f
1    m
2    n
3    f
4    f
5    n
6    m
7    m
8    f
9    f
Name: gender, dtype: object
```

And now we'll use it in our data frame's `groupby()` method. Like this.

In [70]:
```
incomes[['total', 'gender']].groupby('gender').mean()
```

Out[70]:

gender	total
f	216.400000
m	235.333333
n	238.500000

If you are coming from the R/tidyverse world (e.g. if you took PSY420 recently), you'll recognize this command as similar to using the pipe (`%>%`).

What's happening is that

- `incomes[['total', 'gender']]` creates a data frame
- `groupby('gender')` creates another data frame grouped by gender
- `mean()` computes the mean on the grouped data frame

So we could (almost) turn this directly into R code that uses the pipe:

```
incomes[['total', 'gender']] %>%
groupby('gender') %>%
mean()
```

How many people were in each group? Just use the `value_counts()` method!

In [71]:
```
incomes['gender'].value_counts()
```

Out[71]:
```
gender
f    5
m    3
n    2
Name: count, dtype: int64
```

In the cell below, compute the survival rate for passengers on the RMS Titanic grouped by passenger class.

(*hint* - having the Survived variable coded as 0 or 1 works to your advantage)

In [72]:
```
titanic.groupby("Pclass")["Survived"].mean()
```

Out[72]:
```
Pclass
1    0.629630
2    0.472826
3    0.242363
Name: Survived, dtype: float64
```

Multiple statistics using aggregation

We can compute many things at once using the `agg()` (aggregate) method. To use this method, we pass it a dictionary in which the keys are column names and the values are lists of valid statistics (i.e. methods that `DataFrames` know about). Like this.

In [73]:
```
my_stats_dict = {
         "wage": ["mean", "std"],
         "interest": ["mean", "std"],
         "total": ["mean", "std"]
     }

incomes.agg(my_stats_dict)
```

Out[73]:

	wage	interest	total
mean	147.400000	79.100000	226.500000
std	27.281251	51.977452	47.815037

You can do the above in one go (rather than defining a separate `my_stats_dict` object), but it looks a bit messy in our opinion.