Pandas for Everyone

Python Data Analysis

Rough Cuts

Daniel Y. Chen

Contents

Chapter 1. Pandas Dataframe basics

1.1 Introduction

Pandas is an open source Python library for data analysis. It gives Python the ability to work with spreadsheet-like data for fast data loading, manipulating, aligning, merging, etc. To give Python these enhanced features, Pandas introduces two new data types to Python: Series and DataFrame. The DataFrame will represent your entire spreadsheet or rectangular data, whereas the Series is a single column of the DataFrame. A Pandas DataFrame can also be thought of as a dictionary or collection of Series.

Why should you use a programming language like Python and a tool like Pandas to work with data? It boils down to automation and reproducibility. If there is a articular set of analysis that needs to be performed on multiple datasets, a programming language has the ability to automate the analysis on the datasets. Although many spreadsheet programs have its own macro programming language, many users do not use them. Furthermore, not all spreadsheet programs are available on all operating systems. Performing data takes using a programming language forces the user to have a running record of all steps performed on the data. I, like many people, have accidentally hit a key while viewing data in a spreadsheet program, only to find out that my results do not make any sense anymore due to bad data. This is not to say spreadsheet programs are bad or do not have their place in the data workflow, they do, but there are better and more reliable tools out there.

1.2 Concept map

1. Prior knowledge needed (appendix)

(a) relative directories

(b) calling functions

(c) dot notation

(d) primitive python containers

(e) variable assignment

(f) the print statement in various Python environments

2. This chapter

(a) loading data

(b) subset data

(c) slicing

(d) filtering

(e) basic pd data structures (series, dataframe)

(f) resemble other python containers (list, np.ndarray)

(g) basic indexing

1.3 Objectives

This chapter will cover:

1. loading a simple delimited data file

2. count how many rows and columns were loaded

3. what is the type of data that was loaded

4. look at different parts of the data by subsetting rows and columns

5. saving a subset of data

1.4 Loading your first data set

When given a data set, we first load it and begin looking at its structure and contents. The simplest way of looking at a data set is to look and subset specific rows and columns. We can see what type of information is stored in each column, and can start looking for patterns by aggregating descriptive statistics.

Since Pandas is not part of the Python standard library, we have to first tell Python to load `(import)` the library.

```
import pandas
```

With the library loaded we can use the `read_csv` function to load a CSV data file. In order to access the `read_csv` function from pandas, we use something called 'dot notation'. More on dot notations can be found in (TODO Functions appendix and modules).

About the Gapminder dataset

The Gapminder dataset originally comes from:. This particular version the book is using Gapminder data prepared by Jennifer Bryan from the University of British Columbia. The repository can be found at: www.github.com/jennybc/gapminder.

```
# by default the read_csv function will read a comma separated
# our gapminder data set is separated by a tab
# we can use the sep parameter and indicate a tab with \t
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
# we use the head function so Python only shows us the first 5
print(df.head())
```

```
       country continent  year  lifeExp       pop   gdpPercap
0  Afghanistan      Asia  1952   28.801   8425333  779.445314
1  Afghanistan      Asia  1957   30.332   9240934  820.853030
2  Afghanistan      Asia  1962   31.997  10267083  853.100710
3  Afghanistan      Asia  1967   34.020  11537966  836.197138
4  Afghanistan      Asia  1972   36.088  13079460  739.981106
```

Since we will be using Pandas functions many times throughout the book as well as your own programming. It is common to give `pandas` the alias `pd`. The above code will be the same as below:

```
import pandas as pd
df = pd.read_csv('../data/gapminder.tsv', sep='\t')
print(df.head())
```

We can check to see if we are working with a Pandas Dataframe by using the built-in `type` function (i.e., it comes directly from Python, not any package such as Pandas).

```
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

The `type` function is handy when you begin working with many different types of Python objects and need to know what object you are currently working on.

The data set we loaded is currently saved as a Pandas DataFrame object and is relatively small. Every DataFrame object has a `shape` attribute that will give us the number of rows and columns of the DataFrame.

```
print(df.shape)
```

```
(1704, 6)
```

The shape attribute returns a tuple (TODO appendix) where the first value is the number of rows and the second number is the number of columns. From the results above, we see our gapminder data set has 1704 rows and 6 columns.

Since `shape` is an attribute of the dataframe, and not a function or method of the DataFrame, it does not have parenthesis after the period. If you made the mistake of putting parenthesis after the `shape` attribute, it would return an error.

```
print(df.shape())
```

```
<class 'TypeError'>
'tuple' object is not callable
```

Typically, when first looking at a dataset, we want to know how many rows and columns there are (we just did that), and to get a gist of what information it contains, we look at the columns. The column names, like `shape`, is given using the `column` attribute of the dataframe object.

```
# get column names

print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpI
```

Question

What is the `type` of the column names?

The Pandas DataFrame object is similar to other languages that have a DataFrame-like object (e.g., Julia and R) Each column (Series) has to be the same type, whereas, each row can contain mixed types. In our current example, we can expect the `country` column to be all strings and the year to be integers. However, it's best to make sure that is the case by using the `dtypes` attribute or the `info` method. Table 1–1 on page 7 shows what the type in Pandas is relative to native Python.

```
print(df.dtypes)
```

```
country        object
continent      object
year            int64
lifeExp       float64
pop             int64
gdpPercap     float64
dtype: object
```

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
```

```
lifeExp      1704 non-null float64
pop          1704 non-null int64
gdpPercap    1704 non-null float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

Pandas Type	Python Type	Description
object	string	most common data type
int64	int	whole numbers
float64	float	numbers with decimals
datetime64	datetime	datetime is found in the Python standard library (i.e., it is not loaded by default and needs to be imported)

Table 1-1: Table of Pandas dtypes and Python types

1.5 Looking at columns, rows, and cells

Now that we're able to load up a simple data file, we want to be able to inspect its contents. We could `print` out the contents of the dataframe, but with todays data, there are too many cells to make sense of all the printed information. Instead, the best way to look at our data is to inspect it in parts by looking at various subsets of the data. We already saw above that we can use the `head` method of a dataframe to look at the first 5 rows of our data. This is useful to see if our data loaded properly, get a sense of the columns, its name and its contents. However, there are going to be times when we only want particular rows, columns, or values from our data.

Before continuing, make sure you are familiar with Python containers. (TODO Add reference to containers in Appendix)

1.5.1 Subsetting columns

If we wanted multiple columns we can specify them a few ways: by names, positions, or ranges.

1.5.1.1 Subsetting columns by name

If we wanted only a specific column from out data we can access the data using square brackets.

```
#  just get the country column and save it to its own variable
country_df = df['country']

#  show the first 5 observations
print(country_df.head())

0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object

# show the last 5 observations
print(country_df.tail())

1699   Zimbabwe
1700   Zimbabwe
1701   Zimbabwe
1702   Zimbabwe
1703   Zimbabwe
Name: country, dtype: object
```

When subsetting a single column, you can use dot notation and call the column name attribute directly.

```
country_df_dot = df.country
print(country_df_dot.head())
```

```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

In order to specify multiple columns by the column name, we need to pass in a python `list` between the square brackets. This may look a but strange since there will be 2 sets of square brackets.

```
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
print(subset.head())

        country continent  year
 0  Afghanistan      Asia  1952
 1  Afghanistan      Asia  1957
 2  Afghanistan      Asia  1962
 3  Afghanistan      Asia  1967
 4  Afghanistan      Asia  1972

print(subset.tail())

         country  continent  year
 1699   Zimbabwe     Africa  1987
 1700   Zimbabwe     Africa  1992
 1701   Zimbabwe     Africa  1997
 1702   Zimbabwe     Africa  2002
 1703   Zimbabwe     Africa  2007
```

Again, you can opt to `print` the entire `subset` dataframe. I am not doing this for the book as it would take up an unnecessary amount of space.

1.5.1.2 Subsetting columns by index position

At times, you may only want to get a particular column by its position, rather than its name. For example, you want to get the first (country) column and third column (year), or just the last column (gdpPercap).

```
#  try to get the first column by passing the integer 1
subset = df[[1]]
#  we really end up getting the second column
```

```
print(subset.head())

   continent
 0      Asia
 1      Asia
 2      Asia
 3      Asia
 4      Asia
```

You can see when we put `1` into the list, we actually get the second column, and not the first. This follows Python's zero indexed behavior, meaning, the first item of a container is index 0 (i.e., 0th item of the container). More details about this kind of behavior can be found in (TODO Appendix containers)

```
#  get the first column (index 0) and last column
subset = df[[0, -1]]
print(subset.head())

        country   gdpPercap
 0  Afghanistan  779.445314
 1  Afghanistan  820.853030
 2  Afghanistan  853.100710
 3  Afghanistan  836.197138
 4  Afghanistan  739.981106
```

There's other ways of subsetting columns, but that builds on the methods used to subset rows.

1.5.1.3 Subsetting columns by range

You can use the built-in `range` function to create a range of values in Python. This way you can specify a beginning and end value, and python will automatically create a range of values in between. By default, every value between the beginning and end (inclusive left, exclusive right – TODO SEE APPENDIX) will be created, unless you specify a step (More on ranges TODO – SEE APPENDIX). In Python 3 the `range` function returns a generator (TODO SEE APENDIX). If you are using Python 2, the `range` function returns a list (TODO SEE APENDIX), and the `xrange` function returns a generator.

If we look at the code above (section ??), we see that we subset columns using a list of integers. since `range` returns a generator, we have to convert the generator to a list first.

```
#  create a range of integers from 0 - 4 inclusive
small_range = list(range(5))
#  subset the dataframe with the range
subset = df[small_range]
print(subset.head())
```

```
       country continent  year  lifeExp       pop
 0  Afghanistan      Asia  1952   28.801   8425333
 1  Afghanistan      Asia  1957   30.332   9240934
 2  Afghanistan      Asia  1962   31.997  10267083
 3  Afghanistan      Asia  1967   34.020  11537966
 4  Afghanistan      Asia  1972   36.088  13079460
```

Note that when `range(5)` is called, 5 integers are returned from 0 - 4.

Table 1-2: Different methods of indexing rows (and or columns)

Subset method	Description
`loe`	subset based on index label (a.k.a. row name)
`iloc`	subset based on row index (a.k.a. row number)
`ix`	subset based on index label or row index, depends on what's given

```
# create a range from 3 - 5 inclusive
small_range = list(range(3, 6))
subset = df[small_range]
print(subset.head())
```

```
   lifeExp       pop   gdpPercap
 0   28.801   8425333  779.445314
 1   30.332   9240934  820.853030
 2   31.997  10267083  853.100710
```

```
3   34.020  11537966  836.197138
4   36.088  13079460  739.981106
```

Question

What happens when you specify a range that's beyond the number of columns you have?

Again, note that the values are specified in a way such that it is inclusive on the left, and exclusive on the right.

```
# create a range form 0 - 5 inclusive, every other integer
small_range = list(range(0, 6, 2))
subset = df[small_range]
print(subset.head())
```

```
       country  year       pop
0  Afghanistan  1952   8425333
1  Afghanistan  1957   9240934
2  Afghanistan  1962  10267083
3  Afghanistan  1967  11537966
4  Afghanistan  1972  13079460
```

Converting a generator to a list is a bit awkward, but sometimes it's the only way. In the next few sections, we'll show how to subset dataframe with different syntax and methods. And give us a less awkward way to subset rows and columns.

1.5.2 Subsetting rows

Just like columns, rows can be subset in multiple ways: row name, row index, or a combination of both. Table 1–2 gives a quick overview of the various methods.

1.5.2.1 Subset rows by index label - .loc If we take a look at our gapminder data

```
print(df.head())
```

```
       country continent  year  lifeExp       pop   gdpPercap
```

```
 0  Afghanistan      Asia  1952   28.801   8425333  779.445314
 1  Afghanistan      Asia  1957   30.332   9240934  820.853030
 2  Afghanistan      Asia  1962   31.997  10267083  853.100710
 3  Afghanistan      Asia  1967   34.020  11537966  836.197138
 4  Afghanistan      Asia  1972   36.088  13079460  739.981106
```

We can see on the left side of the printed dataframe, what appears to be row numbers. This column-less row of values is the index label of the dataframe. Think of it like column names, but instead for rows. By default, Pandas will fill in the index labels with the row numbers. A common example where the row index labels are not the row number is when we work with time series data. In that case, the index label will be a timestamps of sorts, but for now we will keep the default row number values.

We can use the . loc method on the dataframe to subset rows based on the index label.

```
# get the first row
print(df.loc[0])

 country      Afghanistan
 continent           Asia
 year                1952
 lifeExp           28.801
 pop              8425333
 gdpPercap        779.445
 Name: 0, dtype: object

# get the 100th row
# recall that values start with 0
print(df.loc[99])

 country      Bangladesh
 continent          Asia
 year               1967
 lifeExp          43.453
 pop            62821884
 gdpPercap       721.186
 Name: 99, dtype: object

# get the last row
print(df.loc[-1])

 <class 'KeyError'>
```

```
'the label [-1] is not in the [index]'
```

Note that passing `-1` as the `loc` will cause an error, because it is actually looking for the row index label (row number) `-1`, which does not exist in our example. Instead we can use a bit of Python to calculate the number of rows and pass that value into `loc`.

```
# get the last row (correctly)
# use the first value given from shape to get the total number
number_of_rows = df.shape[0]
# subtract 1 from the value since we want the last index value
last_row_index = number_of_rows - 1
# finally do the subset using the index of the last row
print(df.loc[last_row_index])
```

```
country       Zimbabwe
continent       Africa
year              2007
lifeExp         43.487
pop           12311143
gdpPercap      469.709
Name: 1703, dtype: object
```

Or simply use the `tail` method to return the last 1 row, instead of the default 5.

```
#  there are many ways of doing what you want
print(df.tail(n=1))
```

```
       country continent  year  lifeExp       pop   gdpPercap
1703  Zimbabwe    Africa  2007   43.487  12311143  469.709298
```

Notice that using `tail ()` and `loc` printed out the results differently. Let's look at what type is returned when we use these methods.

```
subset_loc = df.loc[0]
subset_head = df.head(n=1)
print(type(subset_loc))
```

```
<class 'pandas.core.series.Series'>
```

```
print(type(subset_head))
```

```
<class 'pandas.core.frame.DataFrame'>
```

The beginning of the chapter mentioned how Pandas introduces two new data types into Python. Depending on what method we use and how many rows we return, pandas will return a different.

Subsetting multiple rows Just like with columns we can select multiple rows.

```
# select the first, 100th, and 1000th row
# note the double square brackets similar to the syntax used to
# subset multiple columns
print(df.loc[[0, 99, 999]])

         country continent  year  lifeExp       pop    gdpPerc
 0   Afghanistan      Asia  1952   28.801   8425333   779.4453
 99   Bangladesh      Asia  1967   43.453  62821884   721.1860
 999    Mongolia      Asia  1967   51.253   1149500  1226.0411
```

1.5.2.2 Subset rows by row number - `.iloc`

`iloc` does the same thing as `loc` but it is used to subset by the row index number. In our current example `iloc` and `loc` will behave exactly the same since the index labels are the row numbers. However, keep in mind that the index labels do not necessarily have to be row numbers.

```
# get the first row
print(df.iloc[0])
 country      Afghanistan
 continent           Asia
 year                1952
 lifeExp           28.801
 pop              8425333
 gdpPercap        779.445
 Name: 0, dtype: object

## get the 100th row
print(df.iloc[99])
 country      Bangladesh
 continent          Asia
 year               1967
 lifeExp          43.453
 pop            62821884
 gdpPercap       721.186
 Name: 99, dtype: object
```

```
## get the first, 100th, and 1000th row
print(df.iloc[[0, 99, 999]])

         country continent  year  lifeExp       pop    gdpPer
 0   Afghanistan      Asia  1952   28.801   8425333   779.445
 99   Bangladesh      Asia  1967   43.453  62821884   721.186
 999    Mongolia      Asia  1967   51.253   1149500  1226.041
```

1.5.2.3 Subsetting rows with `.ix` (combination of `.loc` and `.iloc`)

#TODO show this example but refer to a future example that have different row index labels

`.ix` allows us to subset by integers and labels. By default it will search for labels, and if it cannot find the corresponding label, it will fall back to using integer indexing. This is the most general form of subsetting. The benefits may not be obvious with our current dataset. But as our data begins to have hierarchies and our subsetting methods become more complex, the flexibility of ix will be obvious.

```
# get the first row
print(df.ix[0])
  country      Afghanistan
  continent           Asia
  year                1952
  lifeExp           28.801
  pop              8425333
  gdpPercap        779.445
  Name: 0, dtype: object

# get the 100th row
print(df.ix[99])
  country      Bangladesh
  continent          Asia
  year               1967
  lifeExp          43.453
  pop            62821884
  gdpPercap       721.186
  Name: 99, dtype: object

# get the first, 100th, and 1000th row
print(df.ix[[0, 99, 999]])
```

```
        country continent  year  lifeExp       pop    gdpPe
 0    Afghanistan      Asia  1952   28.801   8425333   779.44
 99    Bangladesh      Asia  1967   43.453  62821884   721.18
 999     Mongolia      Asia  1967   51.253   1149500  1226.04
```

1.5.3 Mixing it up

1.5.3.1 Subsetting rows and columns

The `loc`, `iloc` , and ix methods all have the ability to subset rows and columns simultaneously. In the previous set of examples, when we wanted to select multiple columns or multiple rows, there was an additional set of square brackets. However if we omit the square brackets, we can actually subset rows and columns simultaneously. Essentially, the syntax goes as follows: separate the row subset values and the column subset values with a comma. The part to the left of the comma will be the row values to subset, the part to the right of the comma will be the column values to subset.

```
# get the 43rd country in our data
print(df.ix[42, 'country'])
 Angola
```

Note the syntax for ix will work for loc and iloc as well

```
print(df.loc[42, 'country'])

 Angola

print(df.iloc[42, 0])

 Angola
```

Just make sure you don’t confuse the differences between `loc` and `iloc`

```
print(df.loc[42, 0])
 <class 'TypeError'>
 cannot do label indexing on <class 'pandas.indexes.base.Index'
 these indexers [0] of <class 'int'>
```

and remember the flexibility of `ix`.

```
# compare this ix code with the one above.
# instead of 'country' I used the index 0
print(df.ix[42, 0])
```

```
 Angola
```

1.5.3.2 Subsetting multiple rows and columns

We can combine the row and column subsetting syntax with the multiple row and column subsetting syntax to get various slices of our data.

```
# get the first, 100th, and 1000th rows from the first, 4th, ar
column
#  note the columns we are hoping to get are: country, lifeExp,
gdpPercap
print(df.ix[[0, 99, 999], [0, 3, 5]])
```

```
          country  lifeExp    gdpPercap
 0    Afghanistan   28.801   779.445314
 99    Bangladesh   43.453   721.186086
 999     Mongolia   51.253  1226.041130
```

I personally try to pass in the actual column names when subsetting data if possible. It makes the code more readable since you do not need to look at the column name vector to know which index is being called. Additionally, using absolute indexes can lead to problems if the column order gets changed for whatever reason.

```
#  if we use the column names directly, it makes the code a bi
to read
print(df.ix[[0, 99, 999], ['country', 'lifeExp', 'gdpPercap']])
```

```
          country  lifeExp    gdpPercap
 0    Afghanistan   28.801   779.445314
 99    Bangladesh   43.453   721.186086
 999     Mongolia   51.253  1226.041130
```

1.6 Grouped and aggregated calculations

If you've worked with other numeric libraries or languages, many basic statistic calculations either come with the library, or are built into the language.

Looking at our gapminder data again

```
print(df.head(n=10))
```

```
       country continent  year  lifeExp       pop   gdpPercap
  0  Afghanistan      Asia  1952   28.801   8425333  779.44531
  1  Afghanistan      Asia  1957   30.332   9240934  820.85303
  2  Afghanistan      Asia  1962   31.997  10267083  853.10071
  3  Afghanistan      Asia  1967   34.020  11537966  836.19713
  4  Afghanistan      Asia  1972   36.088  13079460  739.98110
  5  Afghanistan      Asia  1977   38.438  14880372  786.11336
  6  Afghanistan      Asia  1982   39.854  12881816  978.01143
  7  Afghanistan      Asia  1987   40.822  13867957  852.39594
  8  Afghanistan      Asia  1992   41.674  16317921  649.34139
  9  Afghanistan      Asia  1997   41.763  22227415  635.34135
```

There are several initial questions that we can ask ourselves:

1. For each year in our data, what was the average life expectancy? what about population and GDP?

2. What if we stratify by continent?

3. How many countries are listed in each continent?

1.6.1 Grouped means

In order to answer the questions posed above, we need to perform a grouped (aka aggregate) calculation. That is, we need to perform a calculation, be it an average, or frequency count, but apply it to each subset of a variable. Another way to think about grouped calculations is split-apply-combine. We first split our data into various parts, apply a function (or calculation) of our choosing to each of the split parts, and finally combine all the individual split calculation into a single dataframe. We accomplish grouped/aggregate computations by using the `groupby` method on dataframes.

```
#  For each year in our data, what was the average life expecta
#  To answer this question, we need to split our data into part
year
#  then we get the 'lifeExp' column and calculate the mean
print(df.groupby('year')['lifeExp'].mean())
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Let's unpack the statement above. We first create a grouped object. Notice that if we printed the grouped dataframe, pandas only returns us the memory location

```
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))
print(grouped_year_df)
```

```
<class 'pandas.core.groupby.DataFrameGroupBy'>
<pandas.core.groupby.DataFrameGroupBy object at 0x7f33ff57a24(
```

From the grouped data, we can subset the columns of interest we want to perform calculations on. In our case our question needs the `lifeExp` column. We can use the subsetting methods described in section 1.5.1.1.

```
grouped_year_df_lifeExp = grouped_year_df['lifeExp']
print(type(grouped_year_df_lifeExp))
print(grouped_year_df_lifeExp)
 <class 'pandas.core.groupby.SeriesGroupBy'>
 <pandas.core.groupby.SeriesGroupBy object at 0x7f33ff584f60>
```

Notice we now are given a series (because we only asked for 1 column) where the contents of the series are grouped (in our example by year).

Finally, we know the `lifeExp` column is of type `float64`. An operation we can perform on a vector of numbers is to calculate the mean to get our final desired result.

```
mean_lifeExp_by_year = grouped_year_df_lifeExp.mean()
print(mean_lifeExp_by_year)
 year
 1952     49.057620
 1957     51.507401
 1962     53.609249
 1967     55.678290
 1972     57.647386
 1977     59.570157
 1982     61.533197
 1987     63.212613
 1992     64.160338
 1997     65.014676
 2002     65.694923
 2007     67.007423
  Name: lifeExp, dtype: float64
```

We can perform a similar set of calculations for population and GDP since they are of types `int64` and `float64`, respectively. However, what if we want to group and stratify by more than one variable? and perform the same calculation on multiple columns? We can build on the material earlier in this chapter by using a list!

```
print(df.groupby(['year', 'continent'])[['lifeExp',
' gdpPercap']].mean())
                  lifeExp        gdpPercap
 year continent
 1952 Africa     39.135500   1252.572466
      Americas   53.279840   4079.062552
      Asia       46.314394   5195.484004
      Europe     64.408500   5661.057435
      Oceania    69.255000  10298.085650
 1957 Africa     41.266346   1385.236062
      Americas   55.960280   4616.043733
      Asia       49.318544   5787.732940
      Europe     66.703067   6963.012816
      Oceania    70.295000  11598.522455
 1962 Africa     43.319442   1598.078825
      Americas   58.398760   4901.541870
      Asia       51.563223   5729.369625
      Europe     68.539233   8365.486814
      Oceania    71.085000  12696.452430
 1967 Africa     45.334538   2050.363801
      Americas   60.410920   5668.253496
      Asia       54.663640   5971.173374
      Europe     69.737600  10143.823757
```

```
     Oceania    71.310000  14495.021790
1972 Africa     47.450942   2339.615674
     Americas   62.394920   6491.334139
     Asia       57.319269   8187.468699
     Europe     70.775033  12479.575246
     Oceania    71.910000  16417.333380
1977 Africa     49.580423   2585.938508
     Americas   64.391560   7352.007126
     Asia       59.610556   7791.314020
     Europe     71.937767  14283.979110
     Oceania    72.855000  17283.957605
1982 Africa     51.592865   2481.592960
     Americas   66.228840   7506.737088
     Asia       62.617939   7434.135157
     Europe     72.806400  15617.896551
     Oceania    74.290000  18554.709840
1987 Africa     53.344788   2282.668991
     Americas   68.090720   7793.400261
     Asia       64.851182   7608.226508
     Europe     73.642167  17214.310727
     Oceania    75.320000  20448.040160
1992 Africa     53.629577   2281.810333
     Americas   69.568360   8044.934406
     Asia       66.537212   8639.690248
     Europe     74.440100  17061.568084
     Oceania    76.945000  20894.045885
1997 Africa     53.598269   2378.759555
     Americas   71.150480   8889.300863
     Asia       68.020515   9834.093295
     Europe     75.505167  19076.781802
     Oceania    78.190000  24024.175170
2002 Africa     53.325231   2599.385159
     Americas   72.422040   9287.677107
     Asia       69.233879  10174.090397
     Europe     76.700600  21711.732422
     Oceania    79.740000  26938.778040
2007 Africa     54.806038   3089.032605
     Americas   73.608120  11003.031625
     Asia       70.728485  12473.026870
     Europe     77.648600  25054.481636
     Oceania    80.719500  29810.188275
```

The output data is grouped by year and continent. For each year-continent set, we calculated the average life expectancy and GDP. The data is also printed out a little differently. Notice the year and continent 'column names' are not on the same line as the life expectancy and GPD 'column names'. There is some hierarchal structure between the year and continent row indices. More about working with these types of data in (TODO REFERENCE CHAPTER HERE).

Question: does the order of the list we use to group matter?

1.6.2 Grouped frequency counts

Another common data task is to calculate frequencies. We can use the ‘nunique‘ or ‘value counts’ methods to get a count of unique values, or frequency counts, respectively on a Pandas Series.

```
# use the nunique (number unique) to calculate the number of ui
values in a series
print(df.groupby('continent')['country'].nunique())
 continent
 Africa      52
 Americas    25
 Asia        33
 Europe      30
 Oceania      2
 Name: country, dtype: int64
```

Question

What do you get if you use ‘value counts’ instead of ‘nunique’?

1.7 Basic plot

Visualizations are extremely important in almost every step of the data process. They help identify trends in data when we are trying to understand and clean it, and they help convey our final findings.

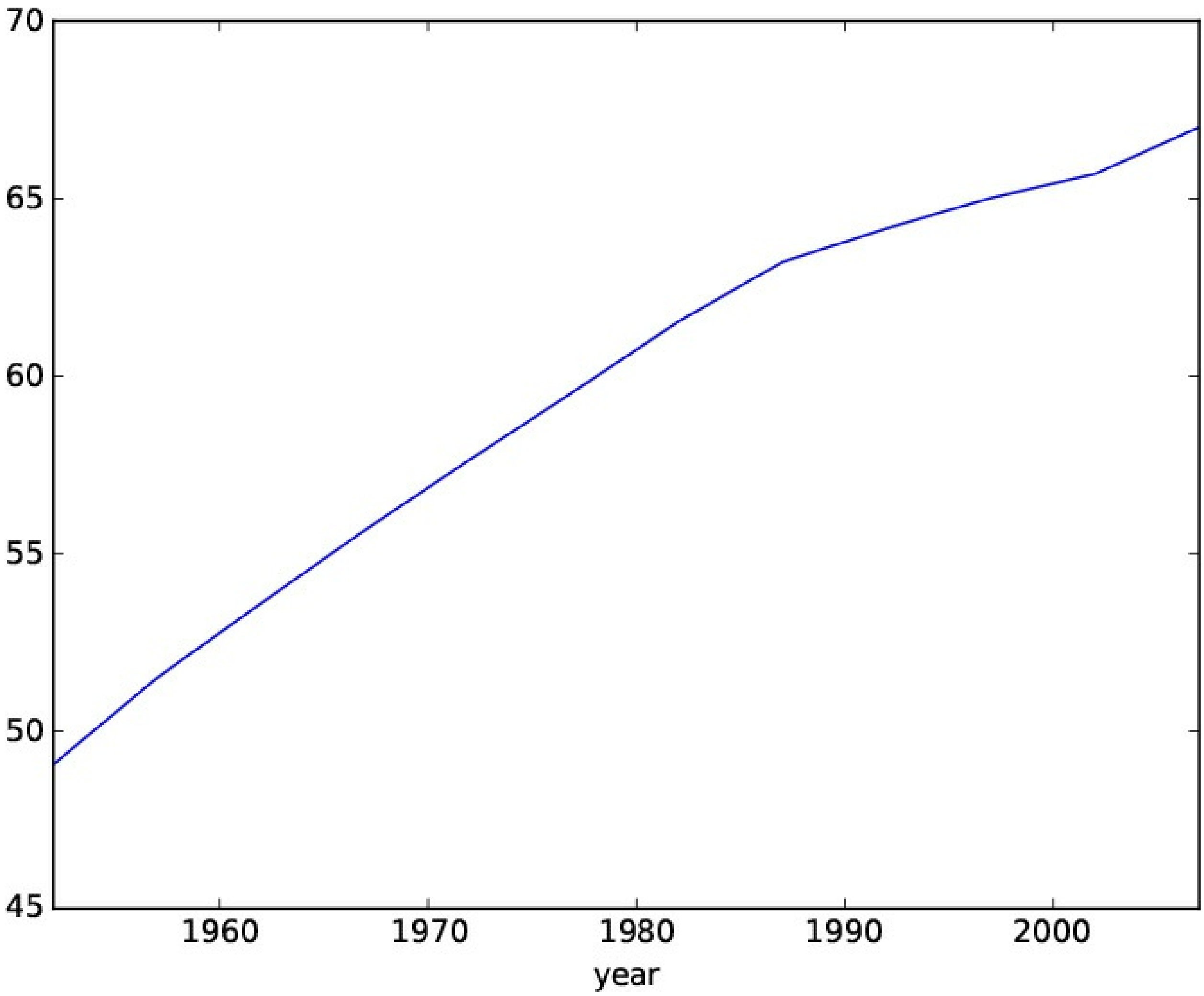
Let’s look at the yearly life expectancies of the world again.

```
global_yearly_life_expectancy = df.groupby('year')['lifeExp'].r
print(global_yearly_life_expectancy)

 year
 1952    49.057620
```

```
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

We can use pandas to do some basic plots.

```
global_yearly_life_expectancy.plot()
```



1.8 Conclusion

In this chapter I showed you how to load up a simple dataset and start looking at specific observations. It may seem tedious at first to look at observations this way especially if you have been coming from a spreadsheet program. Keep in mind, when doing data analytics, the goal is to be reproducible, and not repeat repetitive tasks. Scripting languages give you that ability and flexibility.

Along the way you learned some of the fundamental programming abilities and data structures Python has to offer. As well as a quick way to go aggregated statistics and plots. In the next chapter I will be going into more detail about the Pandas DataFrame and Series object, as well as more ways you can subset and visualize your data.

As you work your way though the book, if there is a concept or data structure that is foreign to you, check the Appendix. I've put many of the fundamental programming features of Python there.

Chapter 2. Pandas data structures

2.1 Introduction

Chapter 1, mentions the Pandas `DataFrame` and codeSeries data structures. These data structures will resemble the primitive Python data containers (lists and dictionaries) for indexing and labeling, but have additional features to make working with data easier.

2.2 Concept map

1. Prior knowledge

(a) Containers

(b) Using functions

(c) Subsetting and indexing

2. load in manual data

3. Series

(a) creating a series

i. dict

ii. ndarray

iii. scalar iv. lists

(b) slicing

2.3 Objectives

This chapter will cover:

1. load in manual data

2. learn about the Series object

3. basic operations on Series objects

4. learn about the DataFrame object

5. conditional subsetting and fancy slicing and indexing

6. save out data

2.4 Creating your own data

Whether you are manually inputting data, or creating a small test example, knowing how to create dataframes without loading data from a file is a useful skill.

2.4.1 Creating a `Series`

The Pandas Series is a one-dimensional container, similar to the built in python `list`. It is the datatype that represents each column of the `DataFrame`. Table 1–1 lists the possible `dtypes` for Pandas `DataFrame` columns. Each column in a dataframe must be of the same `dtype`. Since a dataframe can be thought of a dictionary of `Series` objects, where each `key` is the column name, and the `value` is the `Series`, we can conclude that a series is very similar to a python `list` , except each element must be the same `dtype`. Those who have used the `numpy` library will realize this is the same behavior as the `ndarray`.

The easiest way to create a `series` is to pass in a Python `list` . If we pass in a list of mixed types, the most common representation of both will be used. Typically the dtype will be `object`.

```
import pandas as pd
s = pd.Series(['banana', 42])
print(s)
```

```
0    banana
1        42
dtype: object
```

You'll notice on the left the 'row number' is shown. This is actually the `index` for the series. It is similar to the row name and row index we saw in section 1.5.2 for dataframes. This implies that we can actually assign a 'name' to values in our series.

```
# manually assign index values to a series
# by passing a Python list
s = pd.Series(['Wes McKinney', 'Creator of Pandas'],
               index=['Person', 'Who'])
print(s)
```

```
Person          Wes McKinney
Who        Creator of Pandas
dtype: object
```

Questions

1. What happens if you use other Python containers like `list` , `tuple`, `dict`, or even the `ndarray` from the `numpy` library?

2. What happens if you pass an `index` along with the containers?

3. Does passing in an `index` when you use a `dict` overwrite the index? Or does it sort the values?

2.4.2 Creating a `DataFrame`

As mentioned in section 1.1, a `DataFrame` can be thought of as a dictionary of `Series` objects. This is why dictionaries are the the most common way of creating a `DataFrame`. The `key` will represent the column name, and the `values` will be the contents of the column.

```
scientists = pd.DataFrame({
    ' Name': ['Rosaline Franklin', 'William Gosset'],
```

```
    ' Occupation': ['Chemist', 'Statistician'],
    ' Born': ['1920-07-25', '1876-06-13'],
    ' Died': ['1958-04-16', '1937-10-16'],
    ' Age': [37, 61]})
print(scientists)
```

```
    Age        Born        Died                Name    Occupati
 0   37  1920-07-25  1958-04-16  Rosaline Franklin       Chemi
 1   61  1876-06-13  1937-10-16     William Gosset  Statistici
```

Notice that order is not guaranteed.

If we look at the documentation for `DataFrame`[1], we can use the `columns` parameter or specify the column order. If we wanted to use the `name` column for the row `index`, we can use the index parameter.

```
scientists = pd.DataFrame(
    data={'Occupation': ['Chemist', 'Statistician'],
          'Born': ['1920-07-25', '1876-06-13'],
          'Died': ['1958-04-16', '1937-10-16'],
          'Age': [37, 61]},
    index=['Rosaline Franklin', 'William Gosset'],
    columns=['Occupation', 'Born', 'Died', 'Age'])
print(scientists)
```

```
                     Occupation        Born        Died  Age
 Rosaline Franklin      Chemist  1920-07-25  1958-04-16   37
 William Gosset    Statistician  1876-06-13  1937-10-16   61
```

[1] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

2.5 The `Series`

In section 1.5.2.1, we saw how the slicing method effects the `type` of the result. If we use the `loc` method to subset the first row of our `scientists` dataframe, we will get a `series` object back.

```
first_row = scientists.loc['William Gosset']
print(type(first_row))
print(first_row)
 <class 'pandas.core.series.Series'>
```

```
Occupation    Statistician
Born            1876-06-13
Died            1937-10-16
Age                     61
Name: William Gosset, dtype: object
```

When a series is printed (i.e., the string representation), the index is printed down as the first 'column', and the values are printed as the second 'column'. There are many attributes and methods associated with a series object[2]. Two examples of attributes are `index` and `values`.

```
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

```
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' 61]
```

An example of a `series` method is `keys`, which is an alias for the `index` attribute.

```
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

By now, you may have questions about the syntax between `index`, `values`, and `keys`. More about attributes and methods are described in TODO APPENDIX ON CLASSES. Attributes can be thought of as properties of an object (in this example our object is a `series`). Methods can be thought of as some calculation or operation that is performed. The subsetting syntax for `loc`, `iloc` , and `ix` (from section 1.5.2) are all attributes. This is why the syntax does not have a set of round parenthesis, `()`, but rather, a set of square brackets, `[]`, for subsetting. Since `keys` is a method, if we wanted to get the first key (which is also the first index) we would use the square brackets *after* the method call.

[2] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

`Series` attributes Description

loc	Subset using index value
iloc	Subset using index position
ix	Subset using index value and/or position
dtype or dtypes	The type of the `Series` contents
T	Transpose of the series
shape	Dimensions of the data
size	Number of elements in the `Series`
values	`ndarray` or `ndarray`-like of the `Series`

```
# get the first index using an attribute
print(first_row.index[0])
```

```
 Occupation
```

```
# get the first index using a method
print(first_row.keys()[0])
```

```
 Occupation
```

2.5.1 The `Series` is ndarray-like

The `Pandas.Series` is very similar to the `numpy.ndarray` (TODO SEE APPENDIX). This means, that many methods and functions that operate on a `ndarray` will also operate on a `series`. People will also refer to a `series` as a ‘vector’.

2.5.1.1 series methods

Let’s first get a series of ’Age’ column from our `scientists` dataframe.

```
# get the ’Age’ column
ages = scientists[’Age’]
print(ages)
```

```
 Rosaline Franklin    37
 William Gosset       61
 Name: Age, dtype: int64
```

`Numpy` is a scientific computing library that typically deals with numeric vectors. Since a `series` can be thought of as an extension to the `numpy.ndarray`, there is an overlap of attributes and methods. When we have a vector of numbers, there are common calculations we can perform[3].

[3] http://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics

```
print(ages.mean())
 49.0
print(ages.min())
 37
print(ages.max())
 61
print(ages.std())
 16.970562748477714
```

The `mean`, `min`, `max`, and `std` are also methods in the `numpy.ndarray`

`Series` methods	Description
append	Concatenates 2 or more `Series`

corr	Calculate a correlation with another `Series`*
cov	Calculate a covariance with another `Series`*
describe	Calculate summary statistics*
drop duplicates	Returns a `Series` without duplicates
equals	Sees if a `Series` has the same elements
get values	Get values of the `Series`, same as the `values` attribute
hist	Draw a histogram
min	Return the minimum value
max	Returns the maximum value
mean	Returns the arithmetic mean
median	Returns the median
mode	Returns the mode(s)
quantile	Returns the value at a given quantile

replace	Replaces values in the `Series` with a specified value
sample	Returns a random sample of values from the `Series`
sort values	Sort values
to frame	Converts `Series` to `DataFrame`
transpose	Return the transpose
unique	Returns a `numpy.ndarray` of unique values

indicates missing values will be automatically dropped

2.5.2 Boolean subsetting `Series`

Chapter 1 showed how we can use specific indicies to subset our data. However, it is rare that we know the exact row or column index to subset the data. Typically you are looking for values that meet (or don't meet) a particular calculation or observation.

First, let's use a larger dataset

```
scientists pd.read_csv('../data/scientists.csv')
```

We just saw how we can calculate basic descriptive metrics of vectors

[4] http://does.scipy.org/doc/numpy/reference/arrays.ndarray.html

```
ages = scientists['Age']
print(ages)
 0     37
 1     61
```

```
 2    90
 3    66
 4    56
 5    45
 6    41
 7    77
 Name: Age, dtype: int64
```

```
print(ages.mean())
```

```
 59.125
```

```
print(ages.describe())
```

```
 count     8.000000
 mean     59.125000
 std      18.325918
 min      37.000000
 25%      44.000000
 50%      58.500000
 75%      68.750000
 max      90.000000
 Name: Age, dtype: float64
```

What if we wanted to subset our ages by those above the mean?

```
print(ages[ages > ages.mean()])
 1    61
 2    90
 3    66
 7    77
 Name: Age, dtype: int64
```

If we tease out this statement and look at what `ages > ages.mean()` returns

```
print(ages > ages.mean())
print(type(ages > ages.mean()))
 0      False
 1       True
 2       True
 3       True
 4      False
 5      False
 6      False
 7       True
 Name: Age,      dtype:      bool
```

```
<class 'pandas.core.series.Series'>
```

The statement returns a `Series` with a `dtype` of `bool`.

This means we can not only subset values using labels and indicies, we can also supply a vector of boolean values. Python has many functions and methods. Depending on how it is implemented, it may return labels, indicies, or booleans. Keep this in mind as you learn new methods and have to piece together various parts for your work.

If we wanted to, we could manually supply a vector of `bools` to subset our data.

```
# get index 0, 1, 4, and 5
manual_bool_values = [True, True, False, False, True, True, Fa
print(ages[manual_bool_values])
 0    37
 1    61
 4    56
 5    45
 Name: Age, dtype: int64
```

2.5.3 Operations are vectorized

If you're familiar with programming, you would find it strange `ages > ages.mean()` returns a vector without any `for` loops (TODO SEE APPENDIX). Many of the methods that work on series (and also dataframes) are vectorized, meaning, they work on the entire vector simultaneously. It makes the code easier to read, and typically there are optimizations to make calculations faster.

2.5.3.1 Vectors of same length

If you preform an operation between 2 vectors of the same length, the resulting vector will be an element-by-element calculation of the vectors.

```
print(ages + ages)
 0     74
 1    122
 2    180
```

```
3     132
4     112
5      90
6      82
7     154
Name: Age, dtype: int64
```

```
print(ages * ages)
0    1369
1    3721
2    8100
3    4356
4    3136
5    2025
6    1681
7    5929
Name: Age, dtype: int64
```

2.5.3.2 Vectors with integers (scalars)

When you preform an operation on a vector using a scalar, the scalar will be recycled across all the elements in the vector.

```
print(ages + 100)
0    137
1    161
2    190
3    166
4    156
5    145
6    141
7    177
Name: Age, dtype: int64
```

```
print(ages * 2)
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

2.5.3.3 Vectors with different lengths

When you are working with vectors of different lengths, the behavior will depend on the `type` of the vectors.

With a `Series`, the vectors will preform an operation matched by the index. The rest of the resulting vector will be filled with a ‘missing’ value, this is denoted with a `NaN`, for ’not a number’.

This type of behavior is called ‘broadcasting’ and it differs between languages. Broadcasting in Pandas refers to how operations are calculated between arrays with different shapes.

```
print(ages + pd.Series([1, 100]))
 0      38.0
 1     161.0
 2       NaN
 3       NaN
 4       NaN
 5       NaN
 6       NaN
 7       NaN
 dtype: float64
```

With other `types`, the shapes must match.

```
import numpy as np
print(ages + np.array([1, 100]))

 <class 'ValueError'>
 operands could not be broadcast together with shapes (8,) (2,)
```

2.5.3.4 Vectors with common index labels

What’s cool about Pandas is how data alignment is almost always automatic. If possible, things will always align themselves with the index label when actions are performed.

```
# ages as they appear in the data
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```
rev_ages = ages.sort_index(ascending=False)
print(rev_ages)
7    77
6    41
5    45
4    56
3    66
2    90
1    61
0    37
Name: Age, dtype: int64
```

If we perform an operation using the `ages` and `reverse_ages`, it will sill be conducted element-by-element, however, the vectors will be aligned first before the operation is carried out.

```
# reference output
# to show index label alignment
print(ages * 2)
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```
# note how we get the same values
# even though the vector is reversed
print(ages + reverse_ages)

<class 'NameError'>
name 'reverse_ages' is not defined
```

2.6 The `DataFrame`

The `DataFrame` is the most common `Pandas` object. It can be thought of as Python's way of storing spreadsheet-like data.

Many of the common features with the `Series` carry over into the `DataFrame`.

2.6.1 Boolean subsetting DataFrame

Just like how we were able to subset a `Series` with a boolean vector, we can subset a `DataFrame` with a `bool`.

```
# Boolean vectors will subset rows
print(scientists[scientists['Age'] > scientists['Age'].mean()])
                   Name        Born        Died  Age     Occup
 1        William Gosset  1876-06-13  1937-10-16   61   Statist
 2  Florence Nightingale  1820-05-12  1910-08-13   90
 3           Marie Curie  1867-11-07  1934-07-04   66        Ch
 7         Johann Gauss  1777-04-30  1855-02-23   77  Mathemat
```

Table 2-1: Table of dataframe subsetting methods

Syntax	Selection Result
`df[column name]`	Single column
`df [[ column1, column2, ... ]]`	Multiple columns
`df. loc [ row label ]`	Row by row index label (row name)
`df. loc [[ label1 , label2 , ...]]`	Multiple rows by index label

`df. iloc [row number]`	Row by row number
`df. iloc [[ row1, row2, ...]]`	Multiple rows by row number
`df. ix [ label or number]`	Row by index label or number
`df. ix [[ lab num1, lab num2, ...]]`	Multiple rows by index label or number
`df[bool]`	Row based on bool
`df [[ bool1, bool2, ...]]`	Multiple rows based on bool
`df[ start :stop: step ]`	Rows based on slicing notation

Because of how broadcasting works, if we supply a `bool` vector that is not the same as the number of rows in the dataframe, the maximum possible rows returned would be the length of the `bool` vector.

```
# 4 values passed as a bool vector
# 3 rows returned
print(scientists.ix[[True, True, False, True]])

                   Name        Born        Died  Age    Occupatic
 0   Rosaline Franklin  1920-07-25  1958-04-16   37        Chemis
 1      William Gosset  1876-06-13  1937-10-16   61  Statisticia
 3         Marie Curie  1867-11-07  1934-07-04   66        Chemis
```

To fully summarize all the various subsetting methods:

2.6.2 Operations are automatically aligned and vectorized

NOT SURE IF I NEED THIS SECTION. OTHERWISE NEED TO FIND ANOTHER DATASET

```
first_half = second_half
scientists[: 4] = scientists[ 4 :]
print(first_half)

                   Name        Born        Died  Age    Occupa
 0     Rosaline Franklin  1920-07-25  1958-04-16   37       Che
 1        William Gosset  1876-06-13  1937-10-16   61  Statist:
 2  Florence Nightingale  1820-05-12  1910-08-13   90         N
 3           Marie Curie  1867-11-07  1934-07-04   66       Che

print(second_half)

            Name        Born        Died  Age          Occupat
 4  Rachel Carson  1907-05-27  1964-04-14   56           Biolog
 5      John Snow  1813-03-15  1858-06-16   45           Physic
 6    Alan Turing  1912-06-23  1954-06-07   41  Computer Scient
 7   Johann Gauss  1777-04-30  1855-02-23   77       Mathematic

print(first_half + second_half)

   Name Born Died  Age Occupation
 0  NaN  NaN  NaN  NaN        NaN
 1  NaN  NaN  NaN  NaN        NaN
 2  NaN  NaN  NaN  NaN        NaN
 3  NaN  NaN  NaN  NaN        NaN
 4  NaN  NaN  NaN  NaN        NaN
 5  NaN  NaN  NaN  NaN        NaN
 6  NaN  NaN  NaN  NaN        NaN
 7  NaN  NaN  NaN  NaN        NaN

print(scientists * 2)

                                        Name                 E
 0        Rosaline FranklinRosaline Franklin  1920-07-251920-07
 1              William GossetWilliam Gosset  1876-06-131876-06
 2  Florence NightingaleFlorence Nightingale  1820-05-121820-05
 3                    Marie CurieMarie Curie  1867-11-071867-11
 4                Rachel CarsonRachel Carson  1907-05-271907-05
 5                        John SnowJohn Snow  1813-03-151813-03
 6                    Alan TuringAlan Turing  1912-06-231912-06
 7                  Johann GaussJohann Gauss  1777-04-301777-04

                   Died  Age                        Occupa
 0  1958-04-161958-04-16   74                    ChemistChe
```

```
1  1937-10-161937-10-16  122            StatisticianStatist:
2  1910-08-131910-08-13  180                          NurseN
3  1934-07-041934-07-04  132                       ChemistChe
4  1964-04-141964-04-14  112                    BiologistBiolo
5  1858-06-161858-06-16   90                    PhysicianPhys:
6  1954-06-071954-06-07   82  Computer ScientistComputer Scier
7  1855-02-231855-02-23  154             MathematicianMathemat:
```

2.7 Making changes to `Series` and `DataFrames`

2.7.1 Add additional columns

Now that we know various ways of subsetting and slicing our data (See table 2–1), we should now be able to find values of interest to assign new values to them.

The `type` of the `Born` and `Died` columns are `object`s, meaning they are strings.

```
print(scientists['Born'].dtype)
```

```
object
```

```
print(scientists['Died'].dtype)
```

```
object
```

We can convert the strings to a proper `datetime` type so we can perform common datetime operations (e.g., take differences between dates or calculate the age). You can provide your own `format` if you have a date that has a specific `format`. A list of format variables can be found in the Python `datetime` module documentation[5]. The format of our date looks like `"YYYY-MM-DD", so we can use the '%Y-%m-%d' format.`

```
# format the 'Born' column as a datetime
born_datetime = pd.to_datetime(scientists['Born'], format='%Y-%
print(born_datetime)
  0  1920-07-25
  1  1876-06-13
  2  1820-05-12
  3  1867-11-07
  4  1907-05-27
```

```
  5   1813-03-15
  6   1912-06-23
  7   1777-04-30
  Name: Born, dtype: datetime64[ns]

# format the 'Died' column as a datetime
died_datetime = pd.to_datetime(scientists['Died'], format='%Y-%
```

If we wanted, we can create a new set of columns that contain the `datetime` representations of the `object` (string) dates.

```
scientists['born_dt'], scientists['died_dt'] = (born_datetime,
                                                died_datetime)
print(scientists.head())

                    Name        Born        Died  Age     Occupa
 0     Rosaline Franklin  1920-07-25  1958-04-16   37        Che
 1        William Gosset  1876-06-13  1937-10-16   61  Statist:
 2  Florence Nightingale  1820-05-12  1910-08-13   90          I
 3           Marie Curie  1867-11-07  1934-07-04   66        Che
 4         Rachel Carson  1907-05-27  1964-04-14   56      Biolc

       died_dt
 0  1958-04-16
 1  1937-10-16
 2  1910-08-13
 3  1934-07-04
 4  1964-04-14
print(scientists.shape)
 (8, 7)
```

[5] https://docs.python.org/3.5/library/datetime.html#strftime-and-strptime-behavior

2.7.2 Directly change a column

One way to look at variable importance is to see what happens when you randomly scramble a column. (TODO RANDOM FOREST VIPS)

```
import random
random.seed(42)
random.shuffle(scientists['Age'])
```

You'll notice that the `random.shuffle` method seems to work directly on the column. If you look at the documentation for `random.shuffle`[6] it will mention that the sequence will be shuffled 'in place'. Meaning it will work directly on the sequence. Contrast this with the previous method where we assigned the newly calculated values to a separate variable before we can assign it to the column.

We can recalculate the 'real' age using `datetime` arithmetic.

[6] https://docs.python.org/3.5/library/random.html#random.shuffle

```
# subtracting dates will give us number of days
scientists['age_days_dt'] = (scientists['died_dt'] - scientists
print(scientists)
```

```
                   Name        Born        Died  Age
0     Rosaline Franklin  1920-07-25  1958-04-16   66
1        William Gosset  1876-06-13  1937-10-16   56          St
2  Florence Nightingale  1820-05-12  1910-08-13   41
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   45
6           Alan Turing  1912-06-23  1954-06-07   37  Compute
7          Johann Gauss  1777-04-30  1855-02-23   61         Mat

     born_dt     died_dt  age_days_dt
0 1920-07-25  1958-04-16   13779 days
1 1876-06-13  1937-10-16   22404 days
2 1820-05-12  1910-08-13   32964 days
3 1867-11-07  1934-07-04   24345 days
4 1907-05-27  1964-04-14   20777 days
5 1813-03-15  1858-06-16   16529 days
6 1912-06-23  1954-06-07   15324 days
7 1777-04-30  1855-02-23   28422 days
```

```
# we can convert the value to just the year
# using the astype method
scientists['age_years_dt'] = scientists['age_days_dt'].astype('
print(scientists)
```

```
                   Name        Born        Died  Age
0     Rosaline Franklin  1920-07-25  1958-04-16   66
1        William Gosset  1876-06-13  1937-10-16   56          St
2  Florence Nightingale  1820-05-12  1910-08-13   41
```

```
3           Marie Curie  1867-11-07  1934-07-04   77
4        Rachel Carson  1907-05-27  1964-04-14   90
5            John Snow  1813-03-15  1858-06-16   45
6          Alan Turing  1912-06-23  1954-06-07   37  Compute
7         Johann Gauss  1777-04-30  1855-02-23   61        Mat

      born_dt     died_dt  age_days_dt  age_years_dt
0  1920-07-25  1958-04-16   13779 days          37.0
1  1876-06-13  1937-10-16   22404 days          61.0
2  1820-05-12  1910-08-13   32964 days          90.0
3  1867-11-07  1934-07-04   24345 days          66.0
4  1907-05-27  1964-04-14   20777 days          56.0
5  1813-03-15  1858-06-16   16529 days          45.0
6  1912-06-23  1954-06-07   15324 days          41.0
7  1777-04-30  1855-02-23   28422 days          77.0
```

Note

We could've directly assigned the column to the `datetime` converted, but the point is an assignment still needed to be preformed. The `random.shuffle` example preforms its method 'in place', so there is nothing that is explicitly returned from the function. The value passed into the function is directly manipulated.

2.8 Exporting and importing data

2.8.1 `pickle`

2.8.1.1 `Series`

Many of the export methods for a `Series` are also available for a `DataFrame`. Those who have experience with `numpy` will know there is a `save` method on `ndarrays`. This method has been deprecated, and the replacement is to use the `to_pickle` method in its place.

```
names = scientists['Name']
print(names)
0       Rosaline Franklin
1          William Gosset
```

```
 2    Florence Nightingale
 3             Marie Curie
 4           Rachel Carson
 5               John Snow
 6             Alan Turing
 7            Johann Gauss
 Name: Name, dtype: object
```

```
#  pass in a string to the path you want to save
names.to_pickle('../output/scientists_names_series.pickle')
```

The pickle output is in a binary format, meaning if you try to open it in a text editor, you will see a bunch of garbled characters.

If the object you are saving is an intermediate step in a set of calculations that you want to save, or if you know your data will stay in the Python world, saving objects to a `pickle`, will be optimized for Python as well as disk storage space. However, this means that people who do not use Python, will not be able to read the data.

2.8.1.2 `DataFrame`

The same method can be used on `DataFrame` objects.

```
scientists.to_pickle('../output/scientists_df.pickle')
```

2.8.1.3 Reading pickel data

To read in `pickel` data we can use the `pd. read_pickle` function.

```
# for a Series
scientist_names_from_pickle = pd.read_pickle('../output/scient:
```

```
 0       Rosaline Franklin
 1          William Gosset
 2    Florence Nightingale
 3             Marie Curie
 4           Rachel Carson
 5               John Snow
 6             Alan Turing
 7            Johann Gauss
 Name: Name, dtype: object
```

```
# for a DataFrame
scientists_from_pickle = pd.read_pickle('../output/scientists_(
print(scientists_from_pickle)

                    Name        Born        Died  Age          
0      Rosaline Franklin  1920-07-25  1958-04-16   66          
1         William Gosset  1876-06-13  1937-10-16   56         St
2   Florence Nightingale  1820-05-12  1910-08-13   41          
3            Marie Curie  1867-11-07  1934-07-04   77          
4          Rachel Carson  1907-05-27  1964-04-14   90          
5              John Snow  1813-03-15  1858-06-16   45          
6            Alan Turing  1912-06-23  1954-06-07   37  Compute:
7           Johann Gauss  1777-04-30  1855-02-23   61        Mat

     born_dt     died_dt  age_days_dt  age_years_dt
0 1920-07-25  1958-04-16   13779 days          37.0
1 1876-06-13  1937-10-16   22404 days          61.0
2 1820-05-12  1910-08-13   32964 days          90.0
3 1867-11-07  1934-07-04   24345 days          66.0
4 1907-05-27  1964-04-14   20777 days          56.0
5 1813-03-15  1858-06-16   16529 days          45.0
6 1912-06-23  1954-06-07   15324 days          41.0
7 1777-04-30  1855-02-23   28422 days          77.0
```

You will see `pickle` files saved as `.p`, `. pkl`, or `. pickle`.

2.8.2 CSV

Comma-separated values (CSV) are the most flexible data storage type. For each row, the column information will be separated with a comma. The comma is not the only type of delimiter. Some files will be delimited by a tab (tsv), or even a semi-colon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open it. It can even be opened in a text editor.

The `Series` and `DataFrame` have a `to_csv` method to write a CSV file.

The documentation for `Series`[7] and `DataFrame`[8] have many different ways you can modify the resulting CSV file. For example, if you wanted to save a TSV file because there are commas in your data, you can set the `sep` parameter to `'t'` (TODO USING FUNCTIONS).

```
# save a series into a CSV
names.to_csv('../output/scientist_names_series.csv')

# save a dataframe into a TSV,
# a tab-separated value
scientists.to_csv('../output/scientists_df.tsv', sep='\t')
```

Removing row number from output If you open the CSV or TSV file created, you will notice that the first 'column' will look like the row number of the dataframe. Many times this is not needed, especially when collaborating with other people. However, keep in mind, it is really saving the 'row label', which may be important.

The documentation[9] will show that there is a `index` parameter that to write row names (index).

```
scientists.to_csv('../output/scientists_df_no_index.csv', inde
```

Importing CSV data Importing CSV files was shown in Chapter 1.4. It uses the `pd.read_csv` function. From the documentation[10], you can see there are various ways you can read in a CSV. You can see TODO USING FUNCTIONS of you need more information on using function parameters

[7] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.to_csv.html

[8] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

[9] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

[10] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

2.8.3 Excel

Excel, probably the most common data type (or second most common, next to

CSVs). Excel has a bad reputation within the data science community. I discussed some of the reasons why in Chapter 1.1. The goal of this book isn't to bash Excel, but to teach you a resonable alternative tool for data analytics. In short, the more you can do your work in a scripting language, the easier it will be to scale up to larger projects, catch and fix mistakes, and collaborate. Excel has its own scripting language if you absolutely have to work in it.

2.8.3.1 `Series`

The `Series` does not have an explicit `to_excel` method. If you have a `Series` that needs to be exported to an Excel file. One way is to convert the `Series` into a 1 column `DataFrame`.

```
# convert the Series into a DataFrame
# before saving it to an excel file
names_df = names.to_frame()

# xls file
names_df.to_excel('../output/scientists_names_series_df.xls')

# newer xlsx file
names_df.to_excel('../output/scientists_names_series_df.xlsx')
```

2.8.3.2 DataFrame

From above, you can see how to export a `DataFrame` to an Excel file. The documentation[11] does show ways on how to further fine tune the output. For example, you can output to a specific 'sheet' using the `sheet_name` parameter

```
# saving a DataFrame into Excel format
scientists.to_excel('../output/scientists_df.xlsx',
                    sheet_name='scientists',
                    index=False)
```

2.8.4 Many data output types

There are many ways `Pandas` can export and import data, `to_pickle`, `to_csv`, and `to_excel`, are only a fraction of the dataformats that can make its way into `Pandas DataFrames`.

[11] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_excel.html

Export method	Description
`to_clipboard`	save data into the system clipboard for pasting
`to_dense`	convert data into a regular 'dense' `DataFrame`
`to_dict`	convert data into a `Python dict`
`to_gbq`	convert data into a Google BigQuery table
`toJidf`	save data into a hierarchal data format (HDF)
`to_msgpack`	save data into a portable JSON-like binary
`toJitml`	convert data to a HTML table
`tojson`	convert data into a JSON string
`toJatex`	convert data as a LTEXtabular environment
`to_records`	convert data into a record array
`to_string`	show `DataFrame` as a string for `stdout`

`to_sparse`	convert data into a `SparceDataFrame`
`to_sql`	save data into a SQL database
`to_stata`	convert data into a Stata `dta` file

For more complicated and general data conversions (not necessarily just exporting), the `odo` library[12] has a consistent way to convert between data formats. TODO CHAPTER ON DATA AND ODO.

2.9 Conclusion

This chapter went in a little more detail about how the `Pandas Series` and `DataFrame` objects work in `Python`. There were some simpler examples of data cleaning shown, and a few common ways to export data to share with others. Chapters 1 and 2 should give you a good basis on how `Pandas` as a library works.

The next chapter will cover the basics of plotting in `Pytho` and `Pandas`. Data visualization is not only used in the end of an analysis to plot results, it is heavily utilized throughout the entire data pipeline.

12 http://ocLo.readthedocs.org/en/latest/

Chapter 3. Introduction to Plotting

3.1 Introduction

Data visualization is as much a part of the data processing step as the data presentation step. It is much easier to compare values when they are plotted than numeric values. By visualizing data we are able to get a better intuitive sense of our data, than by looking at tables of values alone. Additionally, visualizations can also bring to light, hidden patterns in data, that you, the analyst, can exploit for model selection.

3.2 Concept map

1. Prior knowledge

(a) Containers

(b) Using functions

(c) Subsetting and indexing

(d) Classes

2. matplotlib

3. seaborn

3.3 Objectives

This chapter will cover:

1. matplotlib

2. seaborn

3. plotting in pandas

The quintessential example for making visualizations of data is Anscombe's quartet. This was a dataset created by English statistician Frank Anscombe to show the importance of statistical graphs.

The Anscombe dataset contains 4 sets of data, where each set contains 2 continuous variables. Each set has the same mean, variance, correlation, and regression line. However, only when the data are visualized is it obvious that each set does not follow the same pattern. This goes to show the benefits of visualizations and the pitfalls of only looking at summary statistics.

```
# the anscombe dataset can be found in the seaborn library
import seaborn as sns
anscombe = sns.load_dataset("anscombe")
print(anscombe)
     dataset     x      y
 0         I  10.0   8.04
 1         I   8.0   6.95
 2         I  13.0   7.58
 3         I   9.0   8.81
 4         I  11.0   8.33
 5         I  14.0   9.96
 6         I   6.0   7.24
 7         I   4.0   4.26
 8         I  12.0  10.84
 9         I   7.0   4.82
 10        I   5.0   5.68
 11       II  10.0   9.14
 12       II   8.0   8.14
 13       II  13.0   8.74
 14       II   9.0   8.77
 15       II  11.0   9.26
 16       II  14.0   8.10
 17       II   6.0   6.13
 18       II   4.0   3.10
 19       II  12.0   9.13
 20       II   7.0   7.26
 21       II   5.0   4.74
 22      III  10.0   7.46
 23      III   8.0   6.77
 24      III  13.0  12.74
 25      III   9.0   7.11
 26      III  11.0   7.81
 27      III  14.0   8.84
```

```
28      III   6.0   6.08
29      III   4.0   5.39
30      III  12.0   8.15
31      III   7.0   6.42
32      III   5.0   5.73
33       IV   8.0   6.58
34       IV   8.0   5.76
35       IV   8.0   7.71
36       IV   8.0   8.84
37       IV   8.0   8.47
38       IV   8.0   7.04
39       IV   8.0   5.25
40       IV  19.0  12.50
41       IV   8.0   5.56
42       IV   8.0   7.91
43       IV   8.0   6.89
```

3.4 `matplotlib`

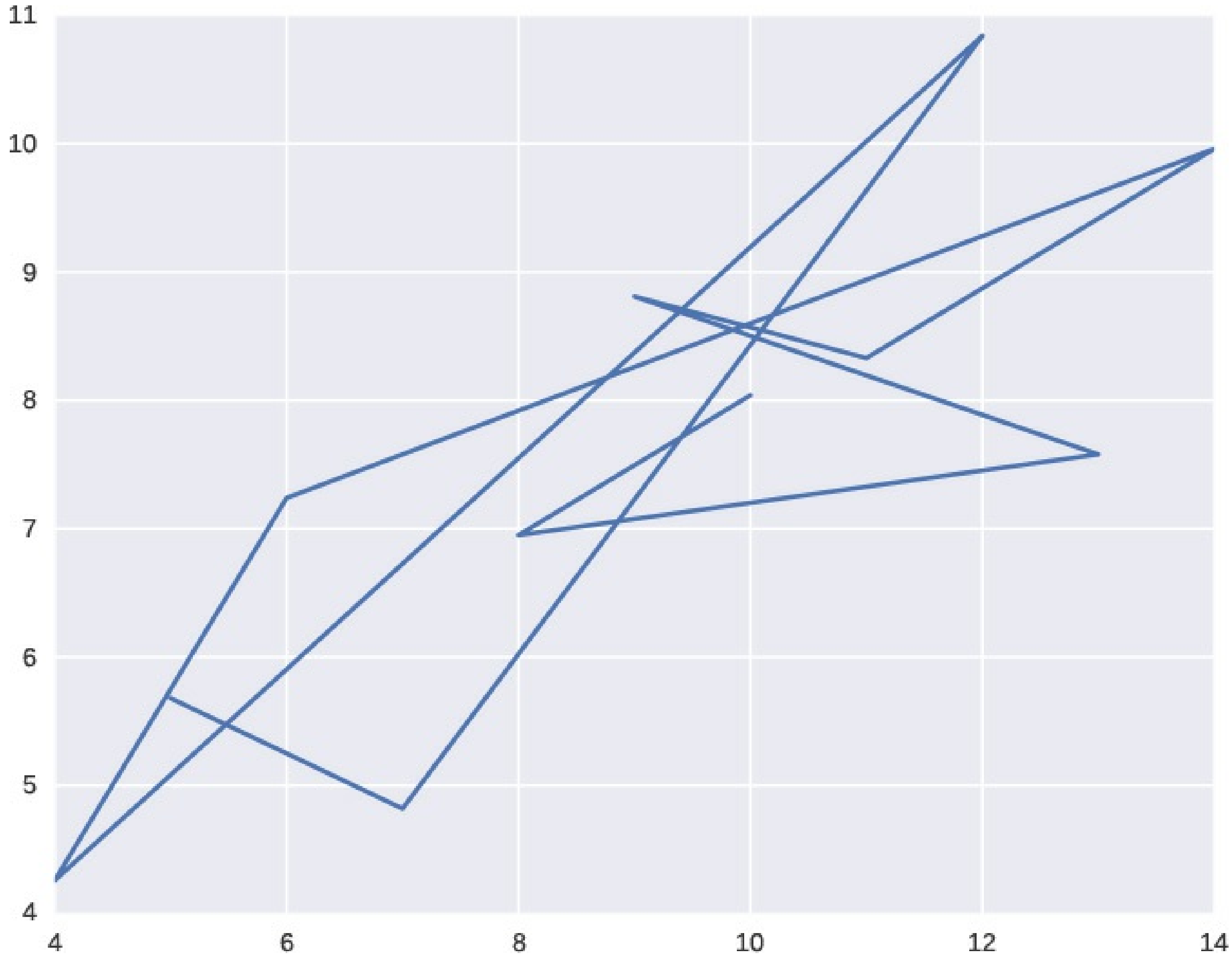
`matplotlib` is Python's fundamental plotting library. It is extremely flexible and gives the user full control of all elements of the plot.

Importing `matplotlib`'s plotting features is a little different from our previous package imports. You can think of it as the package `matplotlib` and all the plotting utilities are under a subfolder (or sub package) called `pyplot`. Just like how we imported a package and gave it an abbreviated name, we can do the same with `matplotlib . pyplot`.

```
import matplotlib.pyplot as pit
```

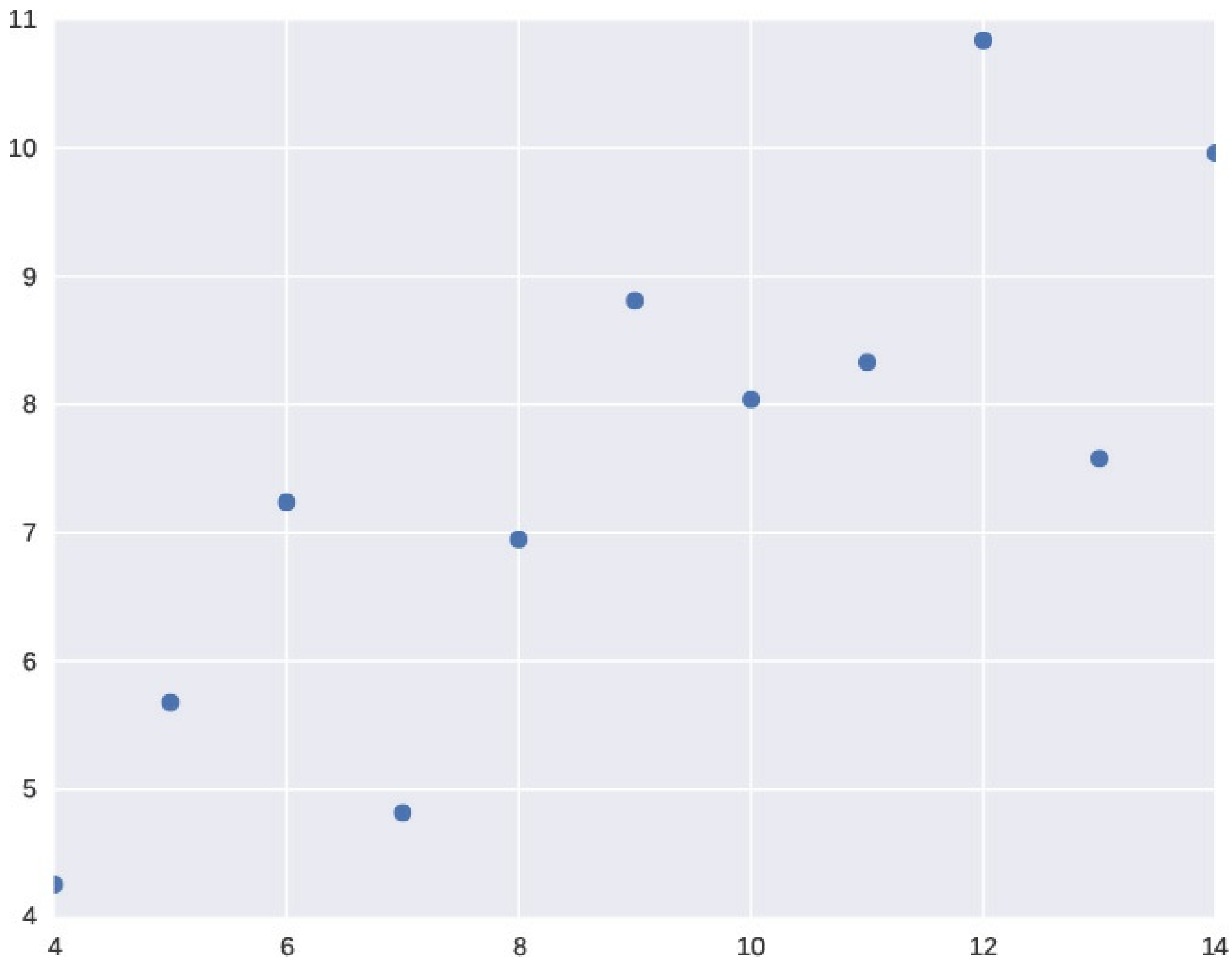
Most of the basic plots will start with `plt. plot`. In our example it takes a vector for the x-values, and a corresponding vector for the y-values.

```
# create a subset of the data
# contains only dataset 1  from anscombe
dataset_1 = anscombe[anscombe['dataset']   == 'I']

plt.plot(dataset_1['x'],  dataset_1['y'])
```



By default, `plt. plot` will draw lines. If we want it to draw circles (points) instead we can pass an `'o'` parameter to tell `plt. plot` to use points.

```
plt.plot(dataset_1['x'],   dataset_1['y'],   'o')
```



We can repeat this process for the rest of the `datasets` in our anscombe data.

```
# create subsets of the anscombe data
dataset_2 = anscombe[anscombe['dataset'] == 'II']
dataset_3 = anscombe[anscombe['dataset'] == 'III']
dataset_4 = anscombe[anscombe['dataset'] == 'IV']
```

Now, we could make these plots individually, one at a time, but `matplotlib` has a way to create subplots. That is, you can specify the dimensions of your final figure, and put in smaller plots to fit the specified dimensions. This way you can present your results in a single figure, instead of completely separate ones.

The `subplot` syntax takes 3 parameters.

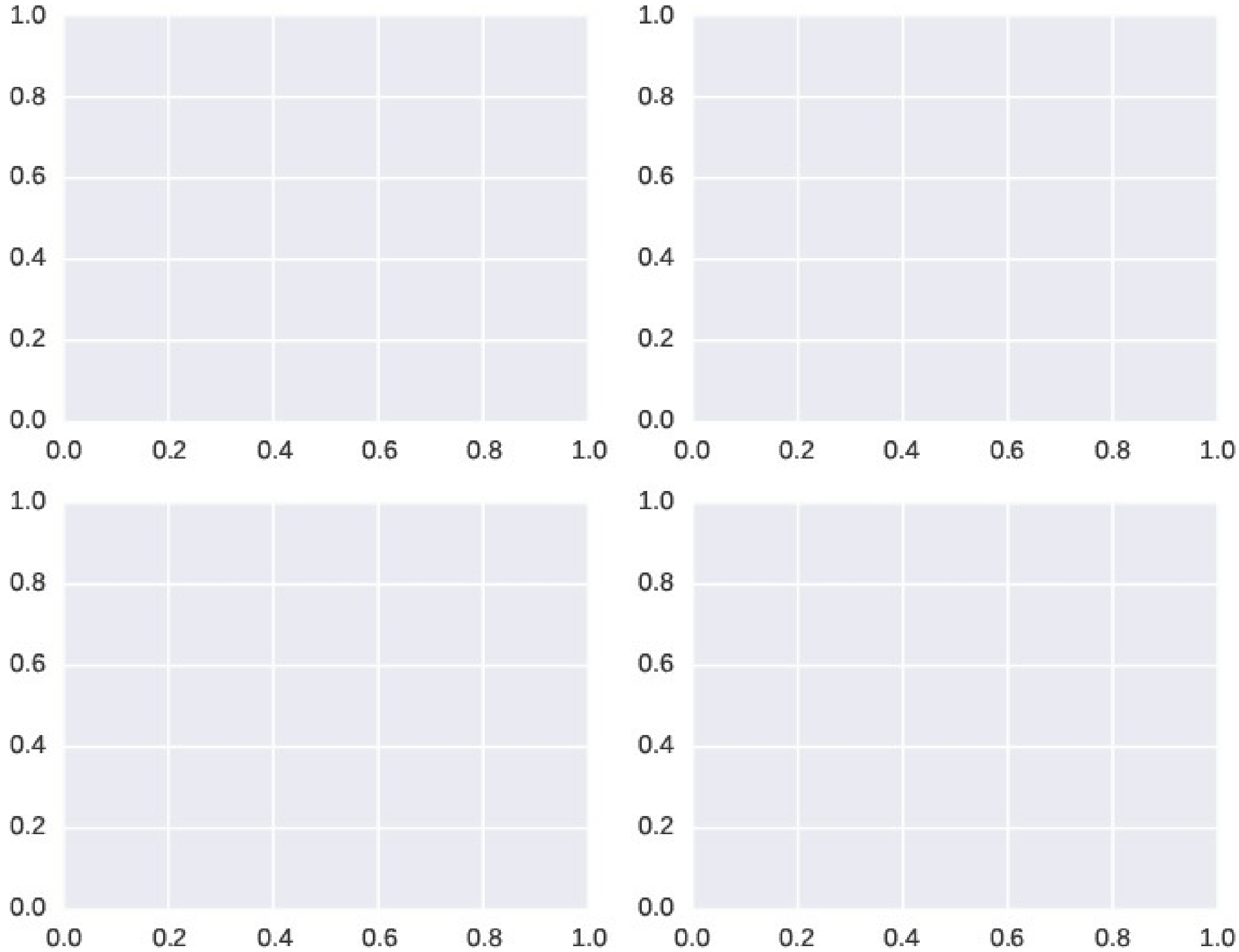
1. number of rows in figure for subplots

2. number of columns in figure for subplots

3. subplot location

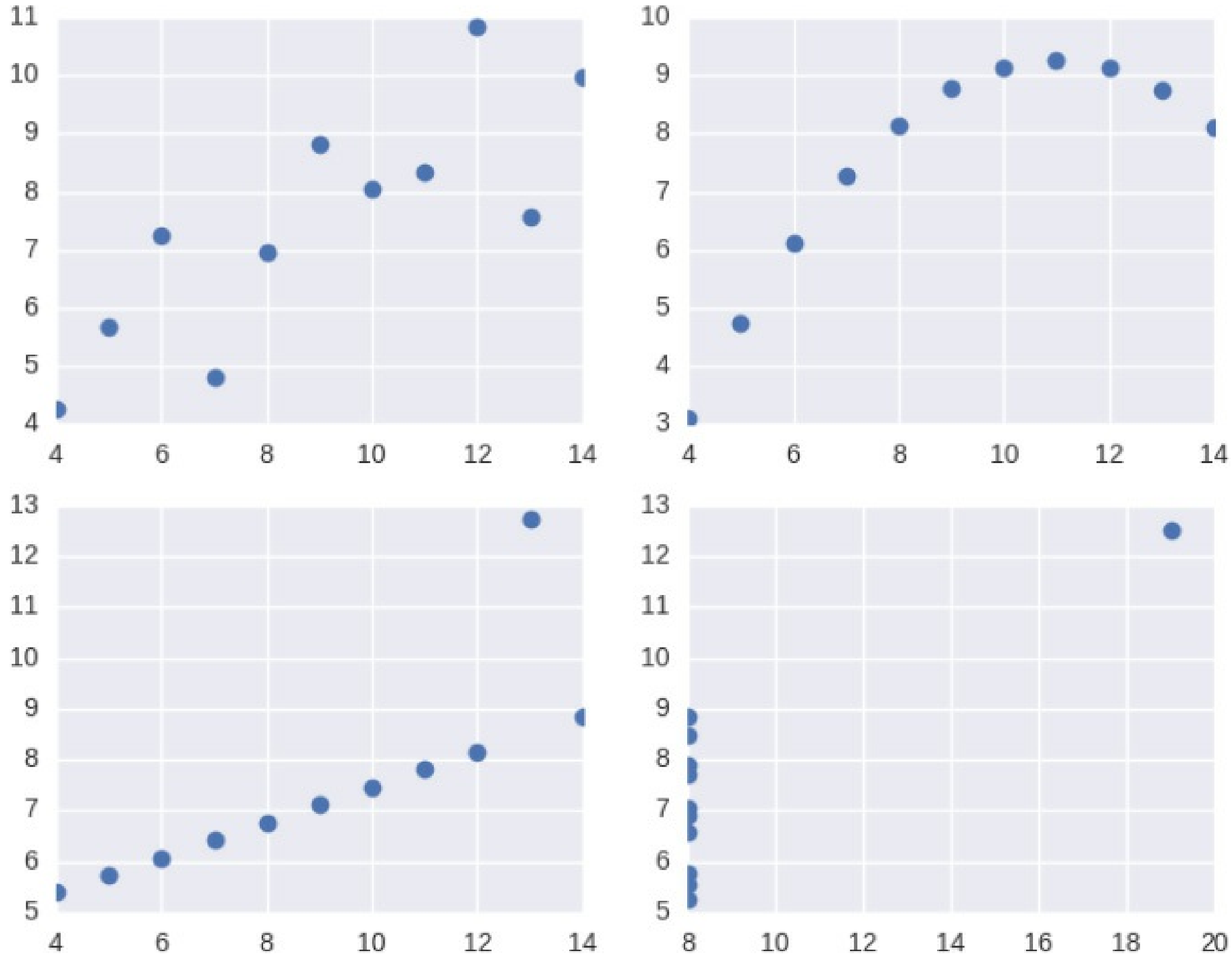
The subplot location is sequentially numbered and plots are placed left-to-right then top-to-bottom.

```
# create the entire figure where our subplots will go
fig = pit.figure()

# tell the figure how the subplots should be laid out
# in the example below we will have
# 2 row of plots,   each row will have 2 plots

# subplot has 2 rows and 2 columns, plot location 1
axesl = fig.add_subplot(2 , 2,   1)

# subplot has 2 rows and 2 columns, plot location 2
axes2 = fig.add_subplot(2 , 2,   2)

# subplot has 2 rows and 2 columns, plot location 3
axes3 = fig.add_subplot(2 , 2,   3)

# subplot has 2 rows and 2 columns, plot location 4
axes4 = fig.add_subplot(2 , 2,   4)
```



If we try to plot this now we will get an empty figure. All we have done so far is create a figure, and split the figure into a 2x2 grid where plots can be placed. Since no plots were created and inserted, nothing will show up.

```
# add a plot to each of the axes created above
axes1.plot(dataset_1['x'], dataset_1['y'], 'o')
axes2.plot(dataset_2['x'], dataset_2['y'], 'o')
axes3.plot(dataset_3['x'], dataset_3['y'], 'o')
axes4.plot(dataset_4['x'], dataset_4['y'],   'o')
```



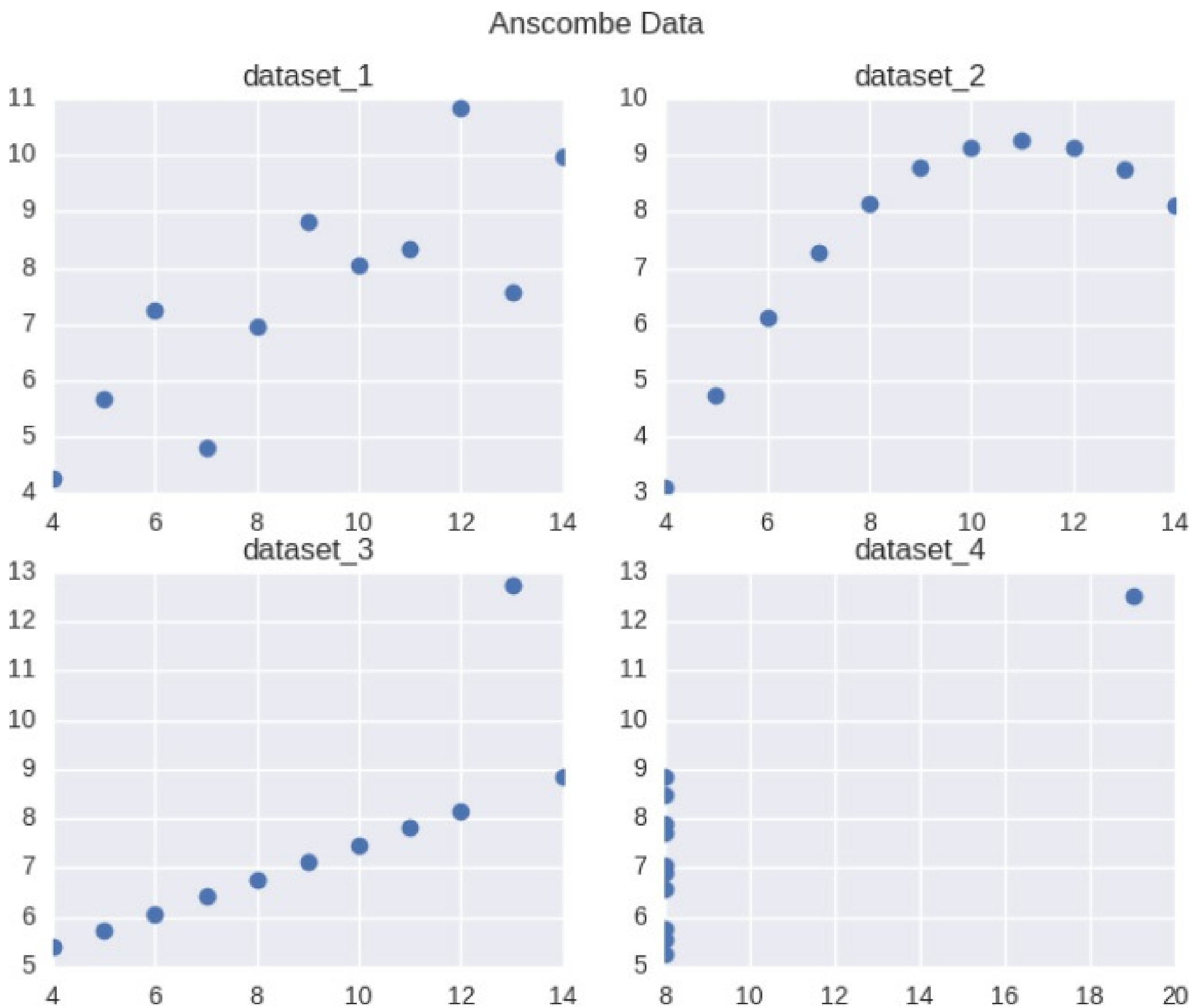
Finally, we can add a label to our subplots.

```
# add a small  title to each subplot
axesl.set_title("dataset_l")
axes2.set_title("dataset_2")
axes3.set_title("dataset_3")
axes4.set_title("dataset_4")

# add a title for the entire figure
fig.suptitle("Anscombe Data")
```

The anscombe data visualizations should depict why just looking at summary statistic values can be misleading. The moment the points were visualized, it becomes clear that even though each dataset has the same summary statistic values, the relationship between points vastly differ across datasets.

To finish off the anscombe example, we can add `setjdabel` () and `set_ylabel` () to each of the subplots to add x and y labels, just like how we added a title to the figure, f

Figure 3-1: Anscombe data visualization



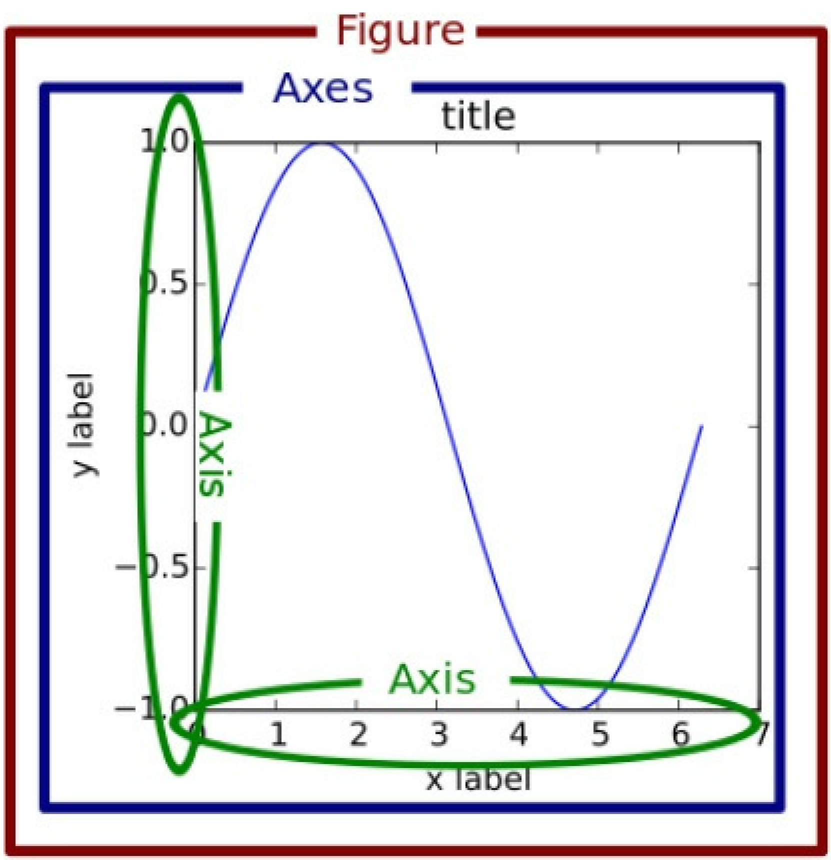
Before moving on and showing how to create more statistical plots, be familiar with the `matplotlib` documentation on "Parts of a Figure" [1]. I have reproduced their figure in Figure 3-2.

One of the most confusing parts of plotting in Python is the use of 'axis' and 'axes'. Especially when trying to verbally describe the different parts (since they are pronounced the same). In the anscombe example, each individual subplot plot was an axes. An axes has both an x and y axis. All 4 subplots make the figure.

The remainder of the chapter will show you how to create statistical plots, first with `matplotlib` and later using a higher-level plotting library based on `matplotlib` specifically made for statistical graphics, `seaborn`.

[1] http://matplotlib.org/faq/usage_faq.html#parts-of-a-figure

Figure 3-2: One of the most confusing parts of plotting in Python is the use of 'axis' and 'axes' since they are pronounced the same but refer to different parts of a figure



3.5 Statistical Graphics using matplotlib

The tips data we will be using for the next series of visualizations come from the seaborn library. This dataset contains the amount of tip people leave for various variables. For example, the total cost of the bill, the size of the party, the day of the week, the time of day, etc.

We can load this data just like the anscombe data above.

```
tips = sns.load_dataset("tips")
print(tips.head())
```

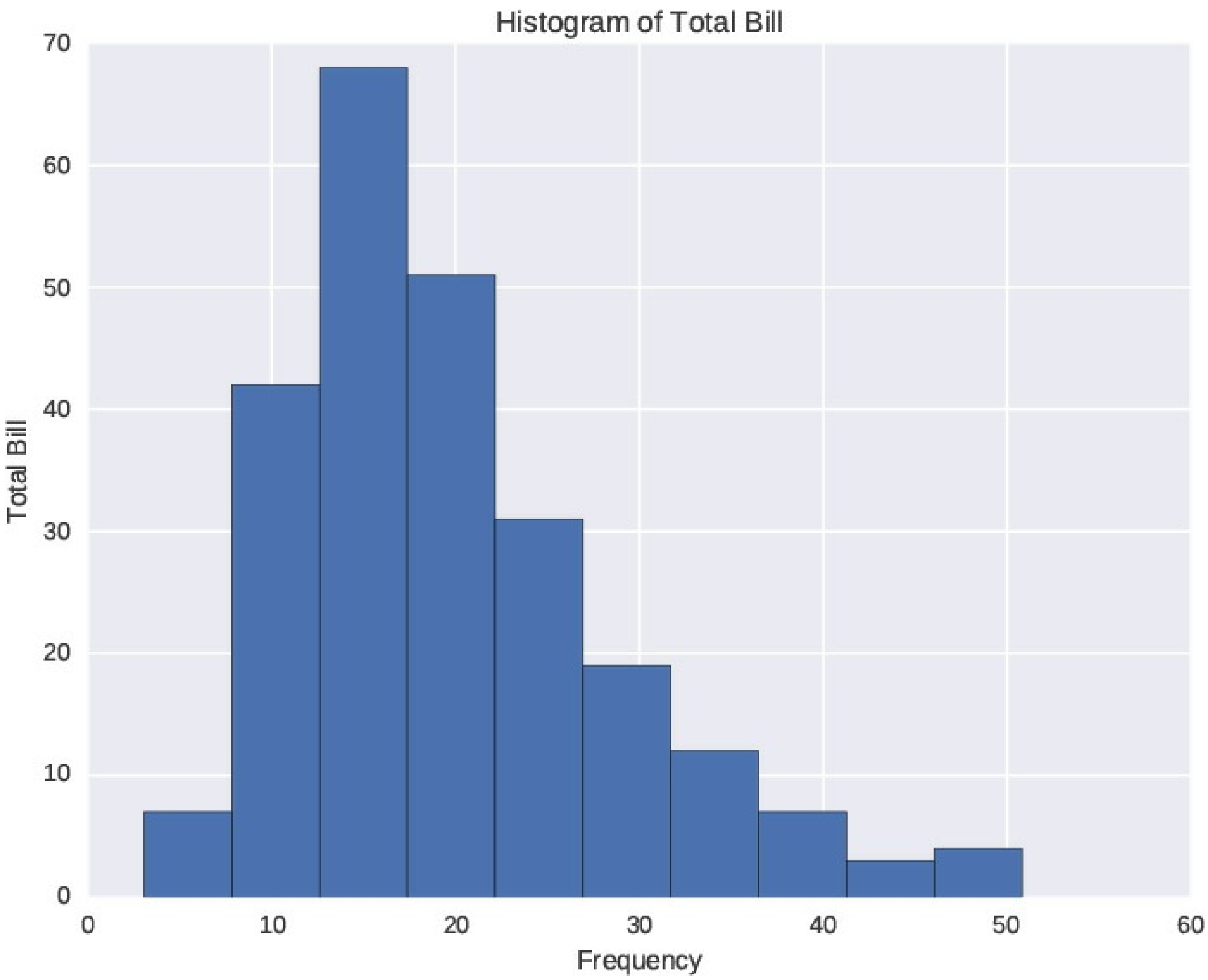
```
    total_bill   tip     sex smoker  day    time  size
 0       16.99  1.01  Female     No  Sun  Dinner     2
 1       10.34  1.66    Male     No  Sun  Dinner     3
 2       21.01  3.50    Male     No  Sun  Dinner     3
 3       23.68  3.31    Male     No  Sun  Dinner     2
 4       24.59  3.61  Female     No  Sun  Dinner     4
```

3.5.1 univariate

In statistics jargon, 'univariate' refers to a single variable. 3.5.1.1 Histograms

Histograms are the most common means of looking at a single variable. The values are 'binned', meaning they are grouped together and plotted to show the distribution of the variable.

```
fig = pit.figure()
axesl = fig.add_subplot(1, 1, 1)
axesl.hist(tips['total_bill'],  bins=10)
axesl.set_title('Histogram of Total Bill')
axesl.set_xlabel('Frequency' )
axesl.set_ylabel('Total Bill')
fig.show ()
```



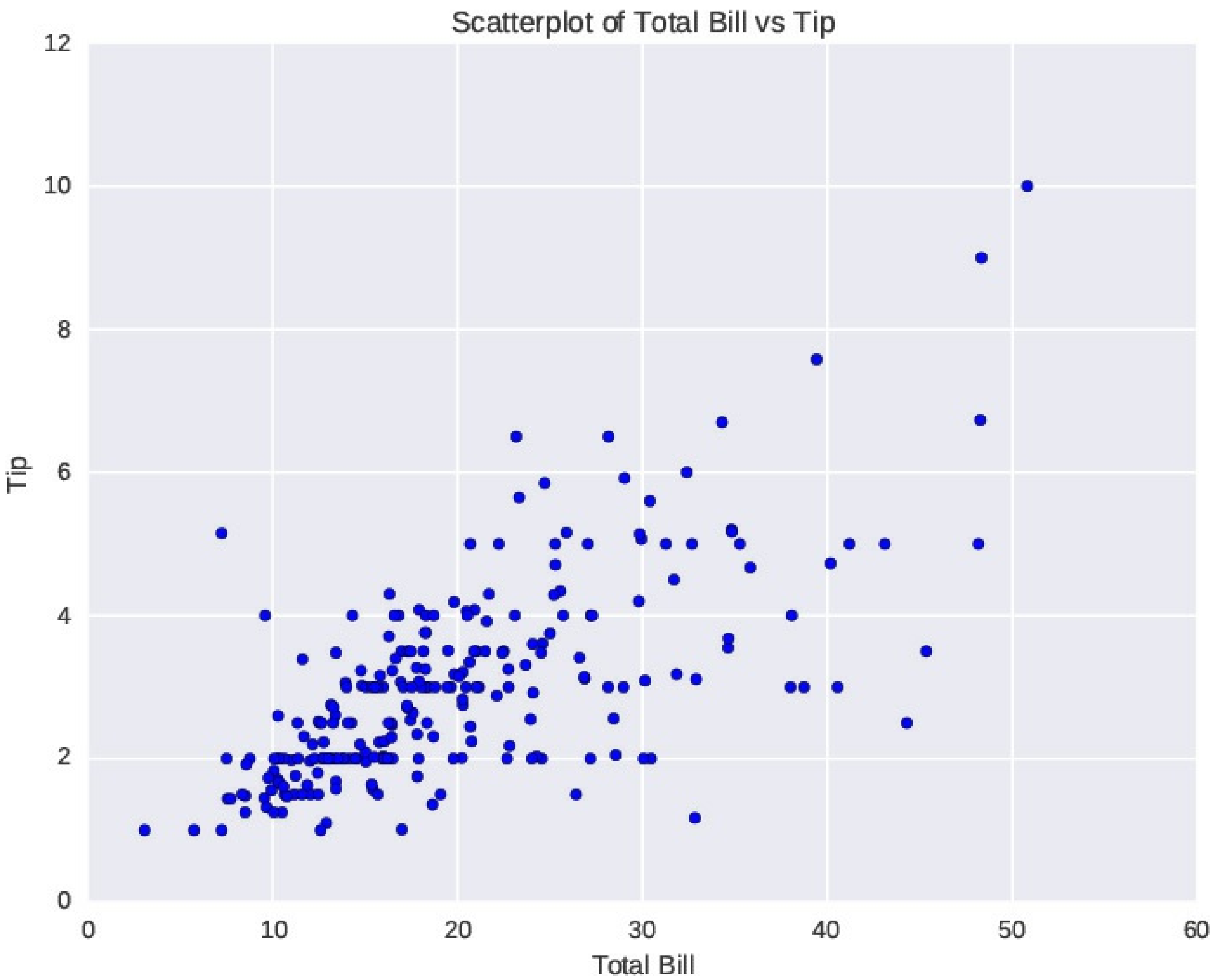
3.5.2 bivariate

In statistics jargon, 'bivariate' refers to a two variables.

3.5.2.1 Scatter plot

Scatter plots are used when a continuous variable is plotted against another continuous variable.

```
scatter_plot = plt.figure()
axesl = scatter_plot.add_subplot(1, 1, 1)
axesl.scatter(tips['total_bill'],  tips['tip'])
axesl.set_title('Scatterplot of Total Bill vs Tip')
axesl.set_xlabel('Total Bill')
axesl.set_ylabel('Tip') scatter_plot.show()
```

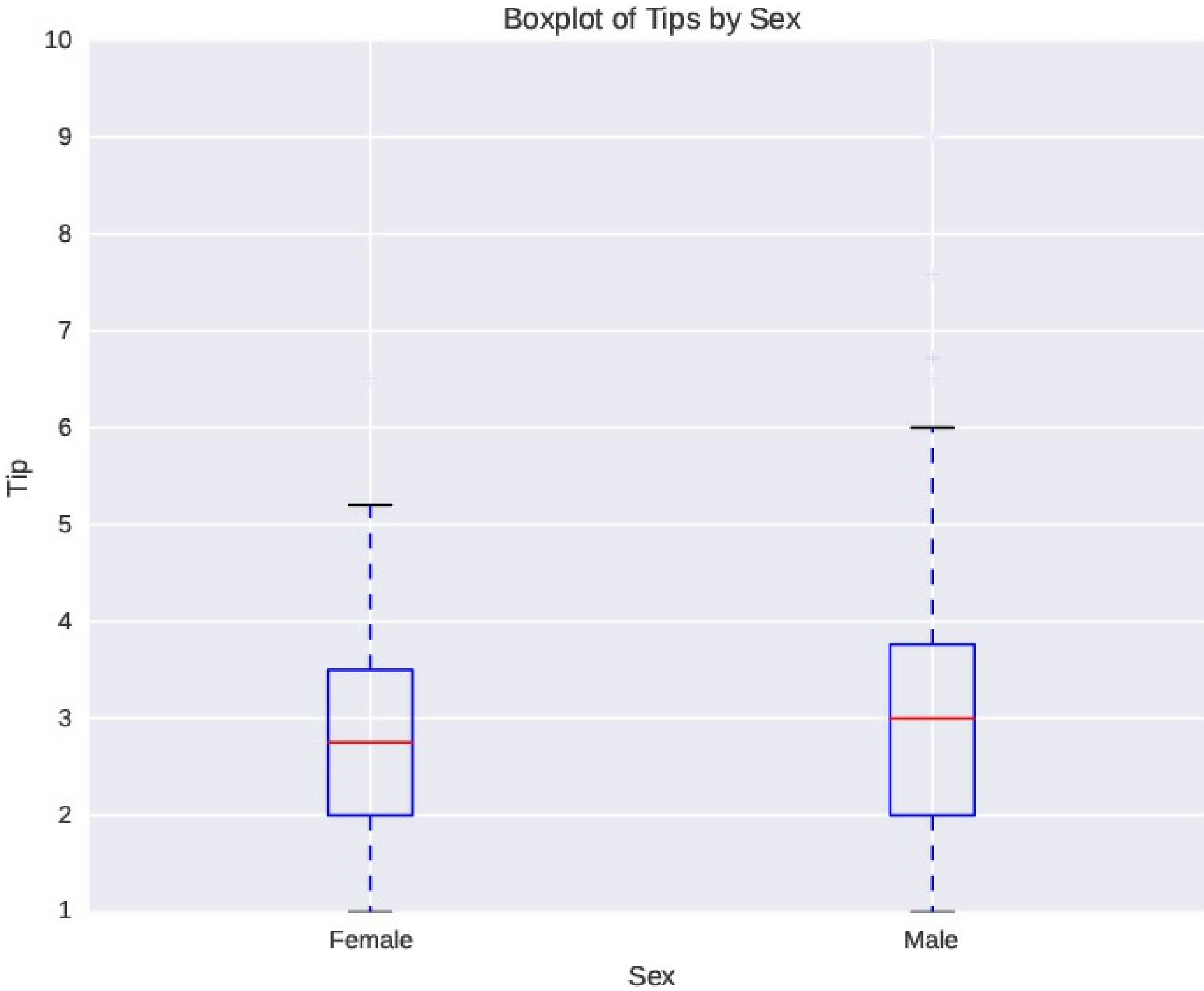


3.5.2.2 Box plot

Boxplots are used when a discrete variable is plotted against a continuous variable.

```
boxplot = pit.figure()
axesl = boxplot.add_subplot(1, 1, 1)
axesl.boxplot(
    # first argument of boxplot is the data
    # since we are plotting multiple pieces of data
    # we have to put each piece of data into a list
    [tips[tips['sex']   == 'Female']['tip'],
     tips [tips ['sex']   == 'Male']['tip']],
# We can then pass in an optional labels parameter
# to label  the data we passed labels=['Female',   'Male'])
axesl.set_xlabel('Sex')
axesl.set_ylabel('Tip')
```

```
axesl.set_title('Boxplot of Tips by Sex')
```



3.5.3 multivariate

Plotting multivariate data is tricky. There isn't a panacea or template that can be used for every case. Let's build on the scatter plot above. If we wanted to add another variable, say `sex`, one option would be to color the points by the third variable.

If we wanted to add a fourth variable, we could add size to the dots. The only caveat with using size as a variable is humans are not very good at differentiating areas. Sure, if there's an enormous dot next to a tiny one, your point will be conveyed, but smaller differences are hard to distinguish, and may add clutter to your visualization. One way to reduce clutter is to add some value of transparency to the individual points, this way many overlapping points will show a darker region of a plot than less crowded areas.

The general rule of thumb is different colors are much easier to distinguish than changes in size. If you have to use areas, be sure that you are actually plotting relative areas. A common pitfall is to use map a value to the radius of a circle for plots, but since the formula for a circle is ^{2}, your areas are actually on a squared scale, which is not only misleading, but wrong.

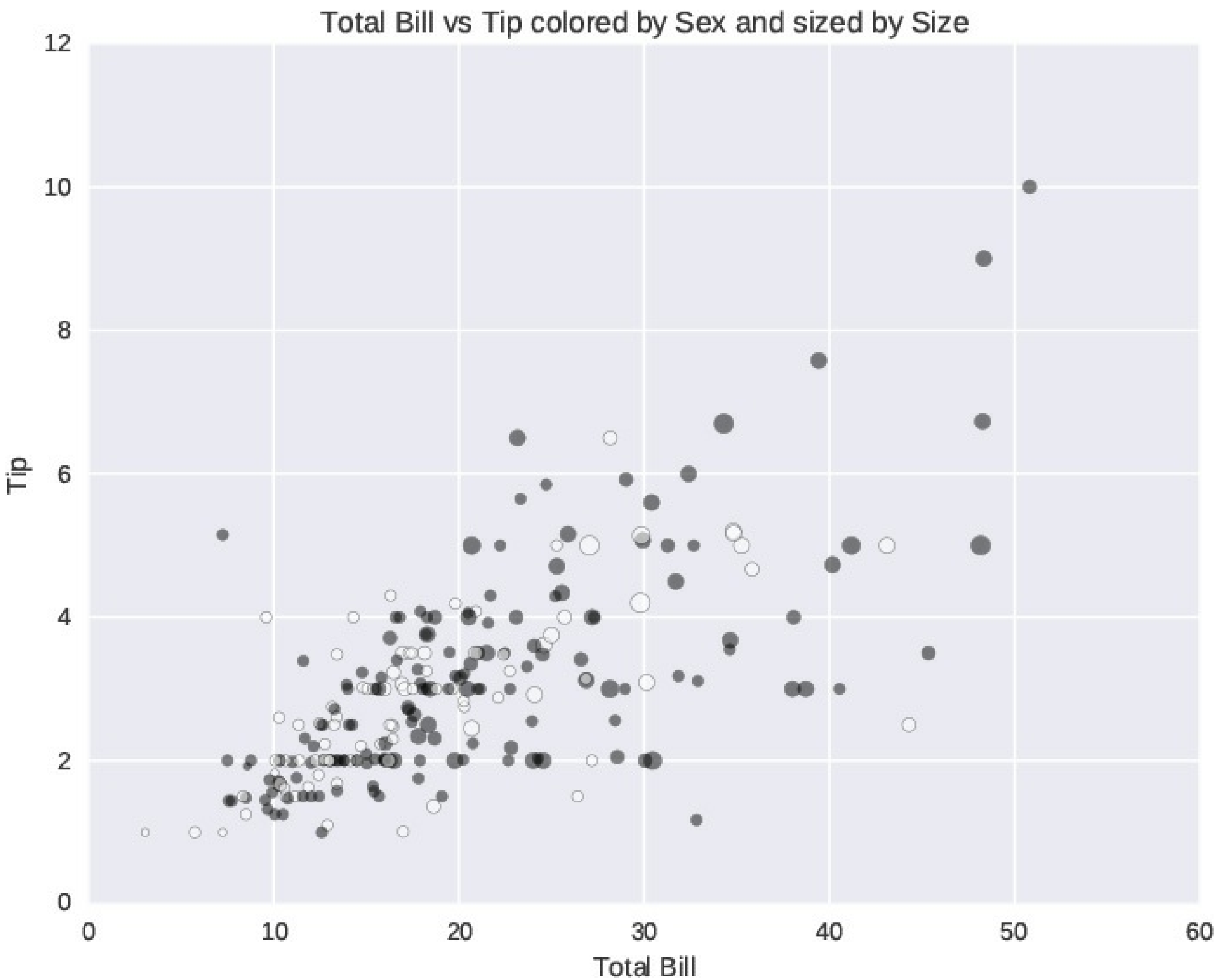
Colors are also difficult to pick. Humans do not perceive hues on a linear scale, so though also needs to go into picking color pallets. Luckily matplotlib [2] and seaborn [3] come with their own set of color pallets, and tools like colorbrewer [4] help with picking good color pallets.

```
# create a color variable based on the sex
def recode_sex(sex):
    if sex == 'Female':
       return 0
    else:
       return 1
tips['sex_color']   = tips['sex'].apply(recode_sex)

scatter_plot = plt.figure()
axesl = scatter_plot.add_subplot(1, 1, 1)
axesl.scatter(x=tips['total_bill'],
              y=tips['tip'],
              # set  the size of the dots based on party size
              # we multiply the values by 10 to make the point:
              # and also to emphasize the difference
              s=tips['size']   *  10,
              # set  the color for the sex
              c=tips['sex_color'],
              # set  the alpha so points are more transparent
              # this helps with overlapping points
              alpha=0.5)
axesl.set_title('Total Bill vs Tip colored by Sex and sized by
axesl.set_xlabel('Total Bill')
axesl.set_ylabel('Tip')
scatter_plot.show()
```

2 http://matplotlib.org/users/colormaps.html

3 http://stanford.edu/~mwaskom/software/seaborn-dev/tutorial/color_palettes.html

4 http://colorbrewer2.org/



3.6 seaborn

`matplotlib` can be thought of as the core foundational plotting tool in Python, `seaborn` builds on `matplotlib` by providing a higher level interface for statistical graphics. It provides an interface to produce prettier and more complex visualizations with fewer lines of code.

`seaborn` is also tightly integrated with `pandas` and the rest of the PyData stack (numpy pandas, scipy, statsmodels), making visualizations from any part of the data analysis process a breeze. Since `seaborn` is built on top of `matplotlib`, the user still has the ability to fine tune the visualizations.

We've already loaded the `seaborn` library for its datasets.

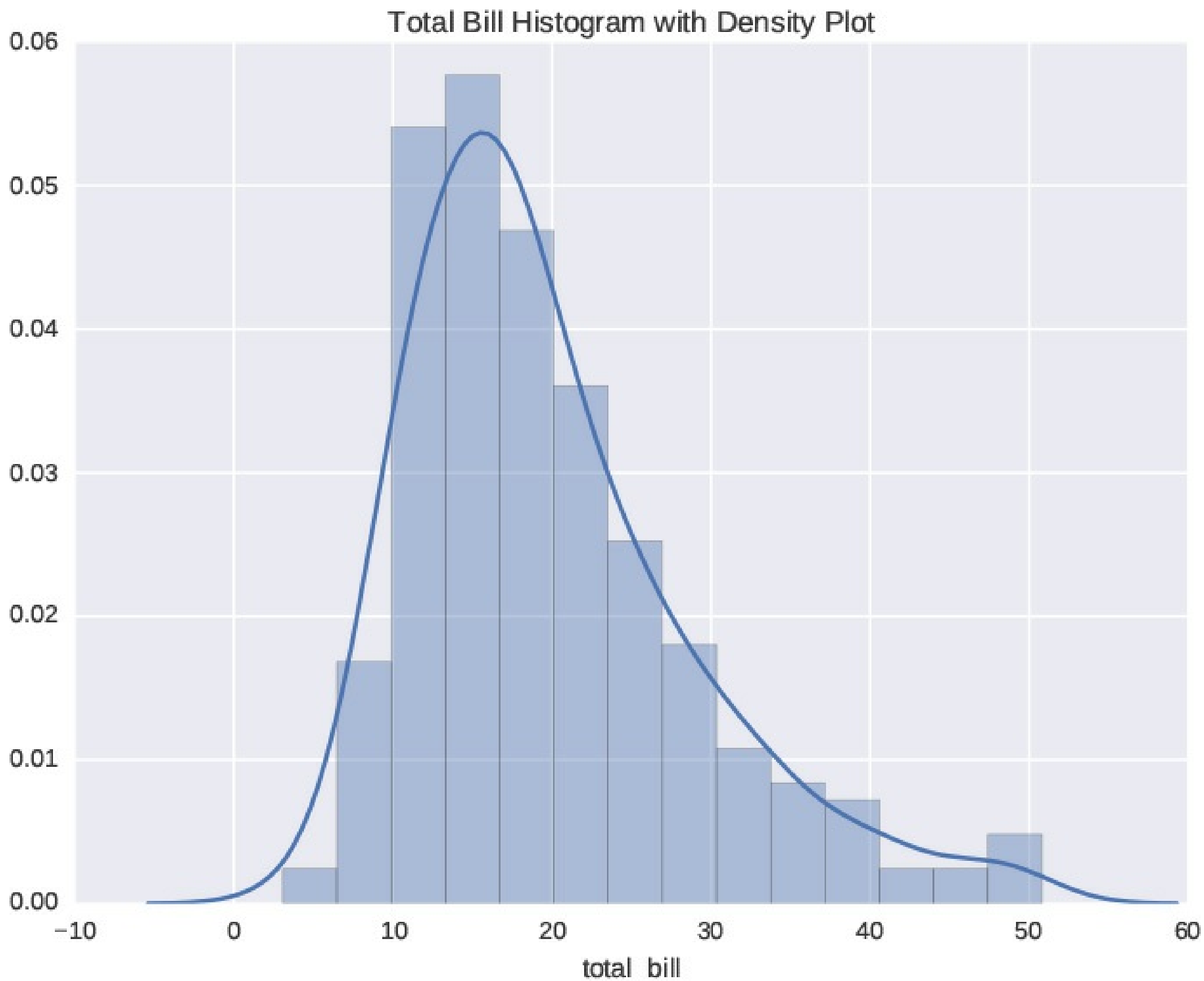
```
# load seaborn if you have not done so already
import seaborn as sns

tips = sns.load_dataset("tips" )
```

3.6.1 univariate

3.6.1.1 Histograms

Histograms are created using `sns. distplot` [5]

```
hist = sns.distplot(tips['total_bill'])
hist.set_title('Total Bill Histogram with Density Plot')
```

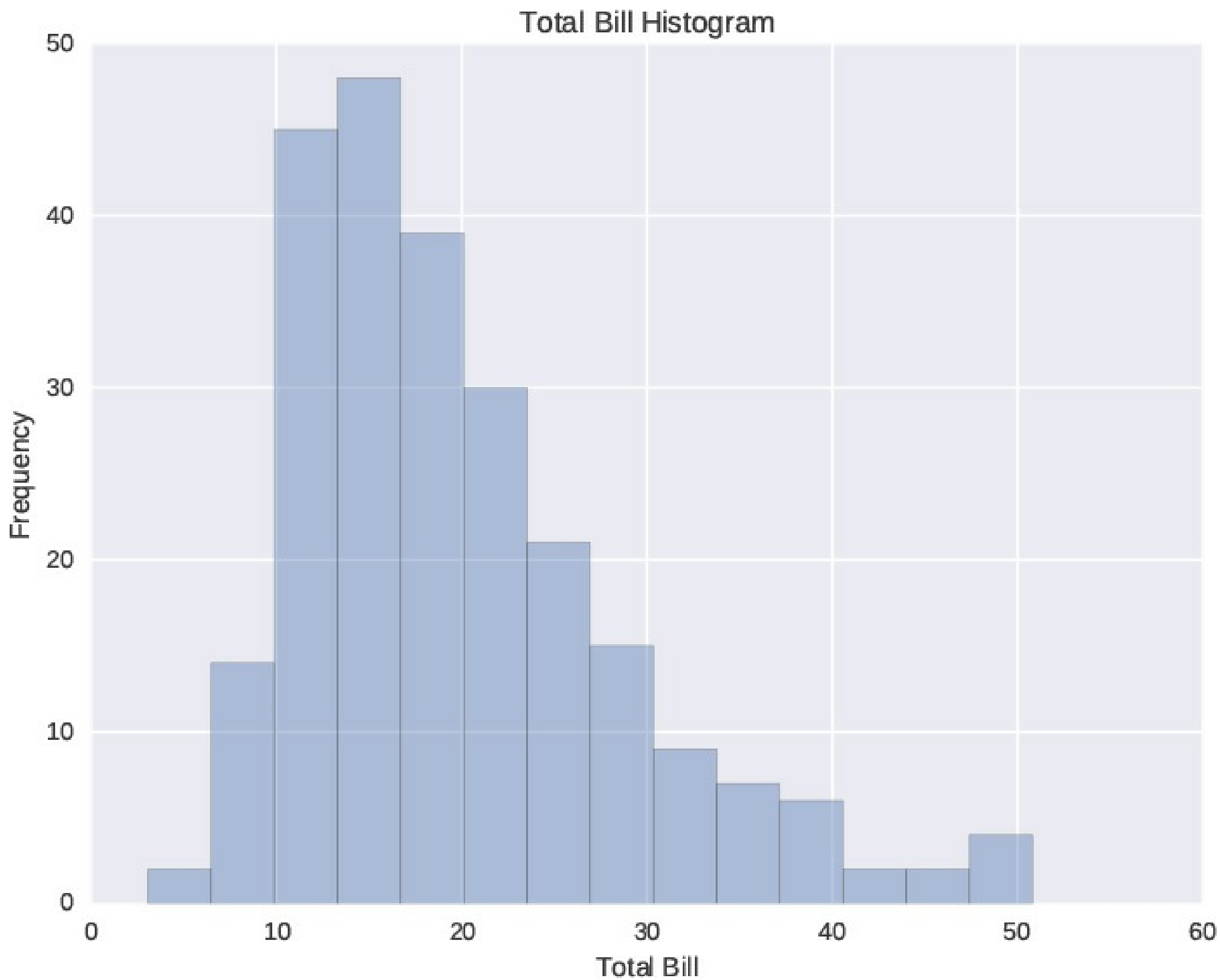


The default `distplot` will plot both a histogram and a density plot (using kernel density estimation).

If we just wanted the histogram we can set the `kde` parameter to `False`.

```
hist = sns.distplot(tips['total_bill'],   kde=False)
hist.set_title('Total Bill Histogram')
hist.set_xlabel('Total Bill')
hist.set_ylabel('Frequency')
```

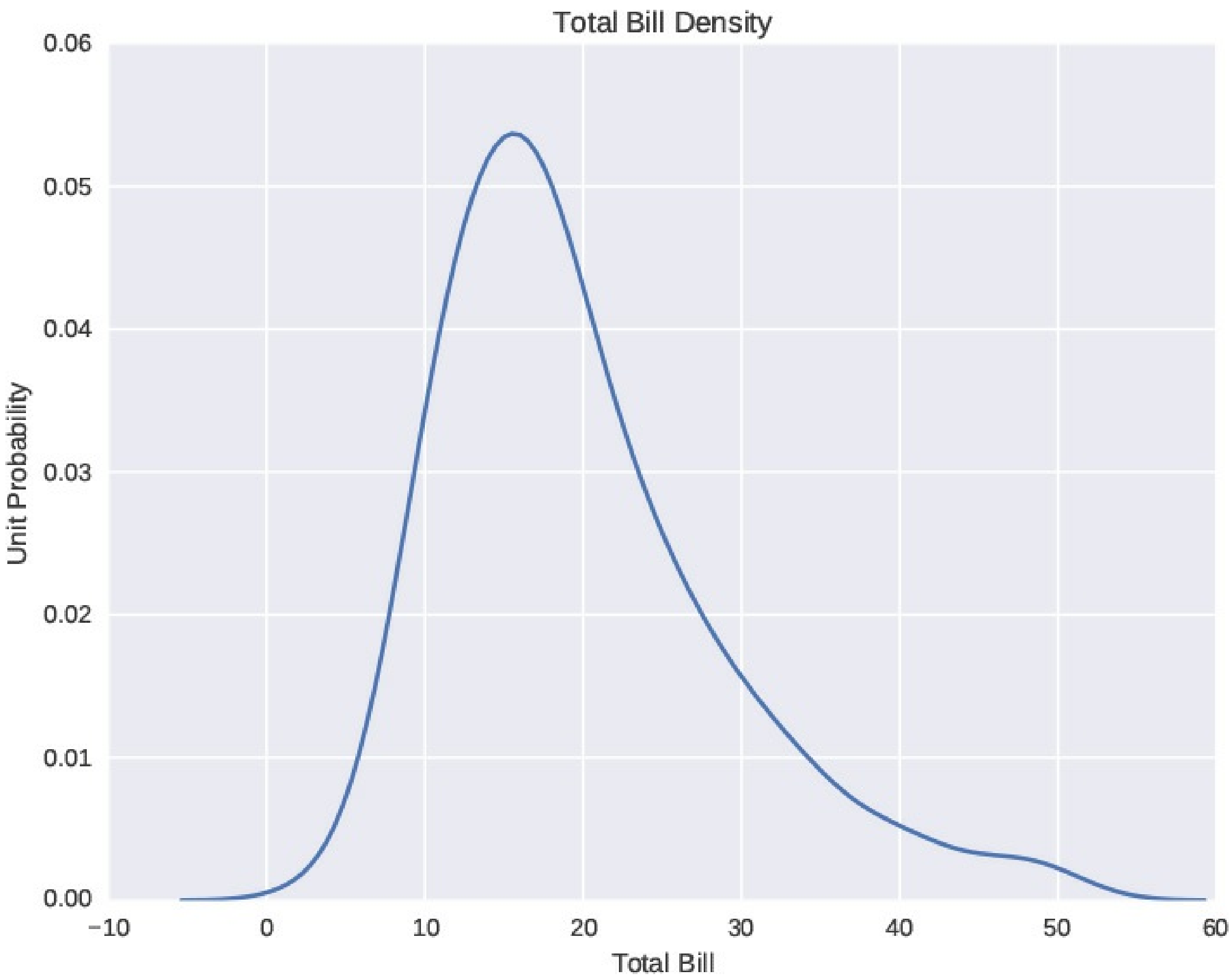
[5] https://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.distplot.html#seaborn.distplot



3.6.1.2 Density Plot (kernel Density Estimation)

Density plots are another way to visualize a univariate distribution. It essentially works by drawing a normal distribution centered at each data point, and smooths out the overlapping plots such that the under the curve is 1.

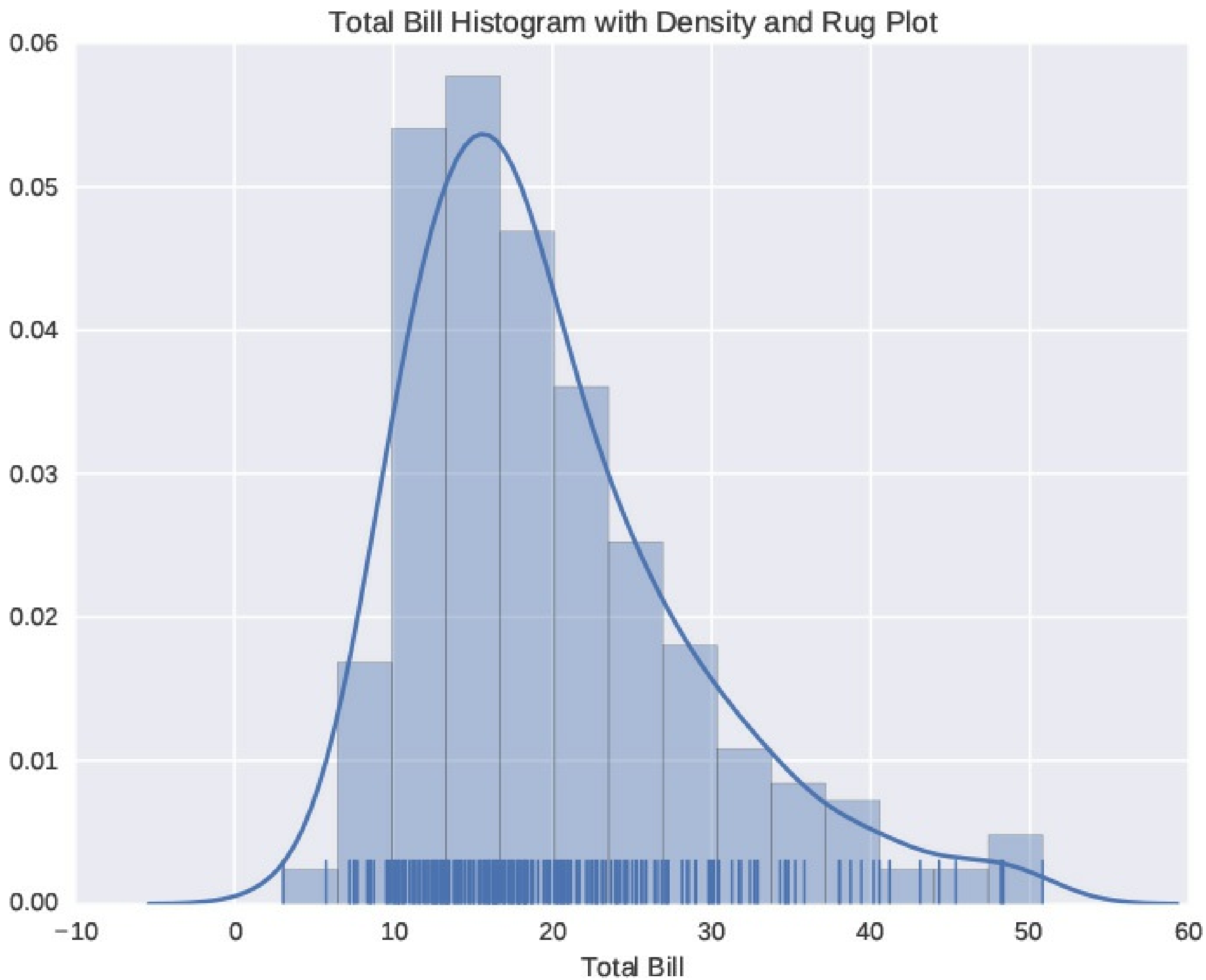
```
den = sns.distplot(tips['total_bill'] ,  hist=False)
den.set_title('Total Bill Density')
den.set_xlabel('Total Bill')
den set_ylabel('Unit Probability')
```



3.6.1.3 Rug plot

Rug plots are a 1-dimensional representation of a variable's distribution. They are typically used with other plots to enhance a visualization. This plot shows a histogram overlaid with a density plot and a rug plot on the bottom.

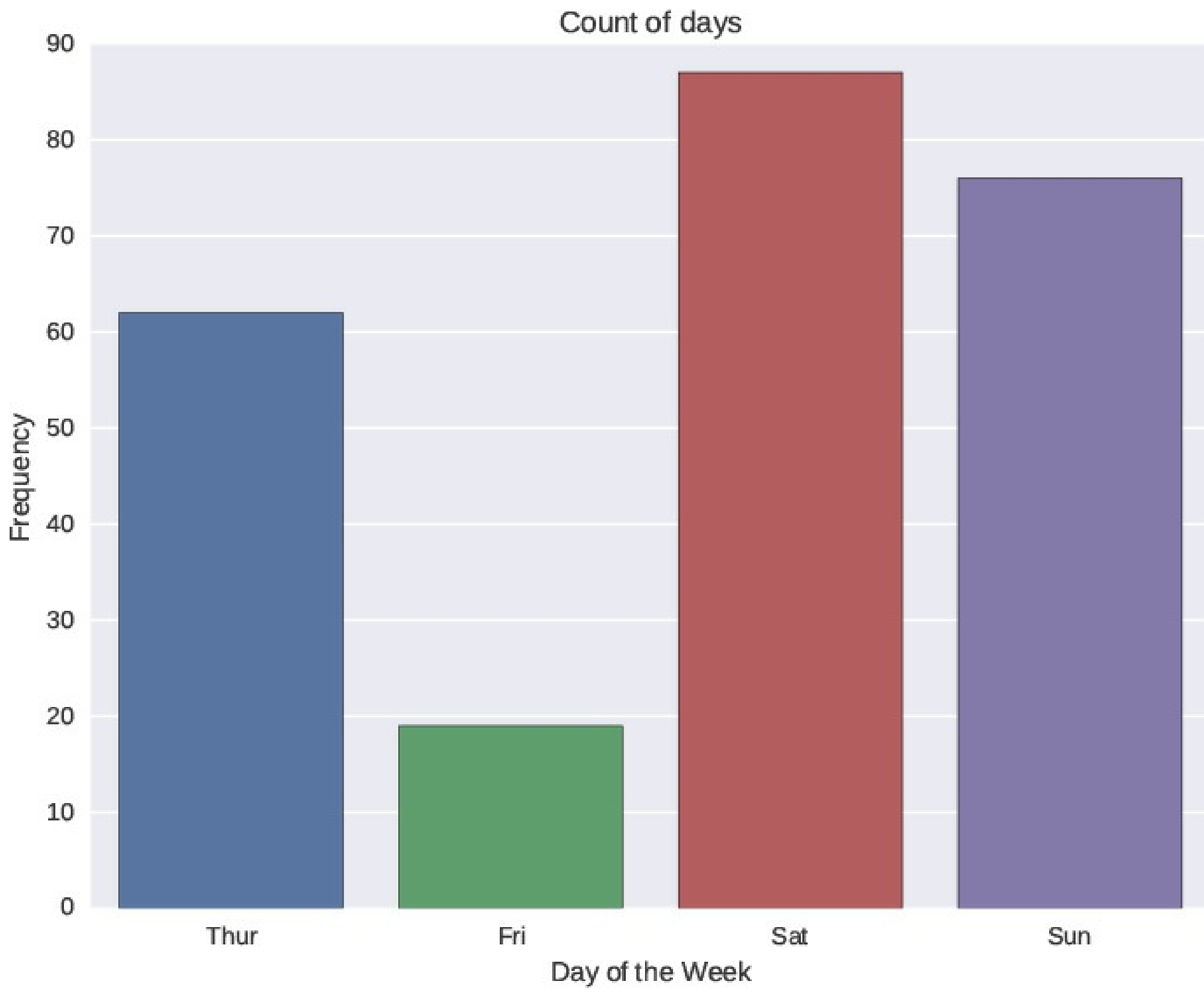
```
hist_den_rug = sns.distplot(tips['total_bill'],   rug=True)
hist_den_rug.set_title('Total Bill Histogram with Density and R
Plot')
hist_den_rug.set_xlabel('Total Bill')
```



3.6.1.4 Count plot (Bar plot)

Bar plots are very similar to histograms, but instead of binning vales to produce a distribution, bar plots can be used to count discrete variables. A `countplot` is used for this purpose.

```
count = sns.countplot('day',   data=tips)
count.set_title('Count of days')
count.set_xlabel('Day of the Week')
count.set_ylabel('Frequency')
```



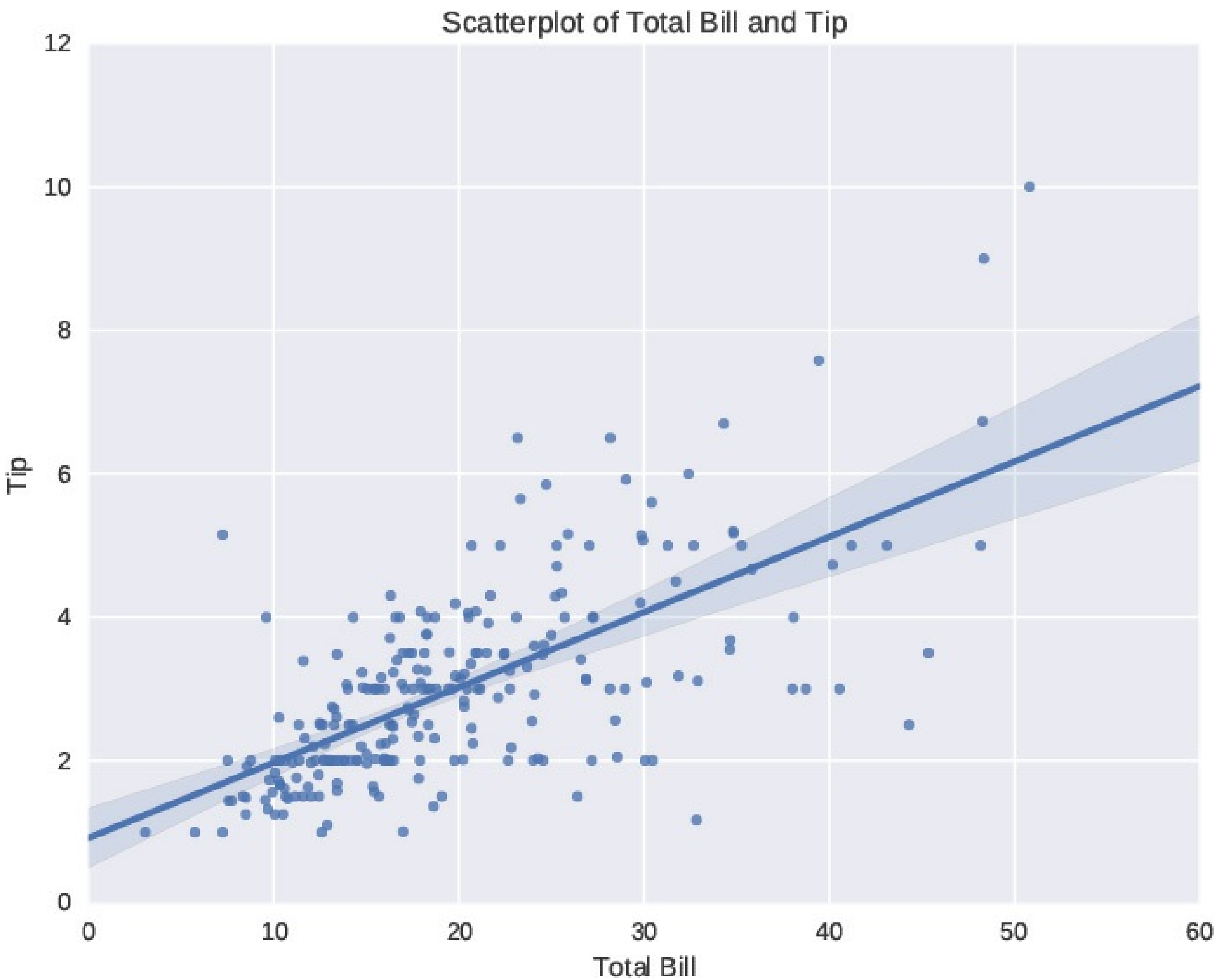
3.6.2 bivariate

3.6.2.1 Scatter plot

There are a few ways to create a scatter plot in `seaborn`. There is no explicit function named `scatter`. Instead, we use `regplot`.

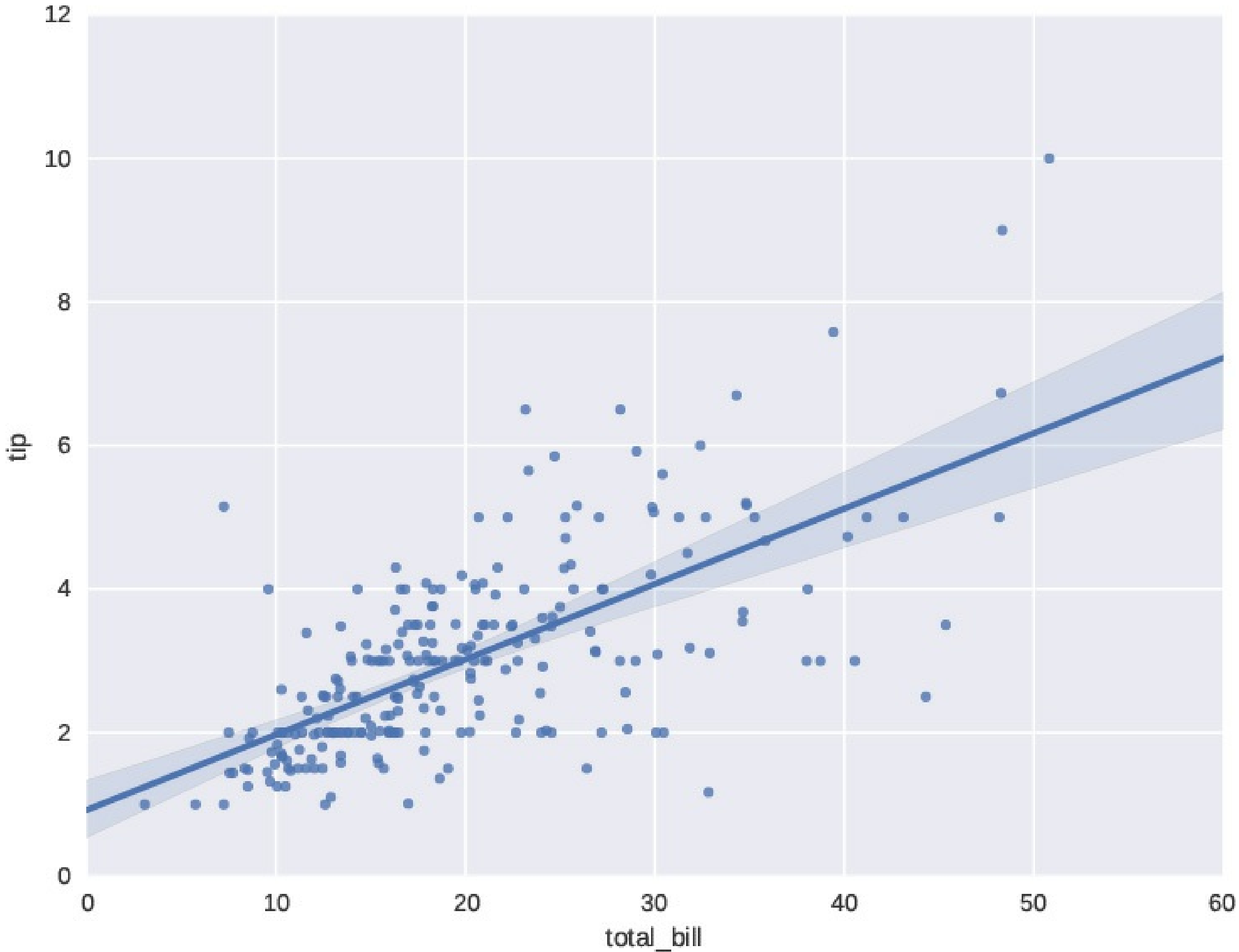
`regplot` will plot a scatter plot and also fit a regression line. We can set `fit_reg =False` so it only shows the scatter plot.

```
scatter = sns.regplot(x='total_bill',   y='tip',   data=tips)
scatter.set_title('Scatterplot of Total Bill and Tip')
scatter.set_xlabel('Total Bill')
scatter.set_ylabel('Tip')
```



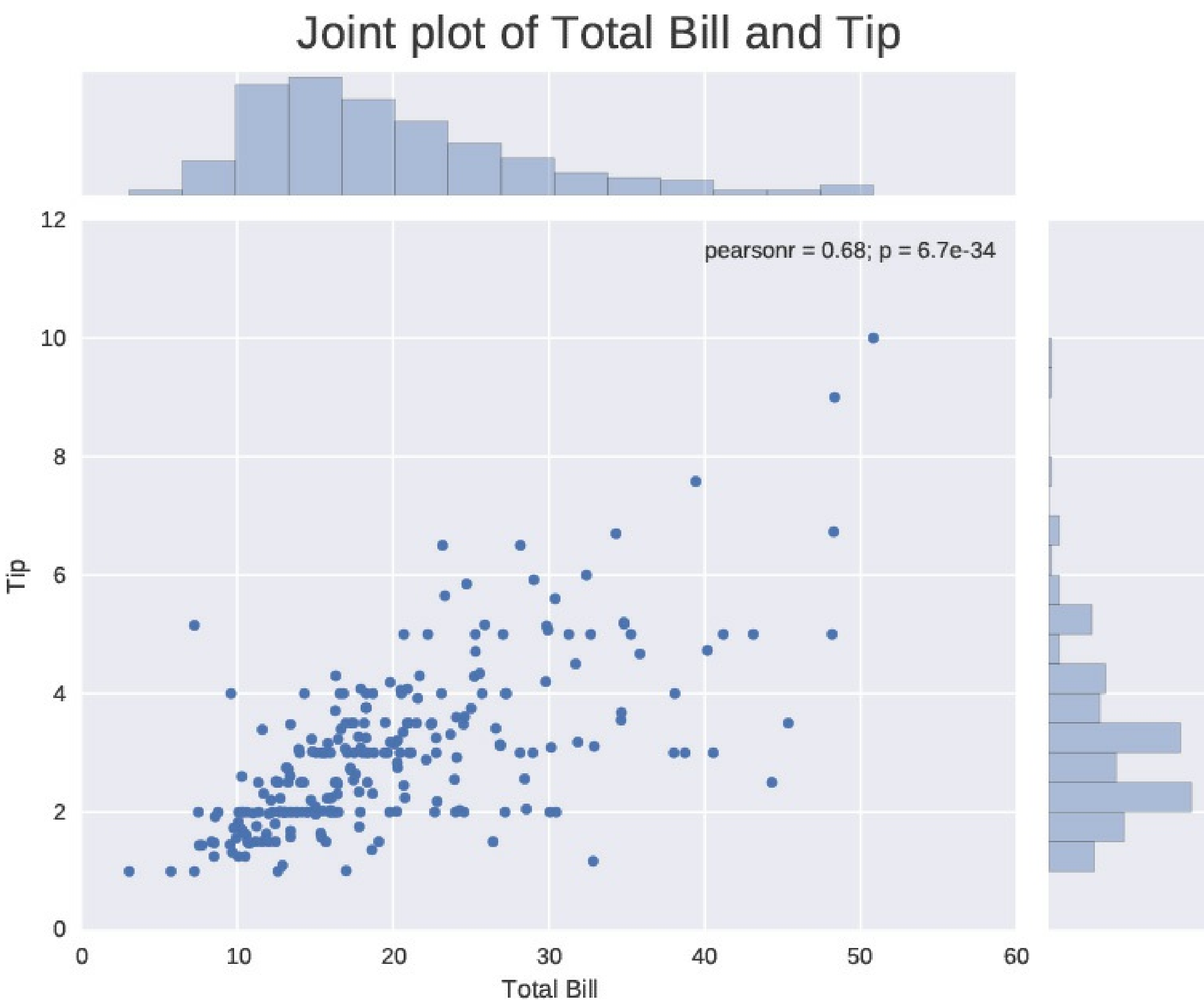
There is a similar function, `Implot`, that can also plot scatter plots. Internally, `Implot` calls `regplot`, so `regplot` is a more general plot function. The main difference is that `regplot` creates an axes (See figure 3-2) and `Implot` creates a figure.

```
sns Implot(x='total_bill',   y='tip',   data=tips)
```



We can also plot our scatter plot with a univariate plot on each axis using `jointplot`.

```
scatter = sns.jointplot(x='total_bill',   y='tip',  data=tips)
scatter.set_axis_labels(xlabel='Total Bill',   ylabel='Tip' )
# add a title, set font size,  and move the text above the tota
axes
scatter.fig.suptitle('Joint plot of Total Bill and Tip',
                     fontsize=20,   y=1.03)
```

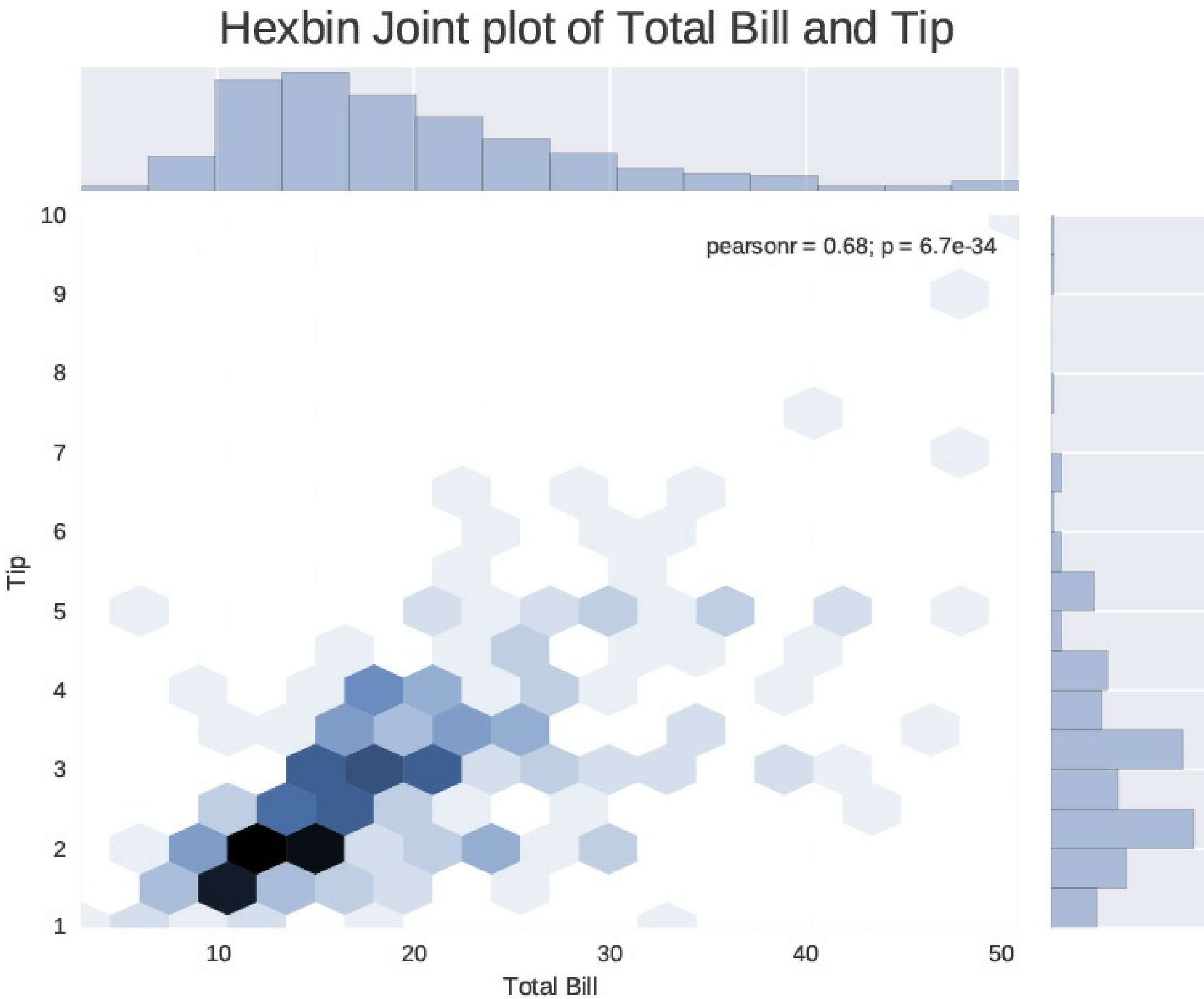


3.6.2.2 Hexbin plot

Scatter plots are great for comparing two variables. However, sometimes there are too many points for a scatter plot to be meaningful. One way to get around this is to bin points on the plot together. Just like how histograms can bin a variable to create a bar, `hexbin` can bin two variables. A hexagon is used because it is the most efficient shape to cover an arbitrary 2D surface.

This is an example of `seaborn` building on top of `matplotlib` as `hexbin` is a `matplotlib` function.

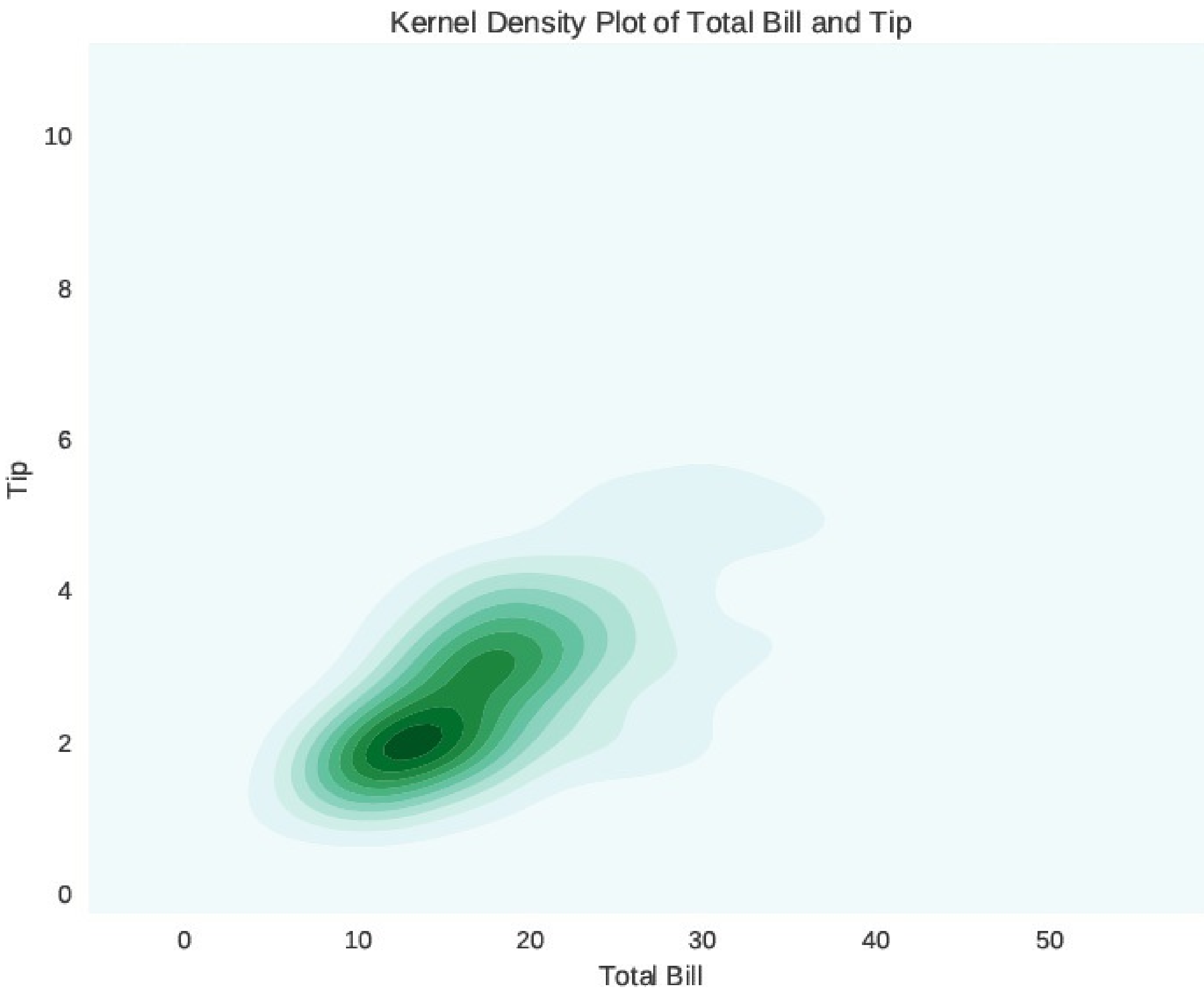
```
hex = sns.jointplot(x="total_bill",  y="tip",   data=tips,   k:
hex.set_axis_labels(xlabel='Total Bill',   ylabel='Tip')
hex.fig.suptitle('Hexbin Joint plot of Total Bill and Tip',
                      fontsize=20,   y=1.03)
```



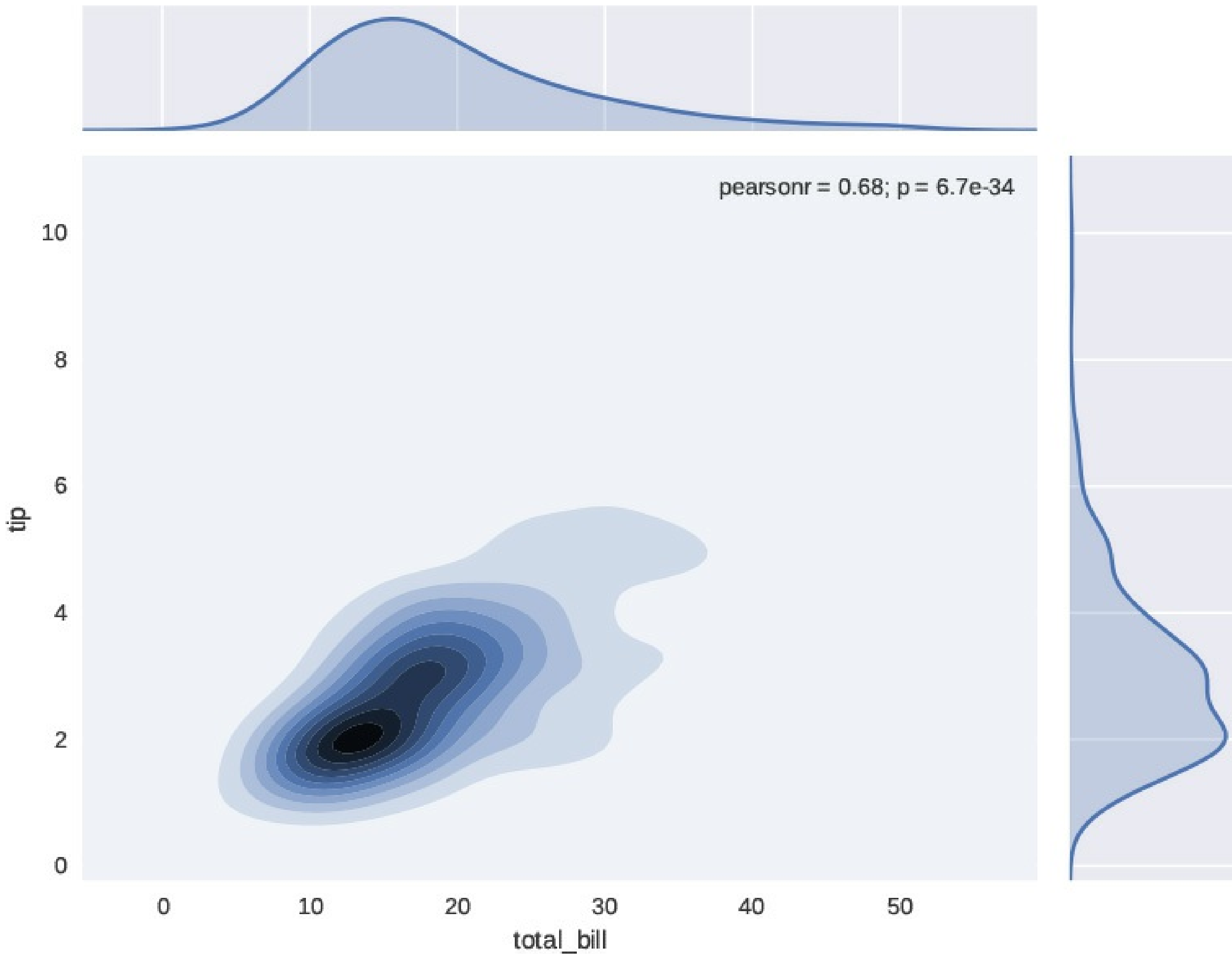
3.6.2.3 2D Density plot

You can also have a 2D kernel density plot. It is similar to how `sns.kdeplot` works, except it can plot a density plot across 2 variables.

```
kde = sns.kdeplot(data tips['total_bill'],
                  data2=tips['tip'],
                  shade=True)    # shade will fill in the conto
kde.set_title('Kernel Density Plot of Total Bill and Tip')
kde.set_xlabel('Total Bill')
kde.set_ylabel('Tip')
```



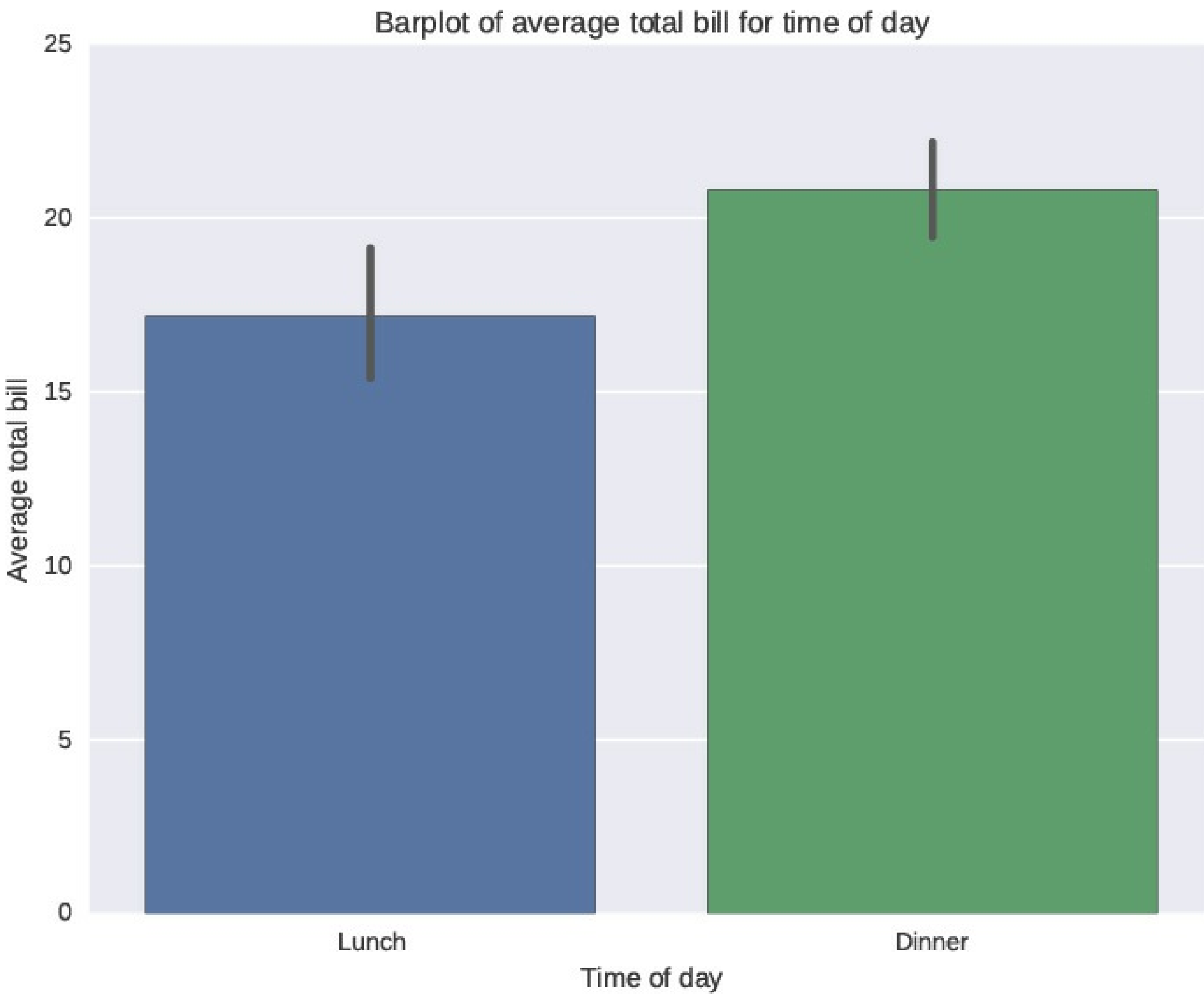

```
kde_joint = sns.jointplot(x='total_bill',   y='tip',
                          data=tips,
                          kind='kde')
```



3.6.2.4 Bar plot

Bar plots can also be used to show multiple variables. By default, `barplot` will calculate a mean, but you can pass any function into the `estimator` parameter, for example, the `numpy.std` function to calculate the standard deviation.

```
bar = sns.barplot(x='time',   y=' total_bill' ,   data=tips)
bar.set_title('Barplot of average total bill for time of day')
bar.set_xlabel('Time of day')
bar.set_ylabel('Average total bill')
```

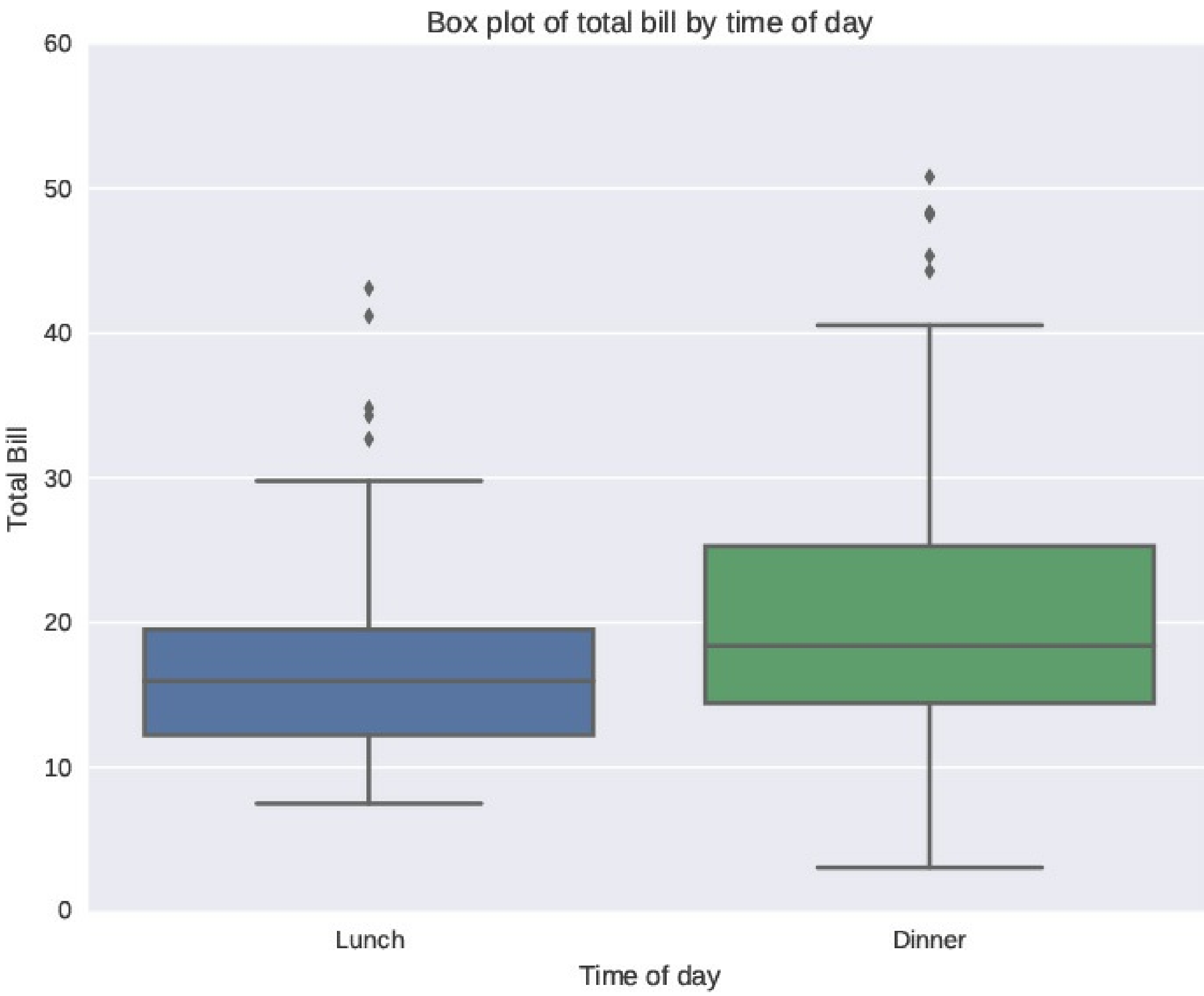


3.6.2.5 Box plot

Unlike previous plots, a box plot shows multiple statistics: the minimum, first quartile, median, third quartile, maximum, and if applicable, outliers based on the interquartile range.

The y parameter is optional, meaning, if it is left out, it will create a single box in the plot.

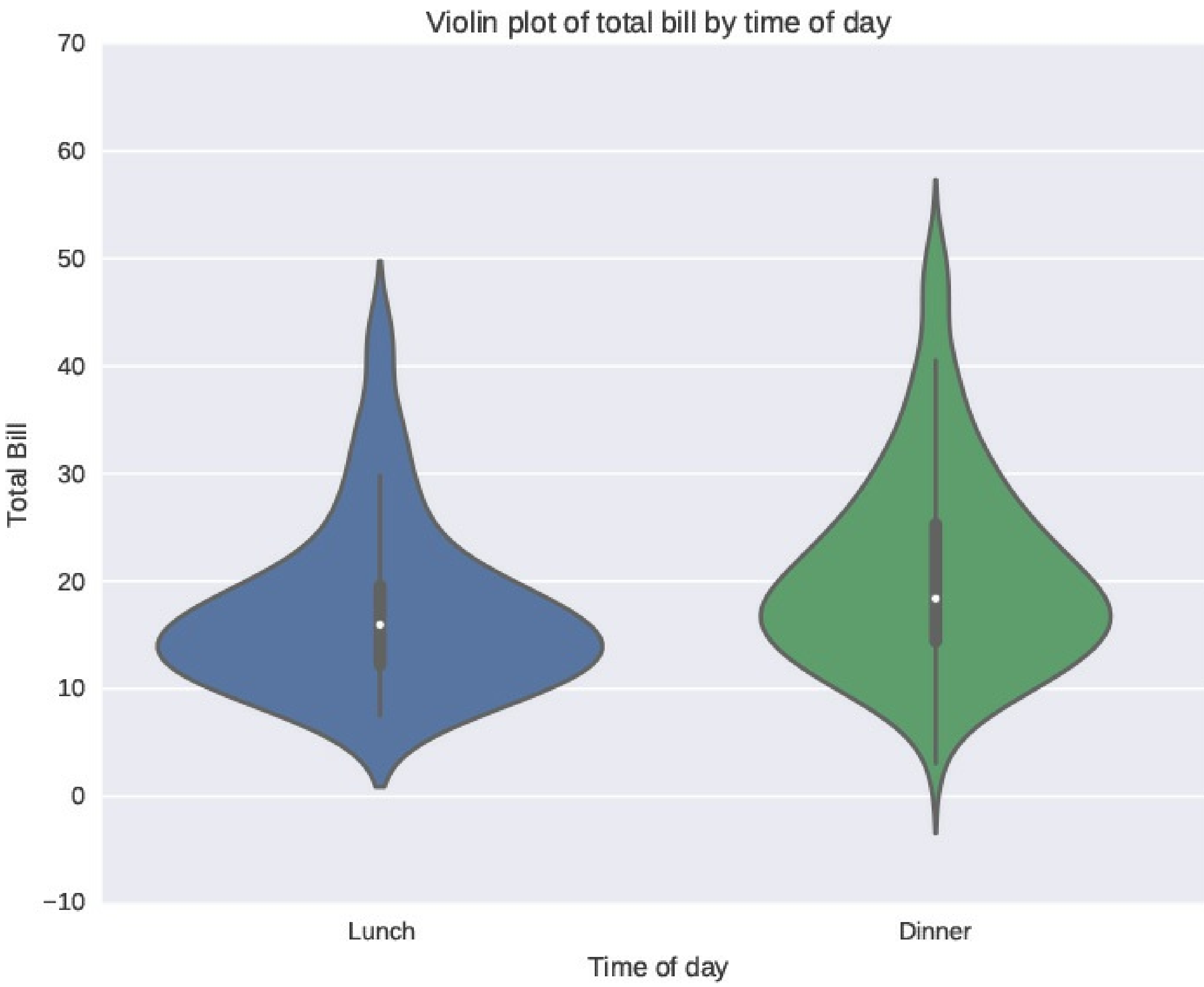
```
box = sns.boxplot(x='time',   y='total_bill',   data=tips)
box.set_title('Box plot of total bill by time of day')
box set_xlabel('Time of day')
box.set_ylabel('Total Bill')
```



3.6.2.6 Violin plot

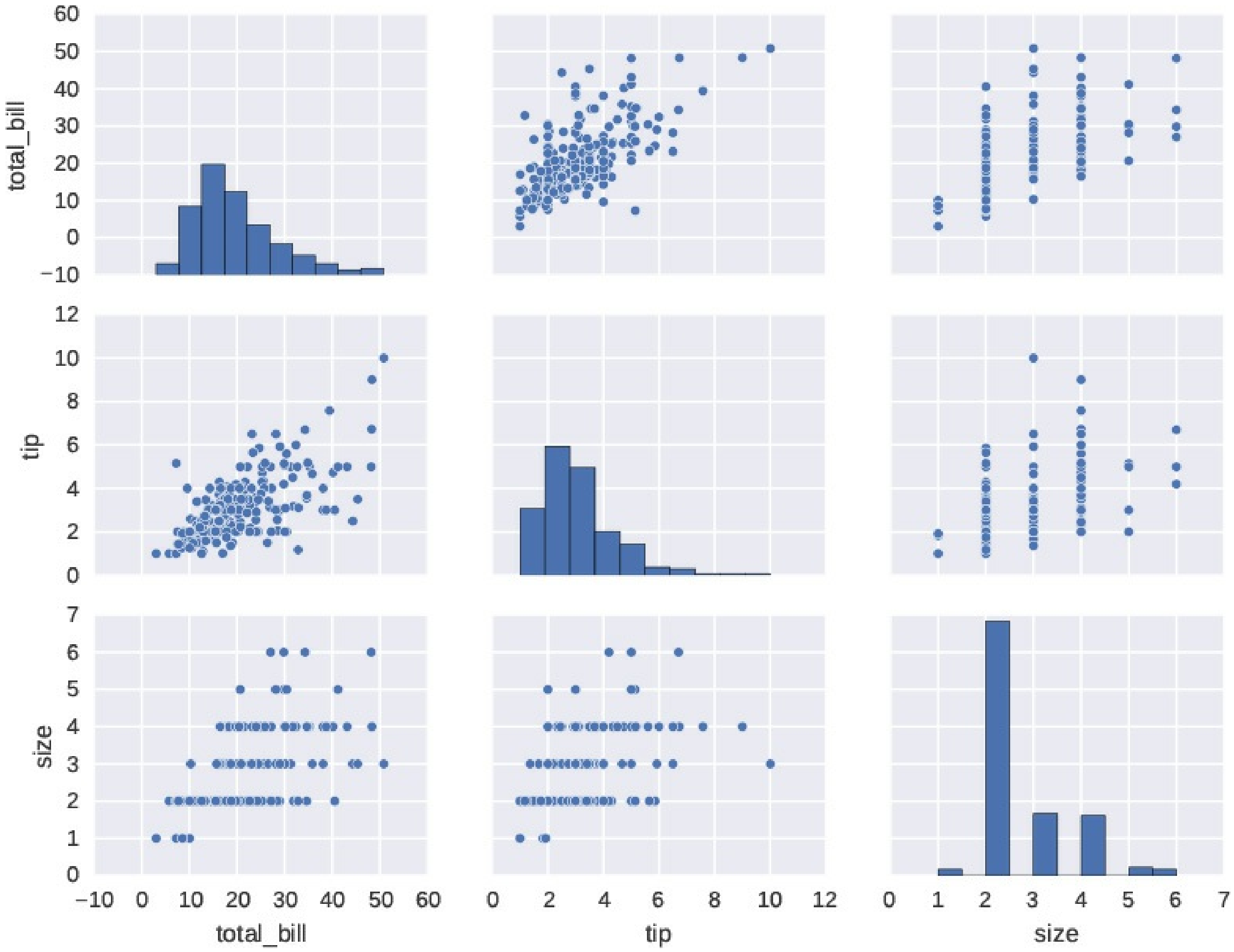
Box plots are a classical statistical visualization. However, they can obscure the underlying distribution of the data. Violin plots are able to show the same values as the box plot, but plots the "boxes" as a kernel density estimation. This can help retain more visual information about your data since only plotting summary statistics can be misleading, as seen by the Anscombe's quartets.

```
violin = sns.violinplot(x='time',   y='total_bill',   data=tips
violin.set_title('Violin plot of total bill by time of day')
violin.set_xlabel('Time of day')
violin.set_ylabel('Total Bill')
```



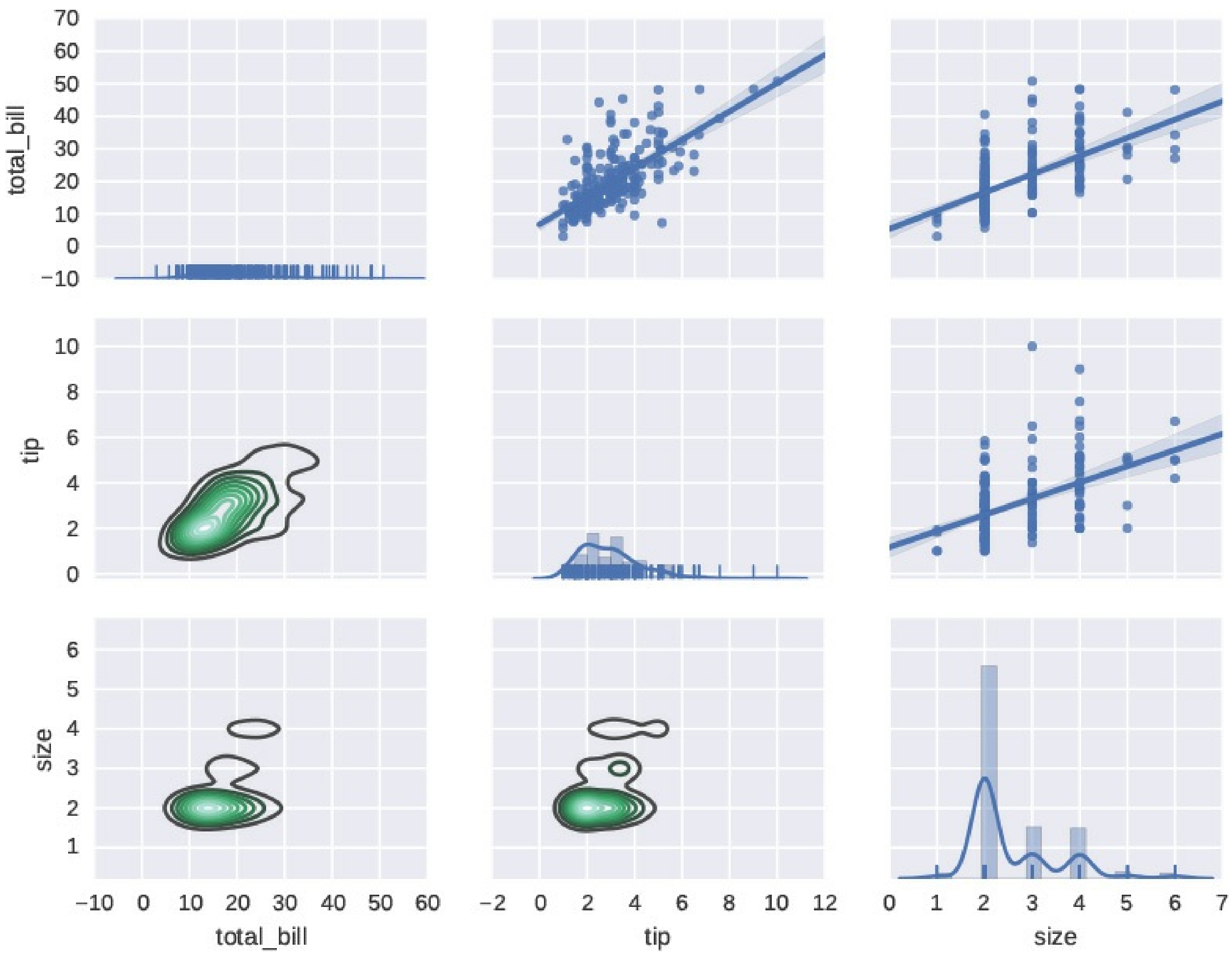
3.6.2.7 Pairwise relationships

When you have mostly numeric data, visualizing all the pairwise relationships can be easily performed using `pairplot`. This will plot a scatter plot between each pair of variables, and a histogram for the univariate.



One thing about `pairplot` is that there is redundant information. The top half of the the visualization is the same as the bottom half. We can use `pairgrid` to manually assign the plots for the top half and bottom half.

```
pair_grid = sns.PairGrid(tips)
# can also use pit.scatter instead of sns.regplot
pair_grid = pair_grid.map_upper(sns.regplot)
pair_grid = pair_grid.map_lower(sns.kdeplot)
pair_grid = pair_grid.map_diag(sns.distplot,   rug=True)
```



3.6.3 multivariate

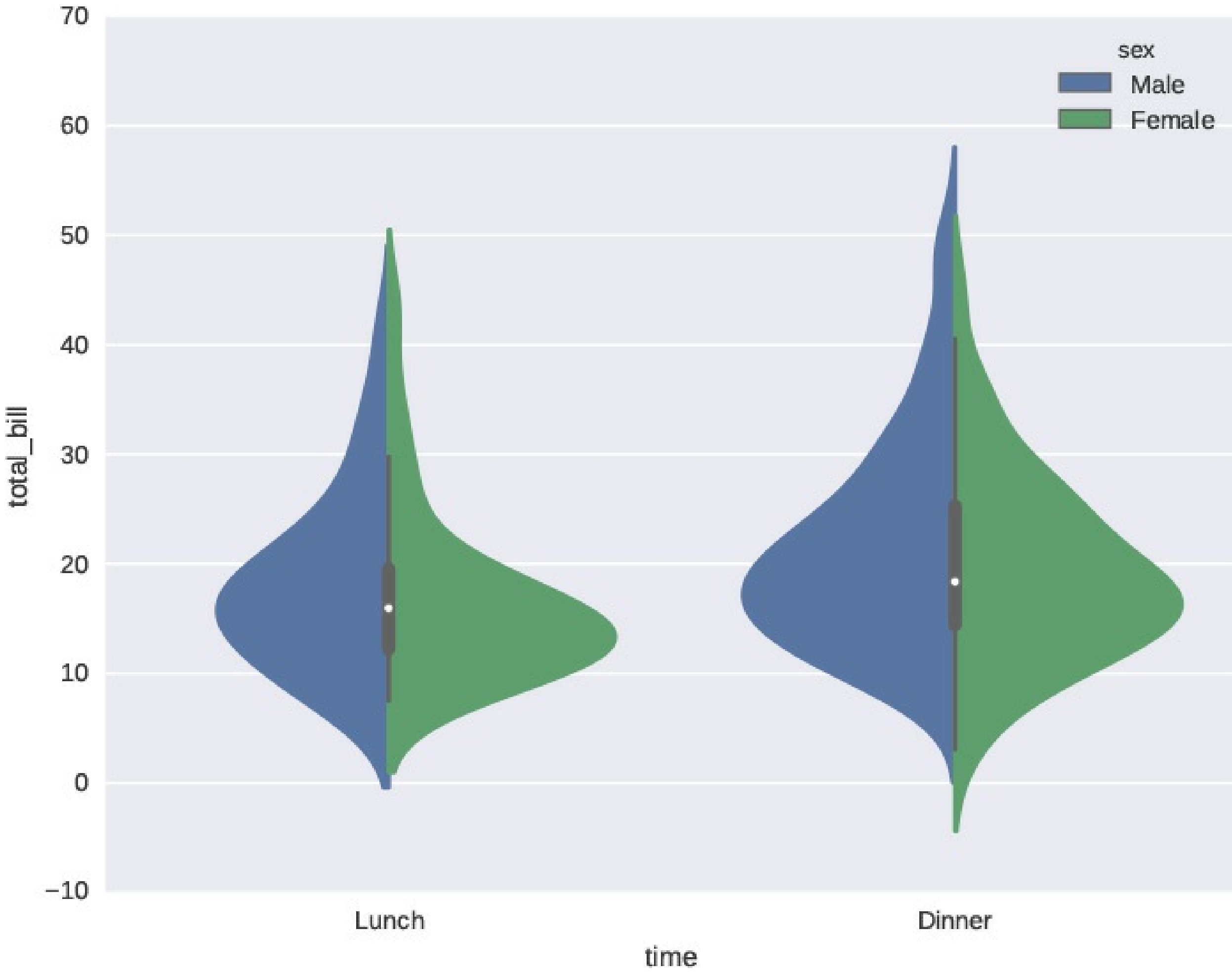
I mentioned in Section 3.5.3, that there is no de facto template for plotting multivariate data.

Possible ways to include more information is to use color, size, and shape to add more information to a plot

3.6.3.1 Colors

In a `violinplot` , we can pass the `hue` parameter to color the plot by `sex`. We can reduce the redundant information by having each half of the violins represent the different `sex`. Try the following code with and without the `split` parameter.

```
violin = sns.violinplot(x='time',   y='total_bill',
                        hue='sex',   data=tips,
                        split=True)
```

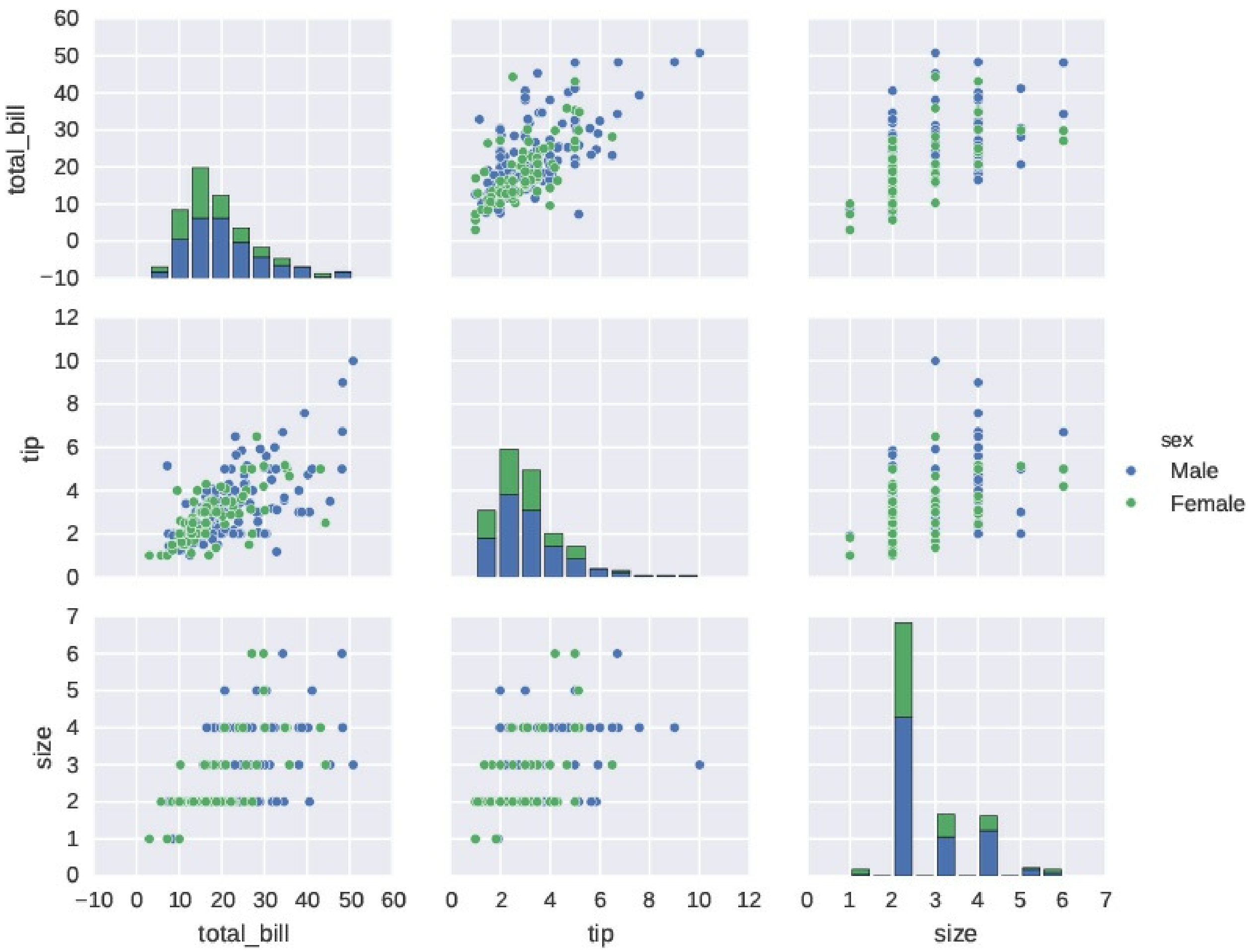


The `hue` parameter can be passed into various other plotting functions as well.

```
# note I'm using lmplot instead of regplot here
scatter = sns.lmplot(x='total_bill', y='tip', data=tips, hue='s
fit_reg=False)
```

We can make our pairwise plots a little more meaningful by passing one of the categorical variables as a `hue` parameter.

```
sns.pairplot(tips,  hue='sex')
```



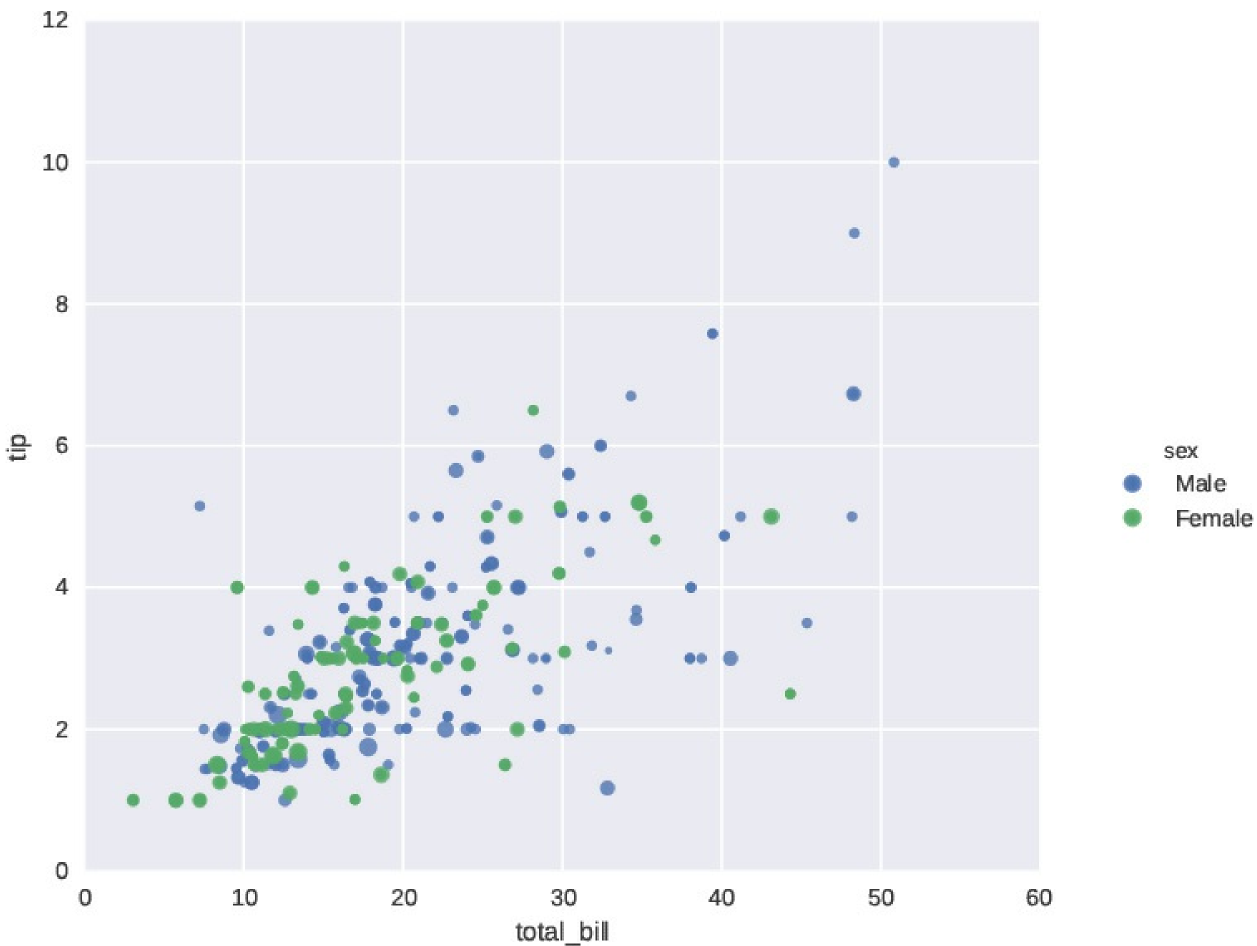
3.6.3.2 Size and Shape

Working with point sizes can also be another means to add more information to a plot. However, this should be used sparingly, since the human eye is not very good at comparing areas.

Here, is an example of how `seaborn` works with `matplotlib` function calls. If you look in the documentation for `Implot` [6], you'll see that `Implot` takes a parameter called `catter,line scatter , line_kws`. This is actually them saying there is a parameter in `Implot` called `scatter_kws` and `line_kws`. Both of these parameters take a key-value pair, a Python `diet` (dictionary) to be more exact (TODO APPENDIX PYTHON DICTONARY). Key-value pairs passed into `scatter_kws` is then passed on to the matplotlib function `pit. scatter`. This is how we would access the `s` parameter to change the size of the points like we did in section 3.5.3.

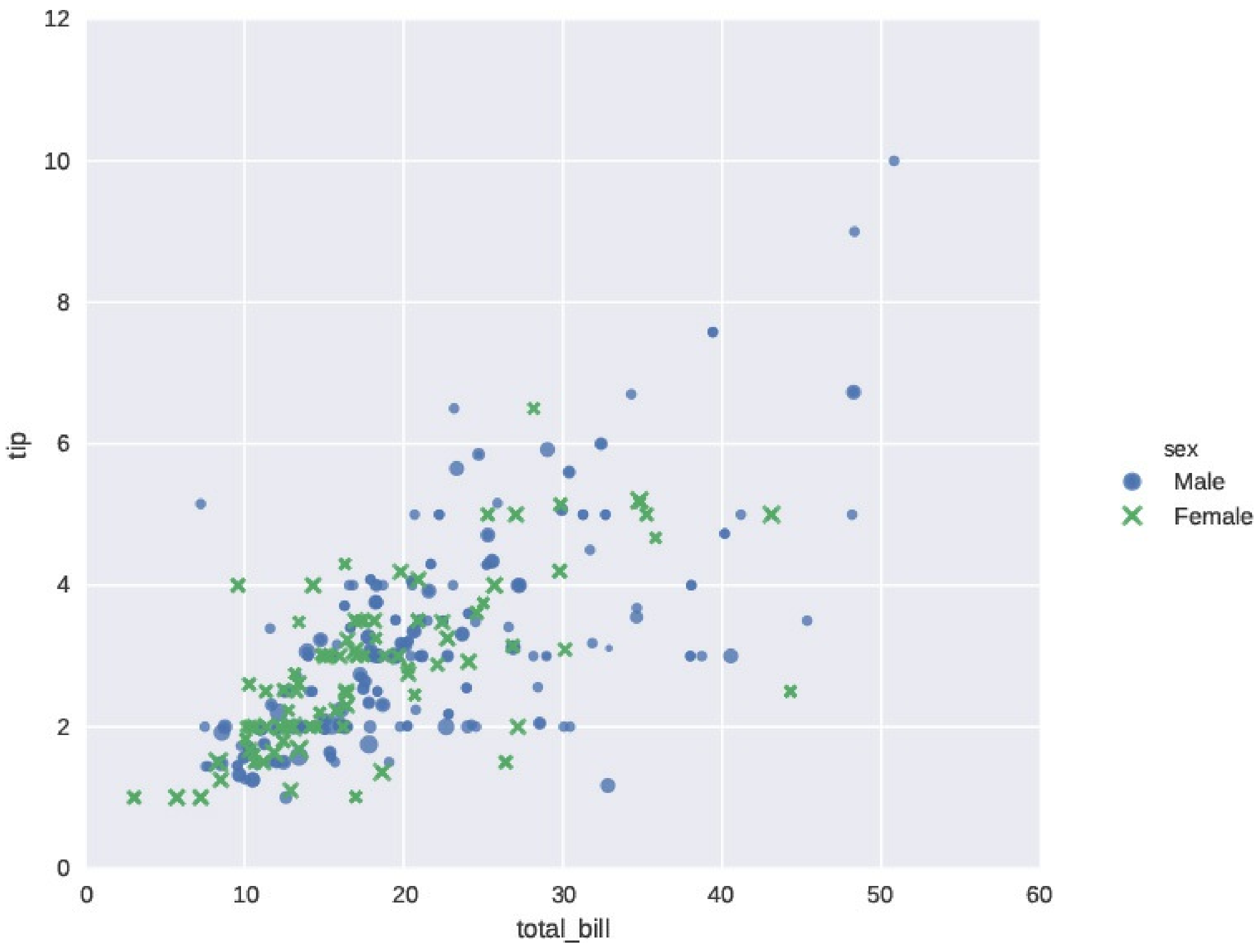
```
scatter = sns.lmplot(x='total_bill',  y='tip',   data=tips,
                     fit_reg=False,
                     hue='sex',
                     scatter_kws={'s':  tips['size']*10})
```

6 https://web.stanford.edu/~mwaskom/software/seaborn/generated/seaborn.lmplot.html



Also, when working with multiple variables, sometimes having 2 plot elements showing the same information is helpful. Here I am using color and shape to distinguish `sex`.

```
scatter = sns.lmplot(x='total_bill',   y='tip',   data=tips,
                     fit_reg=False,  hue='sex',  markers=['o',
                                 scatter_kws={'s':  tips['size'
```



3.6.3.3 facets

What if we want to show more variables? Or if we know what plot we want for our visualization, but we want to make multiple plots over a categorical variable? This is what facets are for. Instead of individually subsetting data and laying out the axes in a figure (we did this in Figure 3-1), facets in `seaborn` handle this for you.

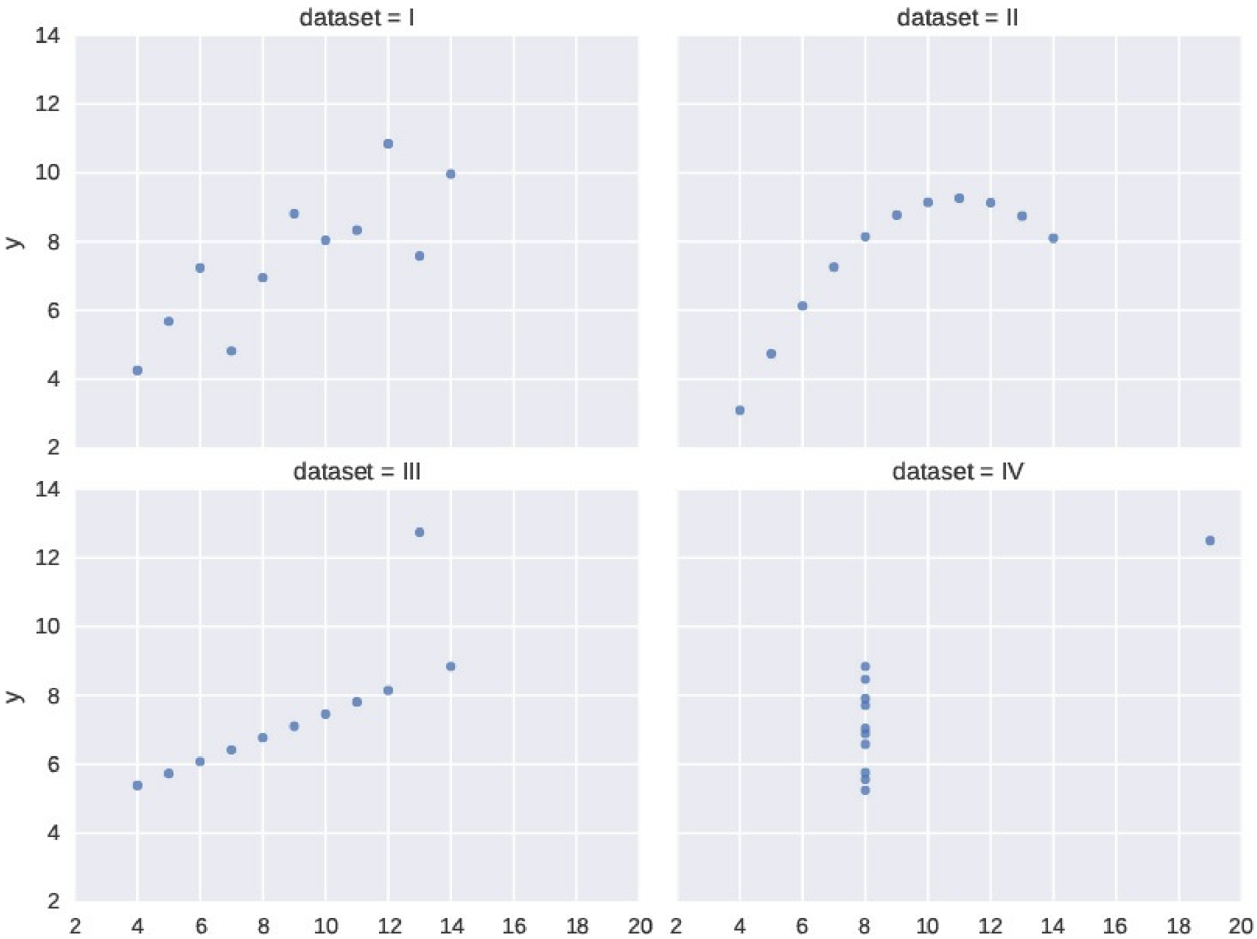
In order to use facets your data needs to be what Hadley Wickham[7] calls "Tidy Data"[8], where each row represents an observation in your data, and each column is a variable (it is also known as "long data").

To recreate our Anscombe's quartet figure from Figure 3-1 in `seaborn`:

```
anscombe = sns.lmplot(x='x',   y='y',   data anscombe,   fit_re
                      col='dataset',   col_wrap=2)
```

[7] http://hadley.nz/

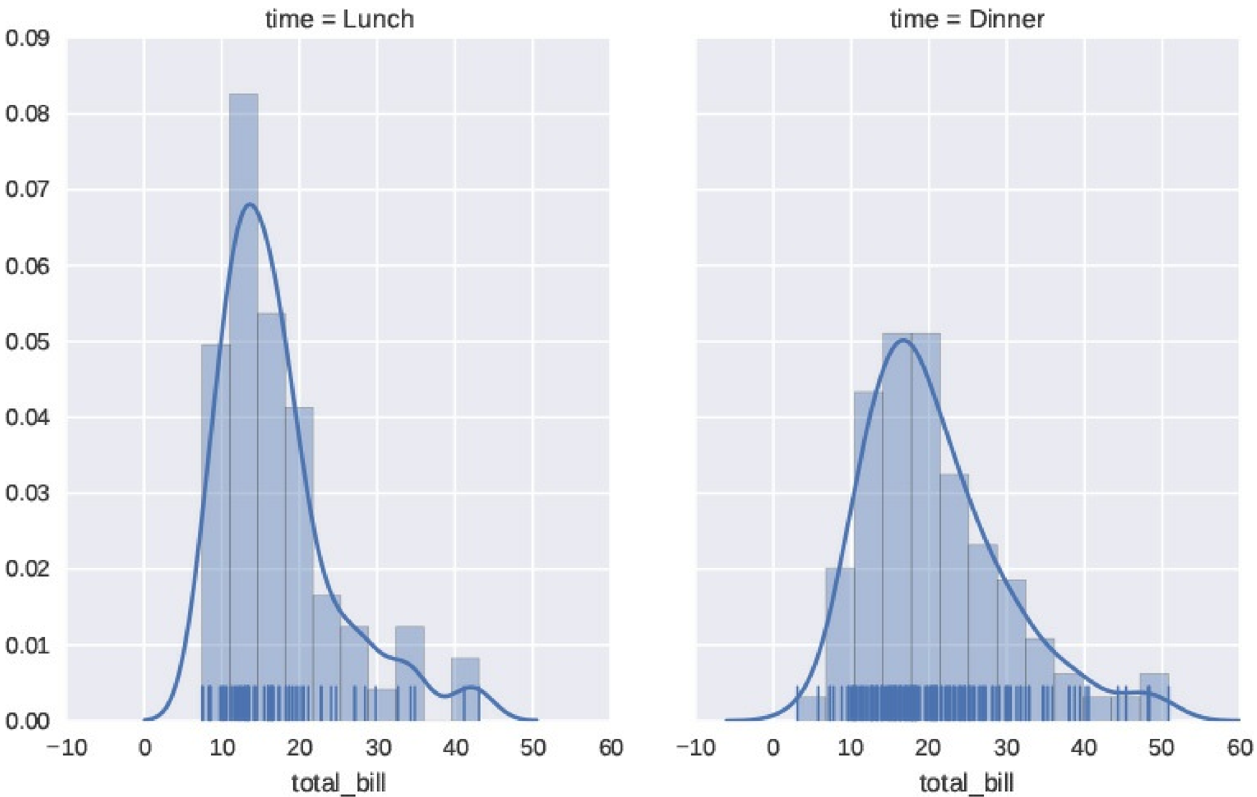
[8] http://vita.had.co.nz/papers/tidy-data.pdf



All we needed to do is pass 2 more parameters into the scatter plot function in `seaborn`. The `col` parameter is the variable the plot will facet by, and the `coLwrap` creates a figure that has 2 columns. If we do not use the `coLwrap` parameter, all 4 plots will be plotted in the same row.

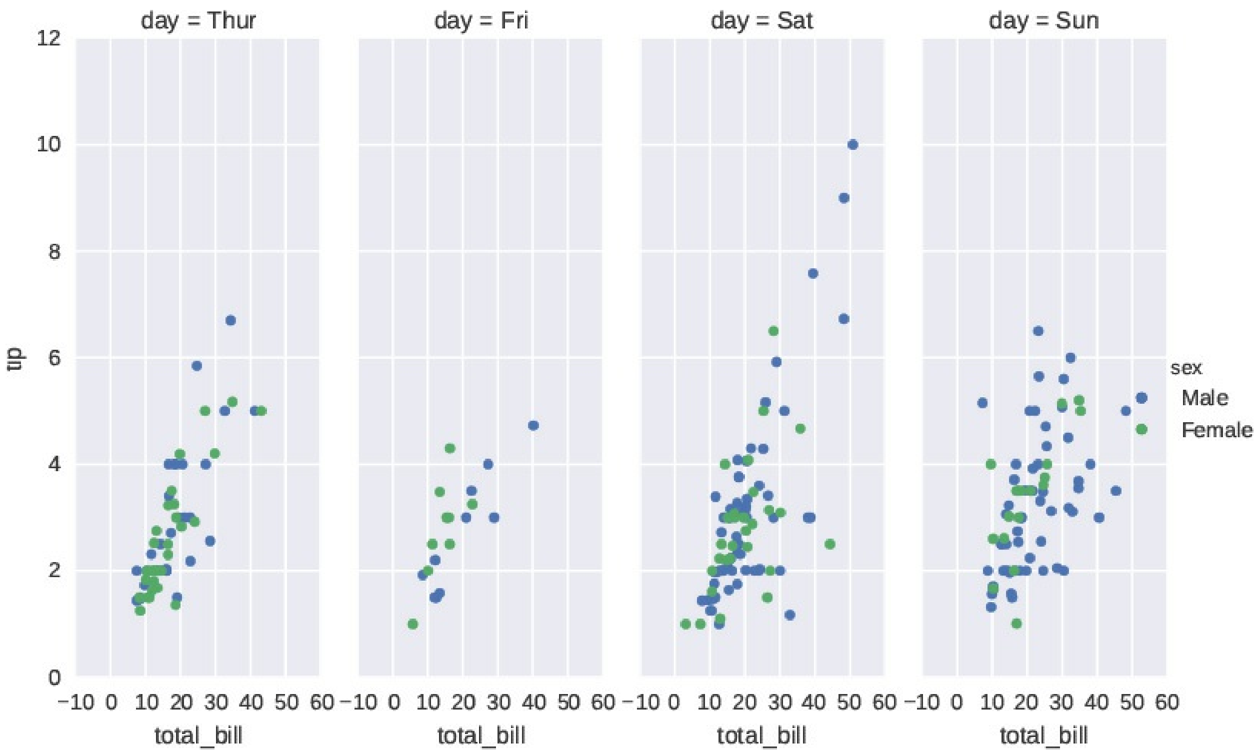
Section 3.6.2.1 discussed the differences between `Implot` and `regplot`. `Implot` is a figure level function. Many of the plots we created in `seaborn` are `axes` level functions. What this means is not every plotting function will have a `col` and `coLwrap` parameter for faceting. Instead we have to create a `FacetGrid` that knows what variable to facet on, and then supply the individual plot code for each facet.

```
# create the FacetGrid
facet = sns.FacetGrid(tips,   col='time')
# for each value in time, plot a histogram of total bill
facet.map(sns.distplot,   'total_bill',   rug=True)
```



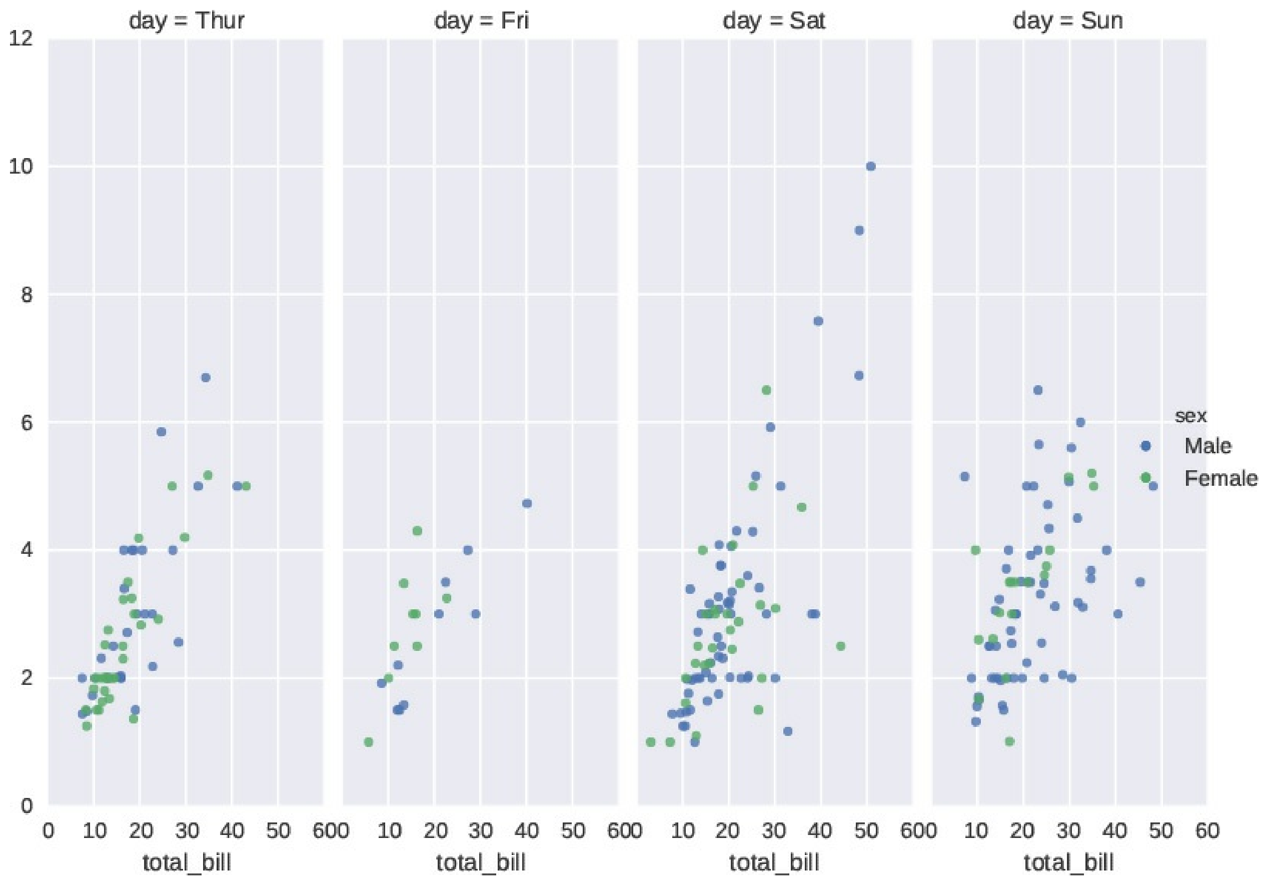
The individual facets need no be univariate plots.

```
facet = sns.FacetGrid(tips,   col = 'day',  hue='sex')
facet = facet.map(pit.scatter,   'total_bill',   'tip')
facet = facet.add_legend()
```



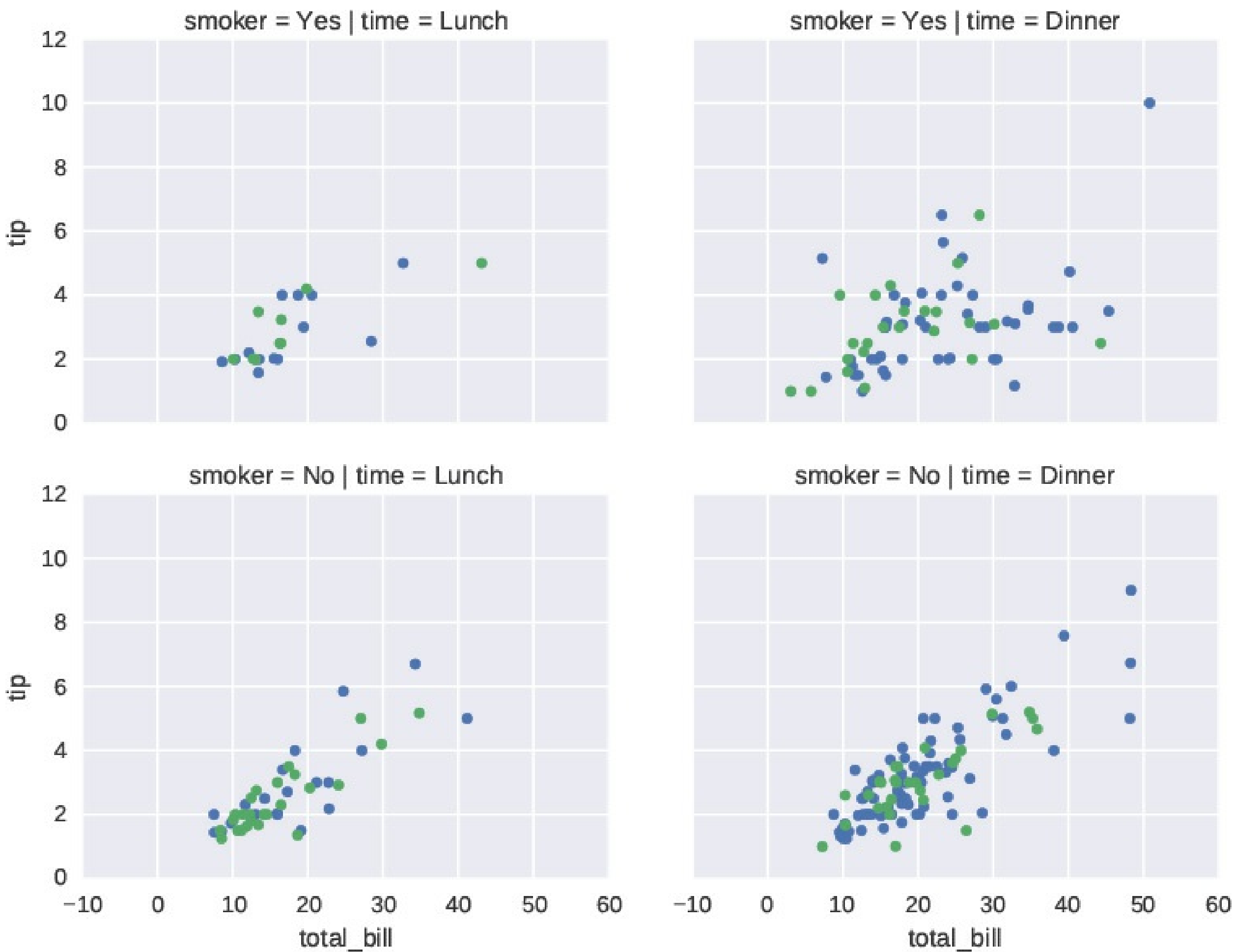
If you wanted to stay in `seaborn` you can do the same plot using `lmplot`

```
sns.lmplot(x='total_bill',   y='tip',  data=tips,   fit_reg=Fal
                hue='sex',    col='day')
```



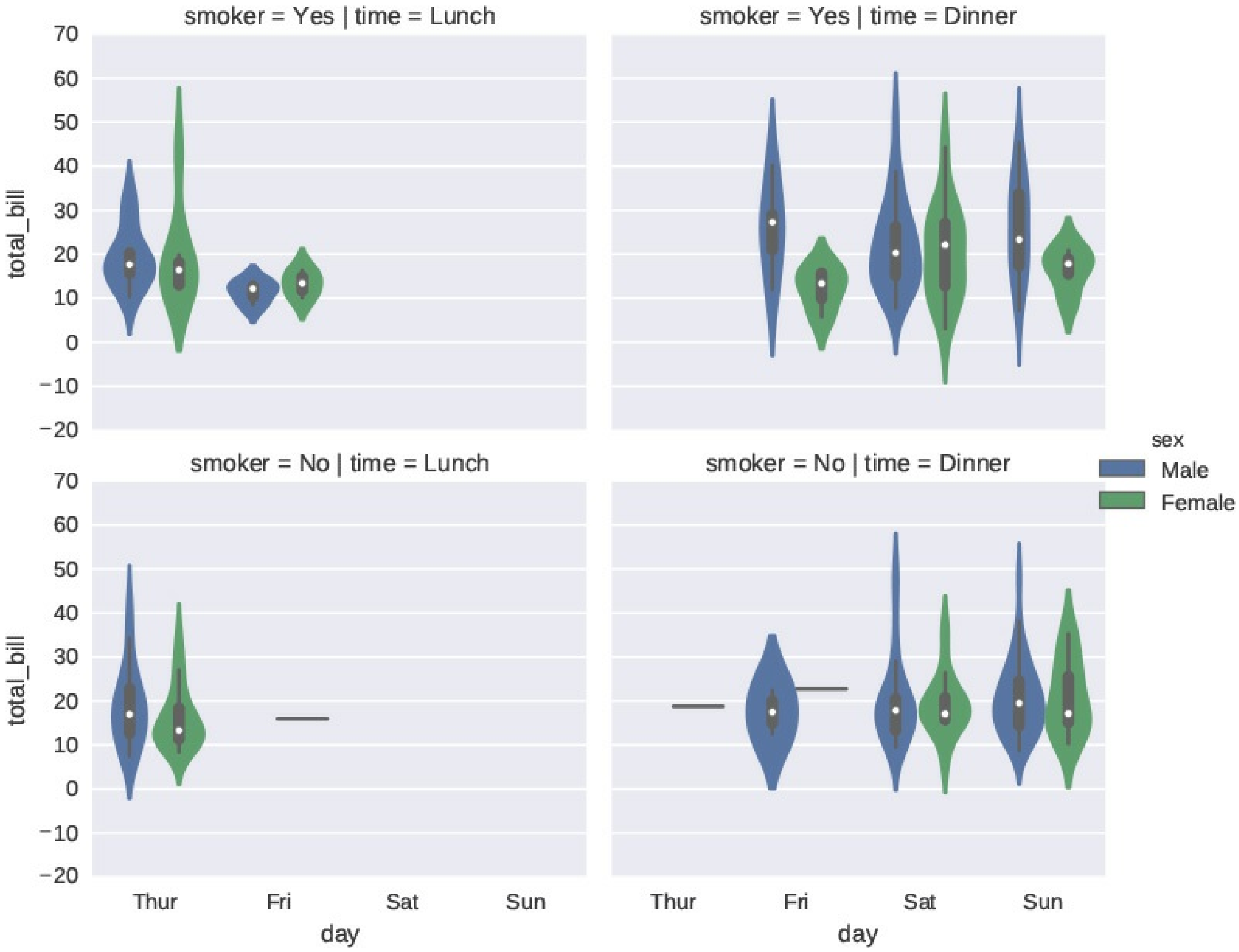
The last thing you can do with facets is to have one variable be faceted on the x axis, and another variable faceted on the y axis. We accomplish this by passing a `row` parameter.

```
facet = sns.FacetGrid(tips, col='time', row='smoker', hue='sex'
facet.map(pit.scatter,   'total_bill',   'tip')
```



If you do not want all the `hue` elements overlapping eather other (i.e., you want this behaviour in scatter plots, but not violin plots), you can use the `sns.factorplot` function.

```
sns.factorplot(x='day', y='total_bill', hue='sex', data=tip
               row='smoker', col='time', kind='violin')
```



3.7 pandas

`pandas` objects also come equipped with their own plotting functions. Just like `seaborn`, the plotting functions built into `pandas` are just wrappers around `matplotlib` with presets.

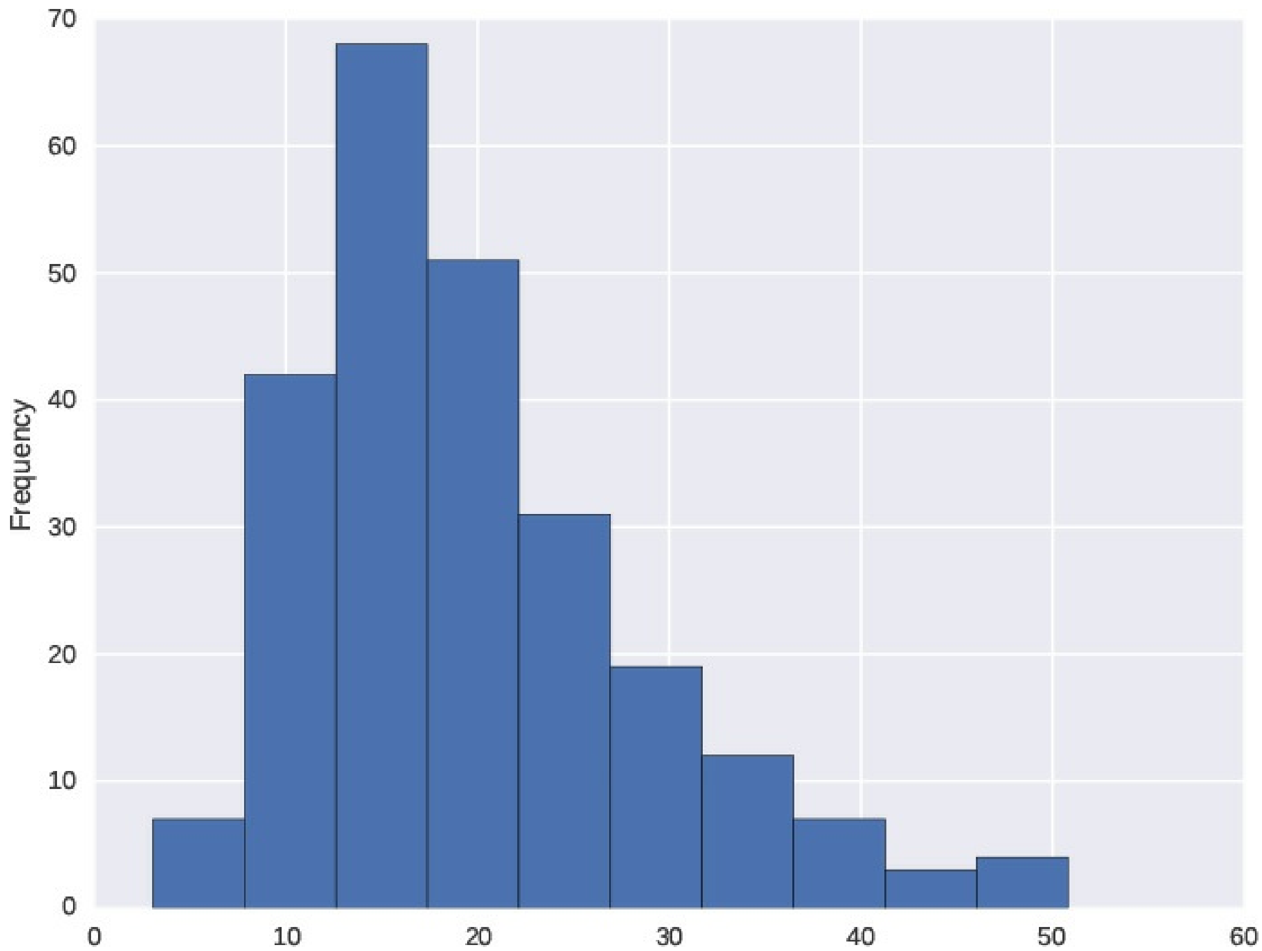
In general, plotting using `pandas` follows the `DataFrame.plot.PLOT_TYPE` or `Series . plot. PLOT_TYPE` functions.

3.7.1 Histograms

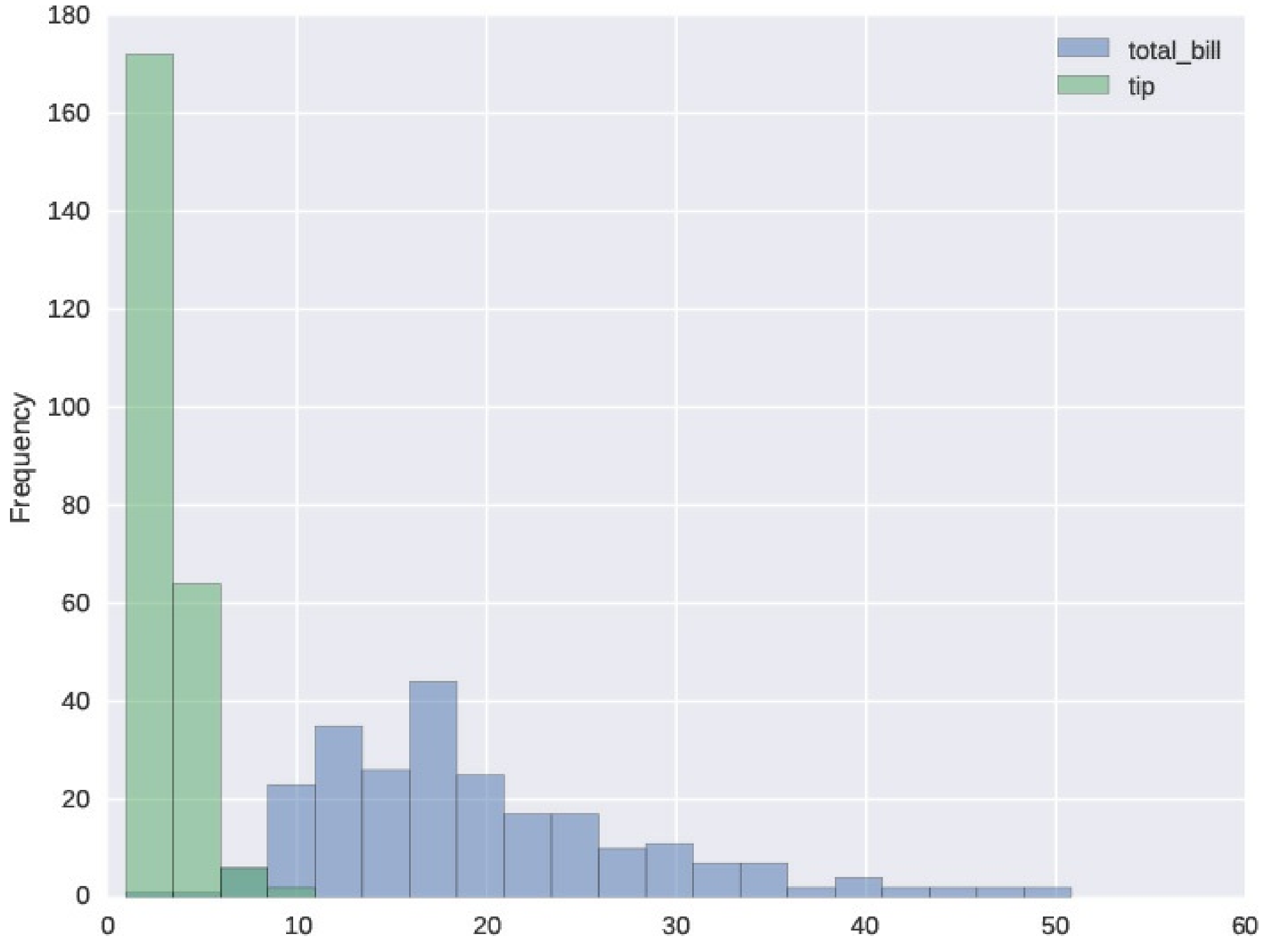
Histograms can be created using the `DataFrame. plot, hist` or `Series . plot, hist` function.

```
# on a series
```

```
tips['total_bill'].plot.hist()
```



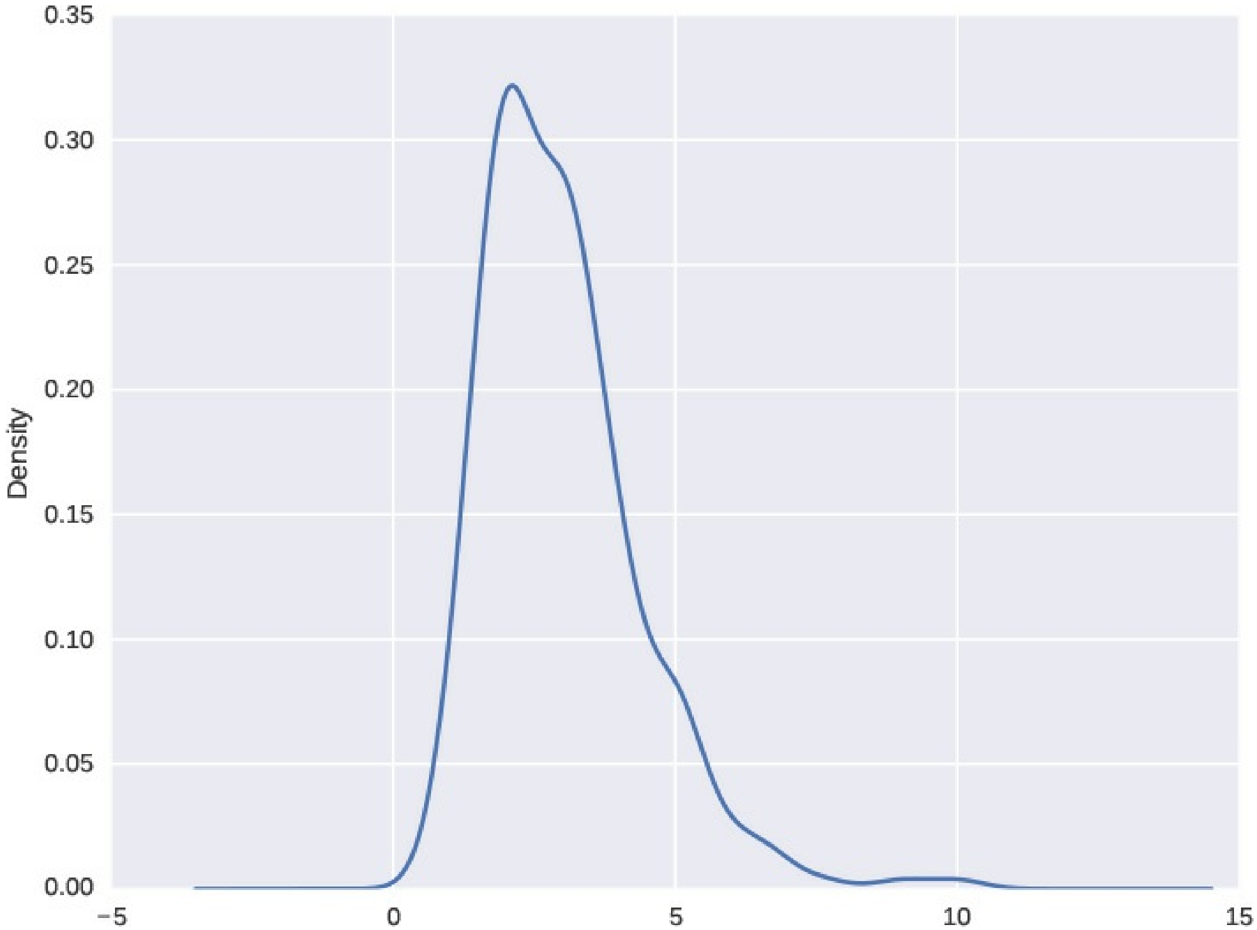

```
# on a data frame
# set an alpha channel  transparency
# so we can see though the overlapping bars
tips[['total_bill',   'tip']].plot.hist(alpha=0.5,  bins=20)
```



3.7.2 Density Plot

The kernel density estimation (density) plot can be created with the `Data Frame, plot, kde` function.

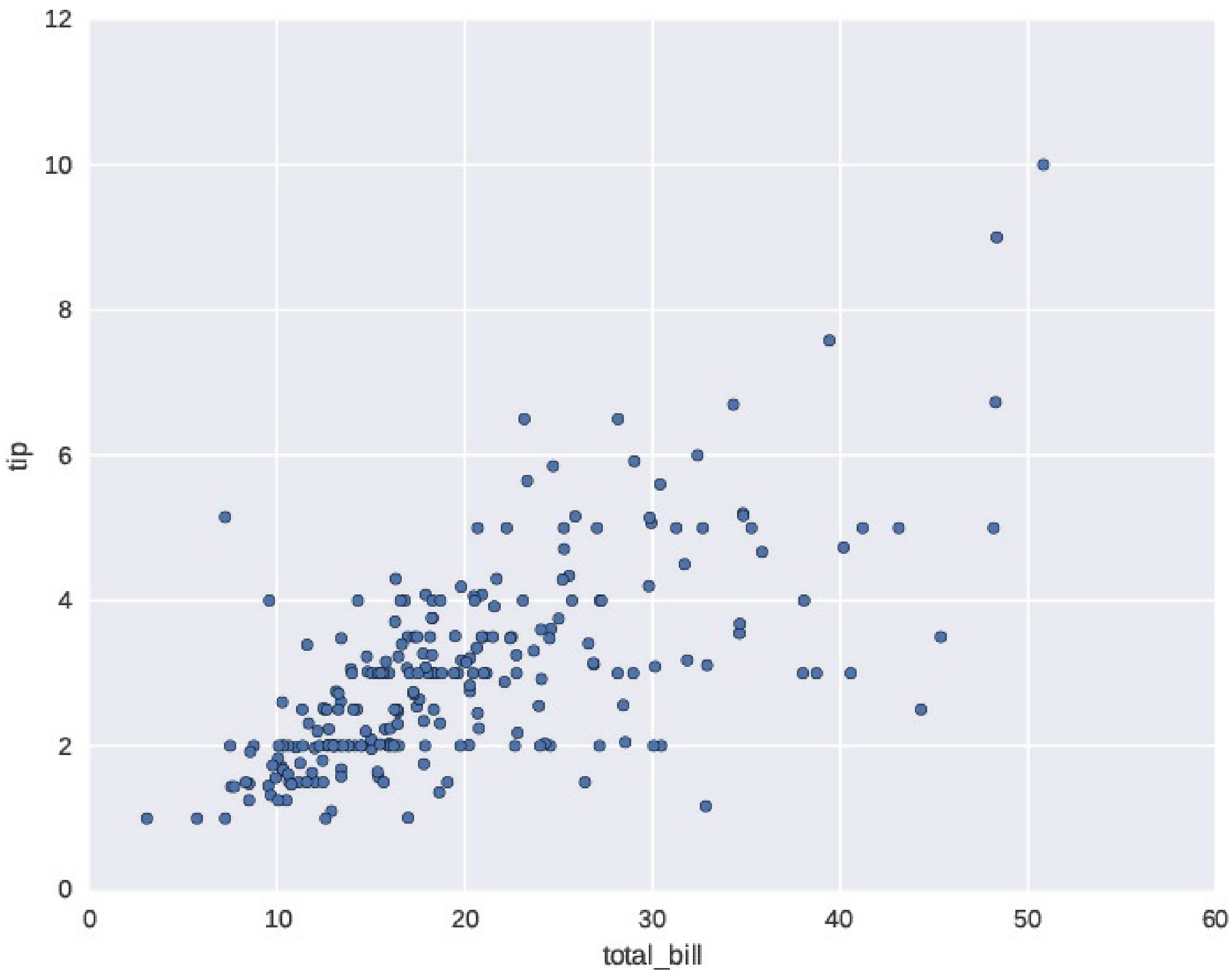
```
tips['tip'] .plot.kde ()
```



3.7.3 Scatter Plot

Scatter plots are created by using the `Data Frame.plot, scatter` function.

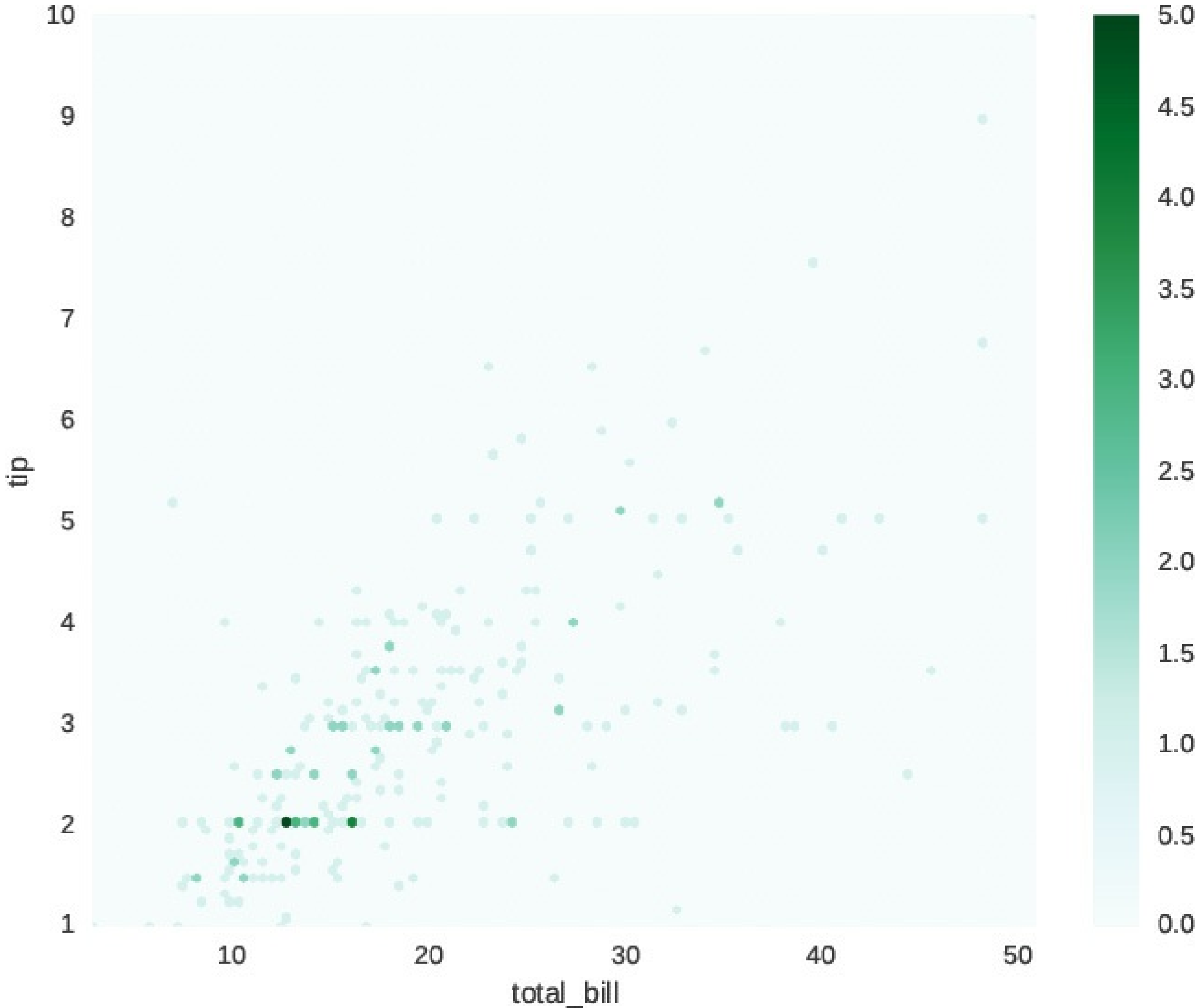
```
tips.plot.scatter(x='total_bill',   y='tip')
```



3.7.4 Hexbin Plot

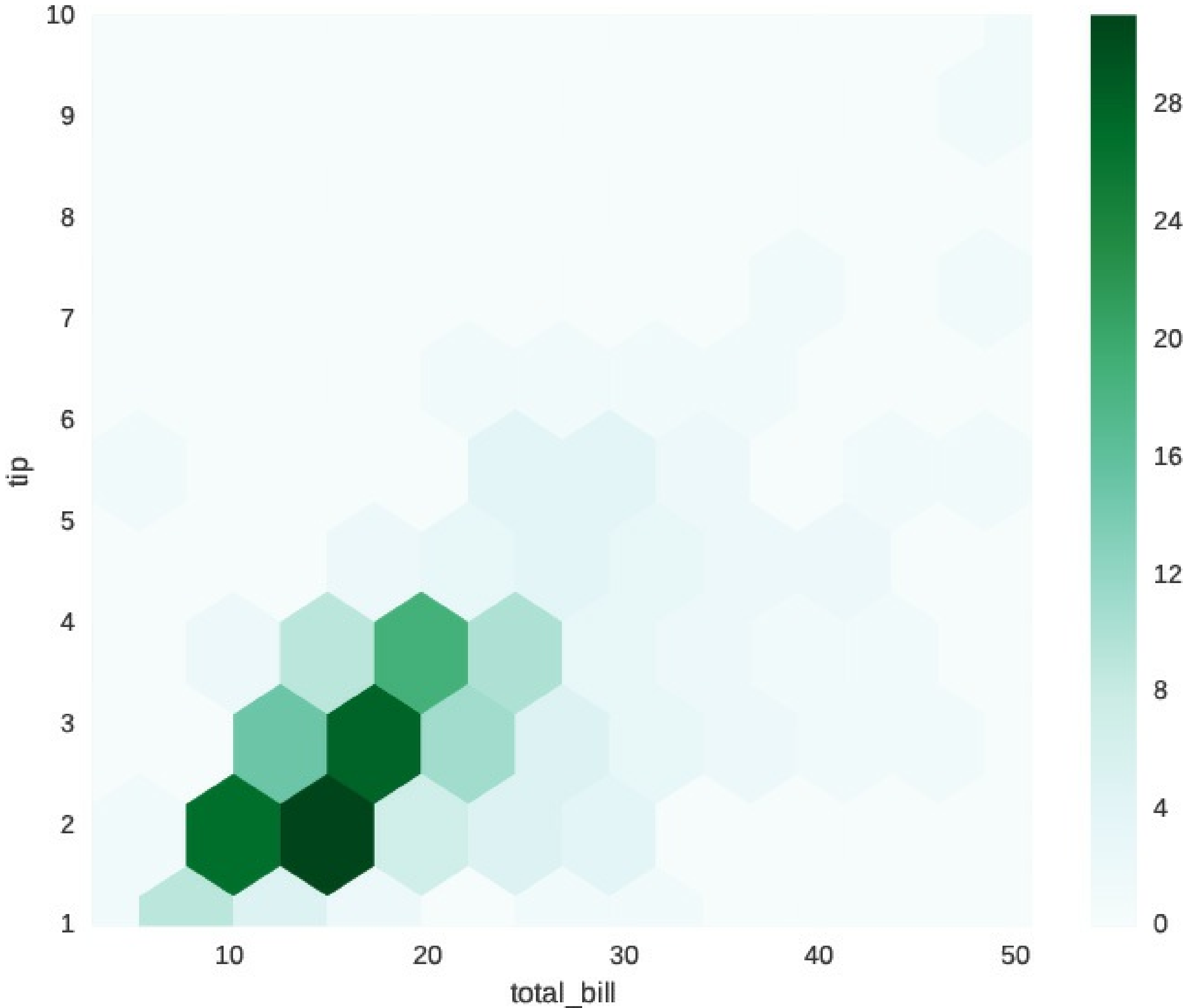
Hexbin plots are created using the `Dataframe.pit.hexbin` function.

```
tips.plot.hexbin(x='total_bill',   y='tip')
```



Gridsize can be adjusted with the `gridsize` parameter

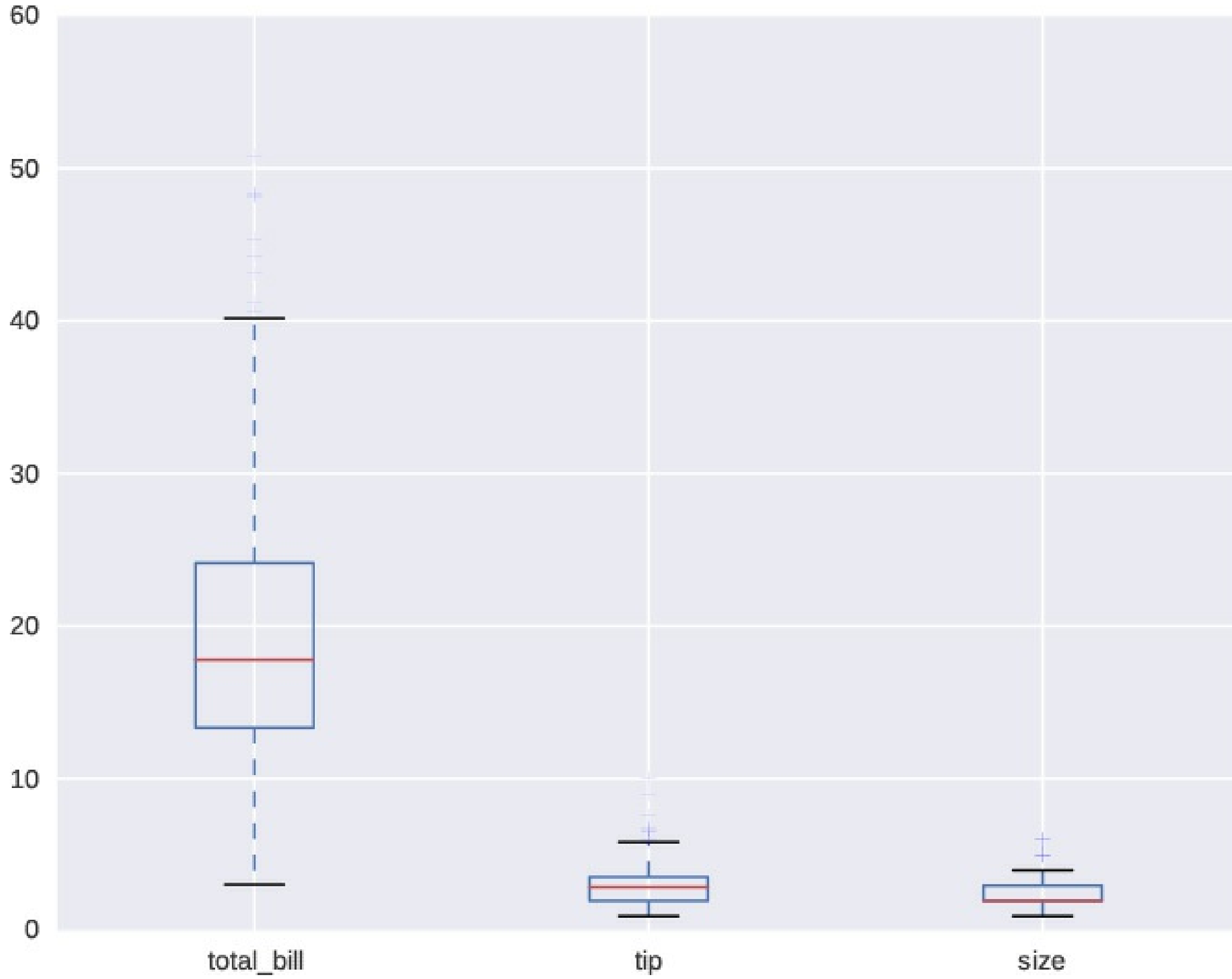
```
tips.plot.hexbin(x='total_bill',   y='tip',   gridsize=10)
```



3.7.5 Box Plot

Box plots are created with the `DataFrame.plot.box` function.

```
tips.plot.box()
```

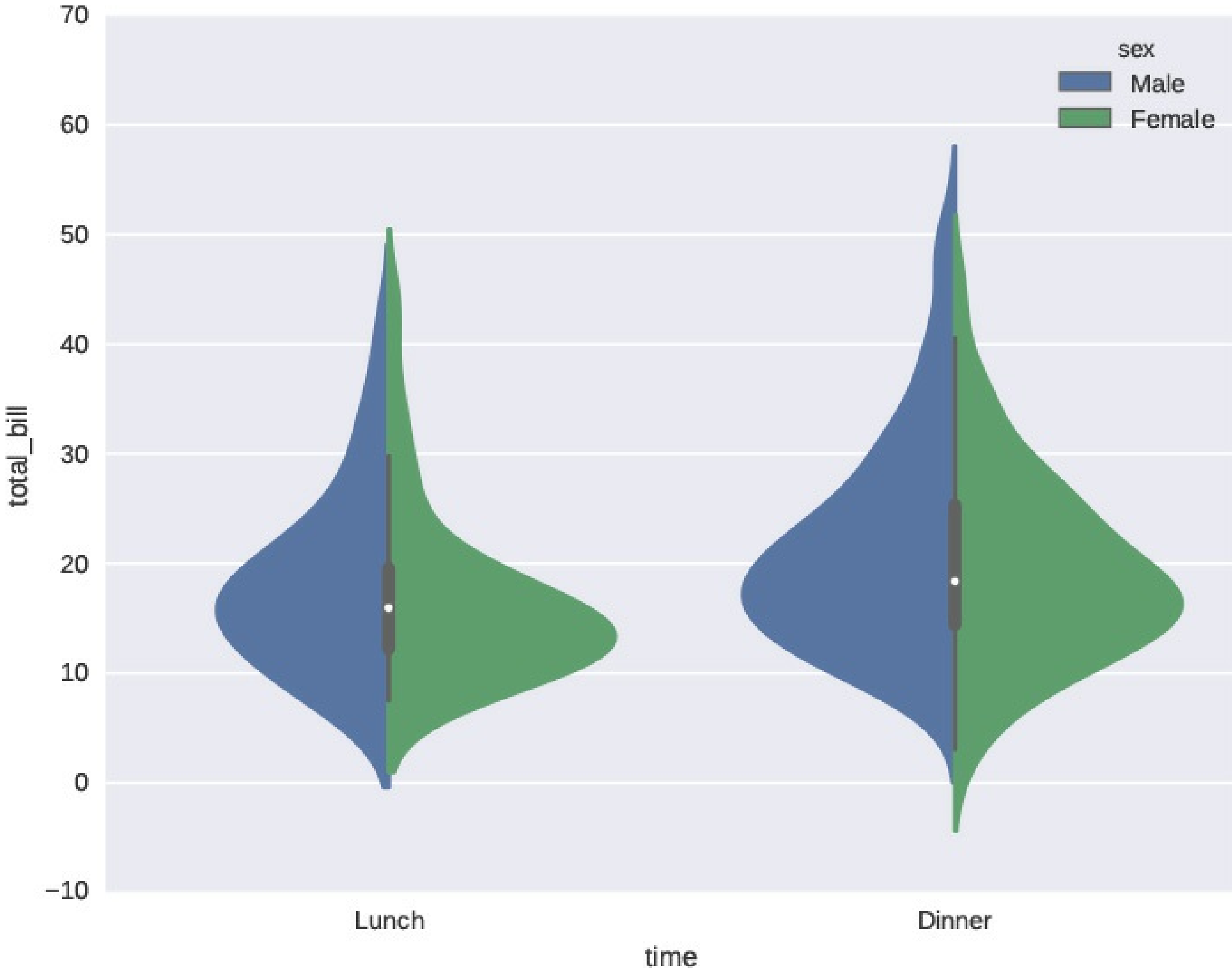


3.8 Themes and Styles

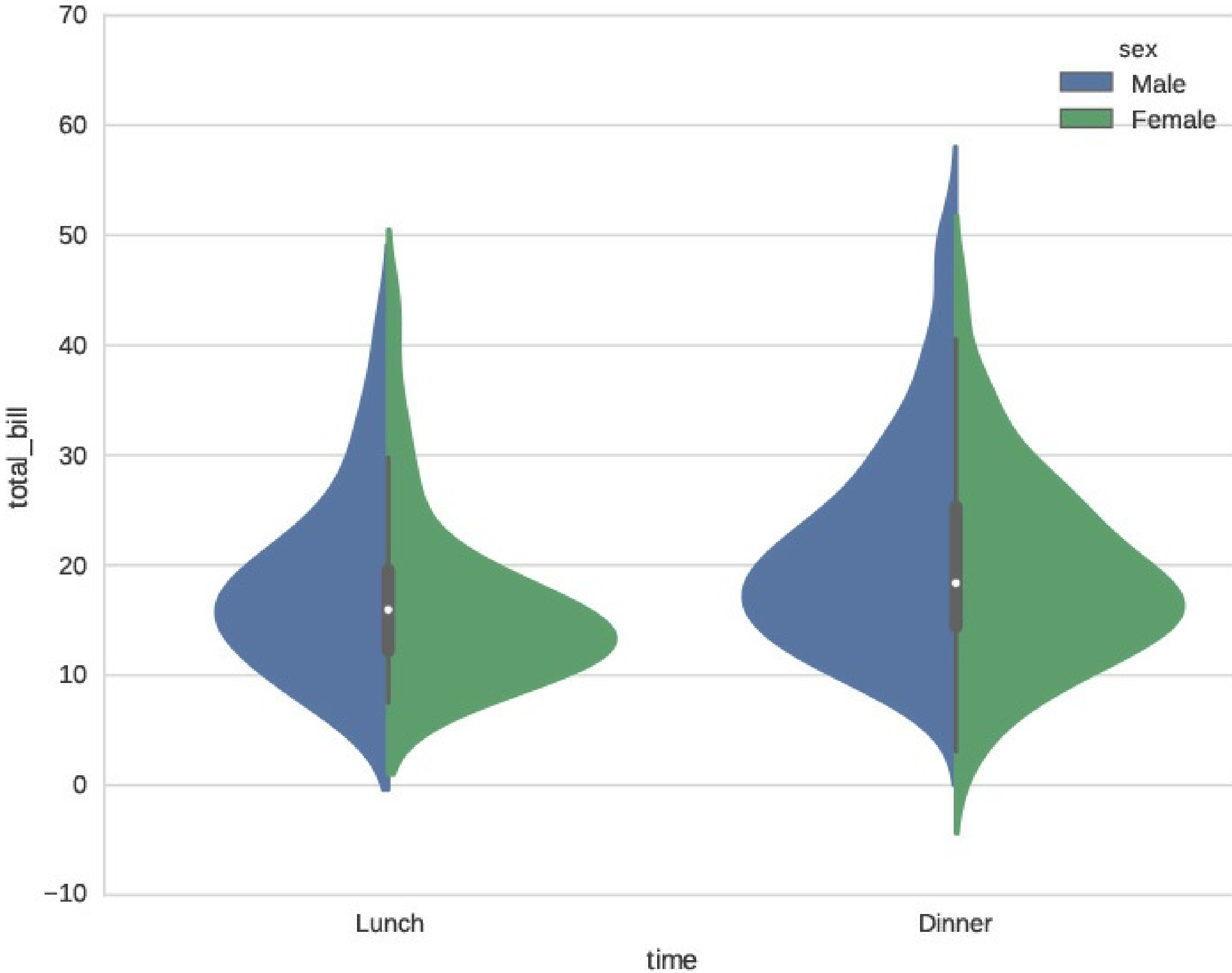
The `seaborn` plots shown in this chapter have all used the default plot styles. We can change the plot style with the `sns. set_style` function. Typically this function is run just once at the top of your code; all subsequent plots will use the style set.

The styles that come with `seaborn` are `darkgrid, whitegrid, dark, white`, and `ticks`.

```
# initial plot for comparison
violin = sns.violinplot(x='time',   y='total_bill',
                        hue='sex',   data=tips,
                        split=True)
```

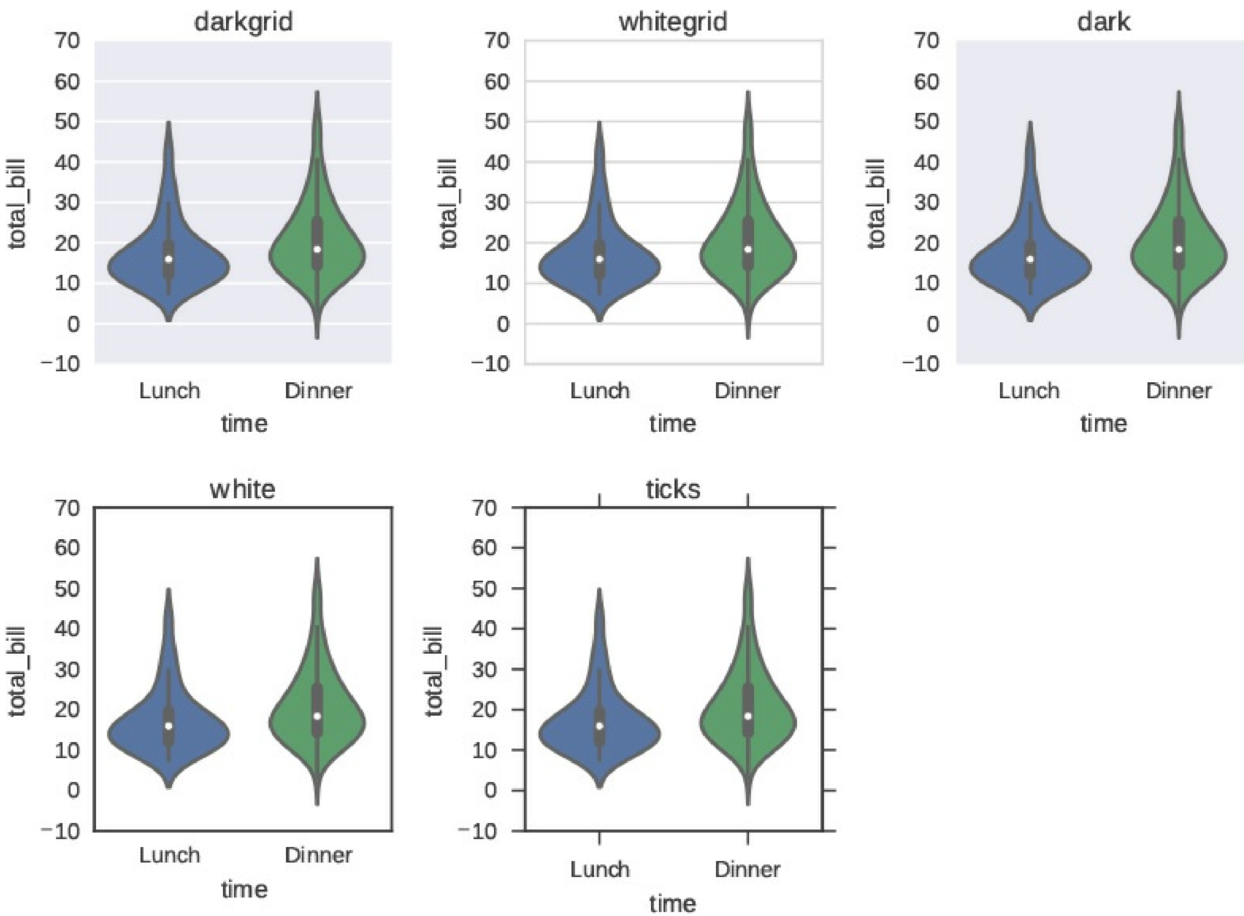



```
# set style and plot
sns set_style('whitegrid')
violin = sns.violinplot(x='time',   y='total_bill',
                        hue='sex',   data=tips,
                        split=True)
```



The following code shows what all the styles look like.

```
fig = pit.figure ()
seaborn_styles =   ['darkgrid',   'whitegrid',   'dark',   'wh:
for idx,   style in enumerate(seaborn_styles):
    plot_position = idx + 1
    with sns.axes_style(style):
        ax = fig.add_subplot(2,   3,  plot_position)
        violin = sns.violinplot(x='time' ,   y='total_bill',
                                data=tips,  ax=ax)
        violin.set_title(style)
fig.tight_layout()
```



3.9 Conclusion

Data visualization is an integral part of exploratory data analysis and data presentation. This chapter gives an introduction to start exploring and presenting your data. As we continue through the book, we will learn about more complex visualizations.

There are a myriad of plotting and visualization resources on the internet. The `seaborn` documentation[9], `pandas` visualization documentation[10], and matplotlib documentation[11] will all provide ways to further tweak your plots (e.g., colors, line thickness, legend placement, figure annotations, etc.). Other resources include colorbrewer[12] to help pick good color schemes. The plotting libraries mentioned in this chapter also have various color schemes that can be used.

9 https://stanford.edu/~mwaskom/software/seaborn/api.html

[10] http://paridas.pydata.org/paridas-docs/stable/visualizatiori.html

[11] http://matplotlib.org/api/index.html

[12] http://colorbrewer2.org/

Chapter 4. Data Assembly

4.1 Introduction

Hopefully by now, you are able to load in data into `pandas` and do some basic visualizations. This part of the book will focus on various data cleaning tasks. We begin with assembling a dataset for analysis.

When given a data problem, all of the information that we need may be recorded in separate files and data frames. For example, there may be a separate table on company information and another table on stock prices. If we wanted to look at all the stock prices within the tech industry we may first have to find all the tech companies from the company information table, and then combine it with the stock price data to get the data we need for our question. The data was split up into separate tables to reduce the amount of redundant information (we don't need to store the company information with each stock price entry), but it means we as data analysts must combine the relevant data ourselves for our question.

Other times a single dataset will be split into multiple parts. This may be timeseries data where each date is in a separate file, or a file may have been split into parts to make the individual files smaller. You may also need to combine data from multiple sources to answer a question (e.g., combining latitudes and longitudes with zip codes). In both cases, you will need to combine data into a single dataframe for analysis.

4.2 Concept map

1. Prior knowledge

(a) Loading data

(b) Subsetting data

(c) functions and class methods

4.3 Objectives

This chapter will cover:

1. Tidy data

2. Concatenating data

3. Merging datasets

4.4 Concatenation

One of the (conceptually) easier forms of combining data is concatenation. Concatenation can be thought of appending a row or column to your data. This is can happen if your data was split into parts or if you made a calculation that you want to append.

Concatenation is all accomplished by using the `concat` function from pandas.

4.4.1 Adding rows

Let's begin with some example data sets so you can see what is actually happening.

```
import pandas as pd

df1 = pd.read_csv('../data/concat_1.csv')
df2 = pd.read_csv('../data/concat_2.csv')
df3 = pd.read_csv('../data/concat_3.csv')
```

```
print(df1)                       print(df2)                        print(d

    A   B   C   D                    A   B   C   D                       A
0  a0  b0  c0  d0                0  a4  b4  c4  d4                 0    a8
1  a1  b1  c1  d1                1  a5  b5  c5  d5                 1    a9
2  a2  b2  c2  d2                2  a6  b6  c6  d6                 2   a10
3  a3  b3  c3  d3                3  a7  b7  c7  d7                 3   a11
```

Stacking the datarames on top of each other uses the `concat` function in `pandas` where all the dataframes to be concatenated are passed in a `list`.

```
row_concat = pd.concat([df1,   df2,   df3])
print(row_concat)
```

```
     A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3
0   a4   b4   c4   d4
1   a5   b5   c5   d5
2   a6   b6   c6   d6
3   a7   b7   c7   d7
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

You can see `concat` blindly stacks the dataraames together. If you look at the row names (a.k.a row index), they are also simply a stacked version of the original row indices.

If we tried the various subsetting methods from Table 2-1, the table will subset as expected.

```
# subset  the 4th row of the concatenated dataframe
print(row_concat.iloc[3,  ])
```

```
A    a3
B    b3
C    c3
D    d3
Name: 3, dtype: object
```

Question

What happens when you use `loc` or `ix` to subset the new dataframe?

In Chapter 2.4.1, I showed how you can create a `series`. However, if we create a new series to append to a dataframe, you'd quickly see, that it does not append correctly.

```
# create a new row of data
new_row_series = pd.Series(['n1',   'n2',   'n3',   'n4'])
print(new_row_series)
```

```
 0    n1
 1    n2
 2    n3
 3    n4
 dtype: object
```

```
# attempt  to add the new row to a dataframe
print(pd.concat([df1,  new_row_series]))
```

```
      A    B    C    D    0
 0   a0   b0   c0   d0  NaN
 1   a1   b1   c1   d1  NaN
 2   a2   b2   c2   d2  NaN
 3   a3   b3   c3   d3  NaN
 0  NaN  NaN  NaN  NaN   n1
 1  NaN  NaN  NaN  NaN   n2
 2  NaN  NaN  NaN  NaN   n3
 3  NaN  NaN  NaN  NaN   n4
```

The first things we will notice are `NaN` values. This is simply Python's way of representing a 'missing value' (Chapter 5). Next, we were hoping to append our new values as a row. Not only did our code not append the values as a row, it created a new column completely misaligned with everything else.

If we pause to think about what actually is happening, we can see the results actually make sense. First, if we look at the new indices that were added, It is very similar to how we concatenated dataframes earlier. The indices of the `newrow series` object are analogs to the row numbers of the dataframe. Next, since our series did not have a matching column, our `newrow` was added to a new column.

To fix this, we can turn our series into a dataframe. This data frame would have 1 row of data, and the column names would be the ones the data would bind to.

```
                                # note the double brackets
new_row_df = pd.DataFrame([['n1',   'n2',   'n3',   'n4']],
                                columns=['A',   'B',   'C',   'D'])
```

```
print(new_row_df)
```

```
    A    B   C   D
 0 n1   n2  n3  n4
```

```
print(pd.concat([df1,  new_row_df]))
```

```
     A   B   C   D
 0  a0  b0  c0  d0
 1  a1  b1  c1  d1
 2  a2  b2  c2  d2
 3  a3  b3  c3  d3
 0  n1  n2  n3  n4
```

`concat` is a general function that can concatenate multiple things at once. If you just needed to append a single object to an existing dataframe, there's the `append` function for that.

Using a `DataFrame` Using a single-row `DataFrame`

```
print(df1.append(df2))
```

```
     A   B   C   D
 0  a0  b0  c0  d0
 1  a1  b1  c1  d1
 2  a2  b2  c2  d2
 3  a3  b3  c3  d3
 0  a4  b4  c4  d4
 1  a5  b5  c5  d5
 2  a6  b6  c6  d6
 3  a7  b7  c7  d7
```

```
print(df1.append(new_row_df))
```

```
     A   B   C   D
 0  a0  b0  c0  d0
 1  a1  b1  c1  d1
 2  a2  b2  c2  d2
 3  a3  b3  c3  d3
 0  n1  n2  n3  n4
```

Using a Python Dictionary

```
data_dict = {'A':   'n1',
             'B':   'n2',
             'C': 'n3',
```

```
             'D':   'n4'}

print(df1.append(data_dict,   ignore_index=True))

    A   B   C   D
 0  a0  b0  c0  d0
 1  a1  b1  c1  d1
 2  a2  b2  c2  d2
 3  a3  b3  c3  d3
 4  n1  n2  n3  n4
```

`Ignoring the index` We saw in the last example when we tried to add a `dict` to a dataframe, we had to use the `ignore_index` parameter. If we look closer, you can see the row index also incremented by 1, and did not repeat a previous index value.

If we simply wanted to concatenate or append data together, we can use the `ignore_index` to reset the row index after the concatenation.

```
row_concat_i = pd.concat([df1,   df2,   df3],   ignore_index Tı
print(row_concat_i)

         A     B     C     D
 0      a0    b0    c0    d0
 1      a1    b1    c1    d1
 2      a2    b2    c2    d2
 3      a3    b3    c3    d3
 4      a4    b4    c4    d4
 5      a5    b5    c5    d5
 6      a6    b6    c6    d6
 7      a7    b7    c7    d7
 8      a8    b8    c8    d8
 9      a9    b9    c9    d9
 10    a10   b10   c10   d10
 11    a11   b11   c11   d11
```

4.4.2 Adding columns

Concatenating columns is very similar to concatenating rows. The main difference is the `axis` parameter in the `concat` function. The default value of `axis` has a value of `0,` so it will concatenate row-wise. However, if we pass `axis=1` to the function, it will concatenate column-wise.

```
col_concat = pd.concat([df1,   df2,   df3],   axis=1)
print(col_concat)

      A    B    C    D    A     B    C    D     A     B     C
 0    a0   b0   c0   d0   a4    b4   c4   d4    a8    b8    c8
 1    a1   b1   c1   d1   a5    b5   c5   d5    a9    b9    c9
 2    a2   b2   c2   d2   a6    b6   c6   d6   a10   b10   c10
 3    a3   b3   c3   d3   a7    b7   c7   d7   a11   b11   c11
```

If we try to subset based on column names, we will get a similar result when we concatenated row-wise and subset by row index.

```
print(col_concat['A'])

      A    A     A
 0    a0   a4    a8
 1    a1   a5    a9
 2    a2   a6   a10
 3    a3   a7   a11
```

Adding a single column to a dataframe can be done directly without using any specific pandas function. Simply pass a new column name the vector you want assigned to the new column.

```
col_concat['new_col_list']   =   ['n1',   'n2',   'n3',   'n4']
print(col_concat)

      A     B    C     D    A    B    C    D     A     B     C
 0    a0    b0   c0    d0   a4   b4   c4   d4    a8    b8    c8
 1    a1    b1   c1    d1   a5   b5   c5   d5    a9    b9    c9
 2    a2    b2   c2    d2   a6   b6   c6   d6   a10   b10   c10
 3    a3    b3   c3    d3   a7   b7   c7   d7   a11   b11   c11

col_concat['new_col_series']   = pd.Series(['n1',   'n2',   'n3
print(col_concat)

      A     B    C     D    A    B    C    D     A     B     C
 0    a0    b0   c0    d0   a4   b4   c4   d4    a8    b8    c8
 1    a1    b1   c1    d1   a5   b5   c5   d5    a9    b9    c9
 2    a2    b2   c2    d2   a6   b6   c6   d6   a10   b10   c10
 3    a3    b3   c3    d3   a7   b7   c7   d7   a11   b11   c11
```

Using the `concat` function still works, as long as you pass it a dataframe. This does require a bit more unnecessary code.

Finally, we can choose to reset the column indices so we do not have duplicated column names.

```
print(pd.concat([df1,   df2,   df3],   axis=1,   ignore_index=T
```

```
     0    1    2    3    4    5    6    7     8     9    10
0   a0   b0   c0   d0   a4   b4   c4   d4    a8    b8    c8
1   a1   b1   c1   d1   a5   b5   c5   d5    a9    b9    c9
2   a2   b2   c2   d2   a6   b6   c6   d6   a10   b10   c10
3   a3   b3   c3   d3   a7   b7   c7   d7   a11   b11   c11
```

4.4.3 Concatenation with different indices

The examples shown so far assume a simple row or column concatenation. It also assumes that the new row(s) had the same column names or the column(s) had the same row indices.

Here I will show you what happens when the row and column indices are not aligned.

4.4.3.1 Concatenate rows with different columns

Let's modify our dataframes for the next few examples.

```
df1.columns = ['A', 'B', 'C', 'D']
df2.columns = ['E', 'F', 'G', 'H']
df3.columns =   ['A',   'C',   'F',   'H']
```

```
print(df1)                       print(df2)                       pr
```

```
     A    B    C    D             E    F    G    H
0   a0   b0   c0   d0        0   a4   b4   c4   d4              0
1   a1   b1   c1   d1        1   a5   b5   c5   d5              1
2   a2   b2   c2   d2        2   a6   b6   c6   d6              2
3   a3   b3   c3   d3        3   a7   b7   c7   d7              3
```

If we try to concatenate the dataframes like we did in section 4.4.1, you will now see the dataframes do much more than simply stack one on top of the other. The columns will align themselves, and a `NaN` value will fill any of the missing areas.

```
row_concat = pd.concat([df1,   df2,   df3])
print(row_concat)

      A         B     C     D     E      F      G     H
 0    a0        b0    c0    d0   NaN    NaN    NaN   NaN
 1    a1        b1    c1    d1   NaN    NaN    NaN   NaN
 2    a2        b2    c2    d2   NaN    NaN    NaN   NaN
 3    a3        b3    c3    d3   NaN    NaN    NaN   NaN
 0   NaN       NaN   NaN   NaN    a4     b4     c4    d4
 1   NaN       NaN   NaN   NaN    a5     b5     c5    d5
 2   NaN       NaN   NaN   NaN    a6     b6     c6    d6
 3   NaN       NaN   NaN   NaN    a7     b7     c7    d7
 0    a8       NaN    b8   NaN   NaN     c8    NaN    d8
 1    a9       NaN    b9   NaN   NaN     c9    NaN    d9
 2   a10       NaN   b10   NaN   NaN    c10    NaN   d10
 3   a11       NaN   b11   NaN   NaN    c11    NaN   d11
```

One way to not have any `NaN` missing values is to only keep the columns that are in common from the list of objects to be concatenated. There is a parameter named `join` that accomplishes this. By default it has a value of `'outer'`, meaning it will keep all the columns. However, we can set `join='inner'` to keep only the columns that

If we try to keep only the columns from all 3 dataframes, we will get an empty dataframe since there are no columns in common.

```
print(pd.concat([df1,   df2,   df3],   join='inner'))
Empty DataFrame
Columns:   []
Index:   [0,   1,   2,   3,   0,   1,   2,   3,   0,   1,   2,
```

If we use the dataframes that have columns in common, only the columns that all of them share will be returned.

```
print(pd.concat([df1,df3],   ignore_index=False,   join='inner

      A      C
 0    a0     c0
 1    a1     c1
 2    a2     c2
 3    a3     c3
 0    a8     b8
 1    a9     b9
 2   a10    b10
```

```
3   a11    b11
```

4.4.3.2 Concatenate columns with different rows

Let's take our dataframes and modify them again with different row indices. I am building on the same dataframe modifications from Section 4.4.3.1.

```
df1.index = [0, 1, 2, 3]
df2.index = [4, 5, 6, 7]
df3.index =  [0,   2,   5,   7]
```

```
print(df1)

      A     B      C     D
0     a0    b0     c0    d0
1     a1    b1     c1    d1
2     a2    b2     c2    d2
3     a3    b3     c3    d3
```

```
print(df2)

      E     F     G     H
4     a4    b4    c4    d4
5     a5    b5    c5    d5
6     a6    b6    c6    d6
7     a7    b7    c7    d7
```

When we concatenate along `axis=1`, we get the same results from concatenating along `axis=0`. The new dataframes will be added column wise and matched against their respective row indices. Missing values will fill in the areas where the indices did not align.

```
col_concat = pd.concat([df1,   df2,   df3],   axis=1)
print(col_concat)

      A          B     C       D     E      F       G     H
 0    a0         b0    c0      d0   NaN    NaN     NaN   NaN
 1    a1         b1    c1      d1   NaN    NaN     NaN   NaN
 2    a2         b2    c2      d2   NaN    NaN     NaN   NaN
 3    a3         b3    c3      d3   NaN    NaN     NaN   NaN
 4   NaN        NaN   NaN     NaN    a4     b4      c4    d4
 5   NaN        NaN   NaN     NaN    a5     b5      c5    d5
 6   NaN        NaN   NaN     NaN    a6     b6      c6    d6
 7   NaN        NaN   NaN     NaN    a7     b7      c7    d7
```

Lastly, just like we did when we concatenated row-wise, we can choose to only keep the results when there are matching indices by using `join ='inner'`.

```
print(pd.concat([df1,   df3],   axis=1,   join='inner'))
```

```
     A     B      C     D     A    C     F    H
0   a0    b0     c0    d0    a8   b8    c8   d8
2   a2    b2     c2    d2    a9   b9    c9   d9
```

4.5 Merging multiple datsets

The end of the previous section alluded to a few database concepts. The `join ='inner'` and the default `join ='outer'` parameters come from working with databases when we want to merge tables.

Instead of simply having a row or column index that we want to concatenate values to, there will be times when you have 2 or more dataframes that you want to combine based on common data values. This is known in the database world as performing a "join".

Pandas has a `pd.join` command that uses `pd.merge` under the hood. `join` will merge dataframe objects by an index, but the `merge` command is much more explicit and flexible. If you are only planning to merge dataframes by the row index, you can look into the `join` function[1].

We will be using the survey data in this series of examples.

```
person = pd.read_csv('../data/survey_person.csv')
site = pd.read_csv('../data/survey_site.csv')
survey = pd.read_csv('../data/survey_survey.csv')
visited = pd.read_csv('../data/survey_visited.csv')
```

[1] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.join.html

```
print(person)

       ident    personal     family
0       dyer     William       Dyer
1         pb       Frank    Pabodie
2       lake    Anderson       Lake
3        roe   Valentina    Roerich
4   danforth       Frank   Danforth
```

```
print(site)
```

```
print(survey)

     taken person quant
0      619   dyer   rad
1      619   dyer   sal
2      622   dyer   rad
3      622   dyer   sal
4      734     pb   rad
5      734   lake   sal
6      734     pb  temp
7      735     pb   rad
```

```
    name    lat    long
0   DR-1 -49.85 -128.57
1   DR-3 -47.15 -126.72
2  MSK-4 -48.87 -123.40
```

```
print(visited)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
8      735   NaN   sal
9      735   NaN  temp
10     751    pb   rad
11     751    pb  temp
12     751  lake   sal
13     752  lake   rad
14     752  lake   sal
15     752  lake  temp
16     752   roe   sal
17     837  lake   rad
18     837  lake   sal
19     837   roe   sal
20     844   roe   rad
```

Currently, our data is split into multiple parts, where each part is an observational unit. If we wanted to look at the dates at each site with the lat long of the site. We would have to combine (and merge) multiple dataframes. We do this with the `merge` function in pandas. `merge` is actually a `DataFrame` method.

When we call this method, the dataframe that is called will be referred to the one on the `' left '`. Within the `merge` function, the first parameter is the `' right'` dataframe. The next parameter is `how` the final merged result looks. See Table 4-1 for more details. The next, we set the `on` parameter. This specifies which columns to match on. If the left and right columns are not the same name, we can use the `left_on` and `right_on` parameters instead.

Table 4-1: My caption

Pandas	SQL	Description
left	left outer	Keep all the keys from the left
right	right outer	Keep all the keys from the right

outer	full outer	Keep all the keys from both left and right
inner	inner	keep only the keys that exist in the left and right

4.5.1 one-to-one

The simplest type of merge we can do is when we have 2 dataframes where we want to join one column to another column, and when the columns we want to join on are

For this example I am going to modify the `visited` dataframe so there are no duplicated `site` values.

```
visited_subset = visited.ix[[0,   2,   6],   ]
```

We can perform our one-to-one merge as follows:

```
# the default value for 'how'  is  'inner'
# so it doesn't need to be specified
o2o_merge = site.merge(visited_subset,
                       left_on='name',   right_on='site')
print(o2o_merge)
        name    lat    long       ident     site          dated
 0      DR-1 -49.85 -128.57         619     DR-1     1927-02-08
 1      DR-3 -47.15 -126.72         734     DR-3     1939-01-07
 2     MSK-4 -48.87 -123.40         837    MSK-4     1932-01-14
```

You can see here that we now have a new dataframe from 2 separate dataframes where the rows were matched based on a particular set of columns. In SQL speak, the columns used to match are called 'key(s)'.

4.5.2 many-to-one

If we choose to do the same merge, but this time without using the subsetted `visited` dataframe, we would perform a many-to-one merge. This happens when performing a merge and one of the dataframe has key values that repeat.

When this happens, the dataframe that contains the single observations will be duplicated in the merge.

```
m2o_merge = site.merge(visited,   left_on='name',   right_on='s
print(m2o_merge)
        name      lat      long   ident    site        dated
 0      DR-1   -49.85   -128.57     619    DR-1   1927-02-08
 1      DR-1   -49.85   -128.57     622    DR-1   1927-02-10
 2      DR-1   -49.85   -128.57     844    DR-1   1932-03-22
 3      DR-3   -47.15   -126.72     734    DR-3   1939-01-07
 4      DR-3   -47.15   -126.72     735    DR-3   1930-01-12
 5      DR-3   -47.15   -126.72     751    DR-3   1930-02-26
 6      DR-3   -47.15   -126.72     752    DR-3          NaN
 7     MSK-4   -48.87   -123.40     837   MSK-4   1932-01-14
```

As you can see, the `site` information `(name, lat, and long)` were duplicated and matched to the `visited` data.

4.5.3 many-to-many

Lastly, there will be times when we want to perform a match based on multiple columns. This can also be performed.

Let's say we have 2 dataframes that come from the person merged with survey, and another dataframe that comes from visited merged with survey.

```
ps = person.merge(survey, left_on='ident', right_on='person')
vs = visited.merge(survey,   left_on='ident',   right_on='taken
print(ps)

      ident     personal     family    taken person quant
 0     dyer      William       Dyer      619   dyer   rad
 1     dyer      William       Dyer      619   dyer   sal
 2     dyer      William       Dyer      622   dyer   rad
 3     dyer      William       Dyer      622   dyer   sal
 4       pb        Frank    Pabodie      734     pb   rad
 5       pb        Frank    Pabodie      734     pb temp
 6       pb        Frank    Pabodie      735     pb   rad
 7       pb        Frank    Pabodie      751     pb   rad
 8       pb        Frank    Pabodie      751     pb temp
 9     lake     Anderson       Lake      734   lake   sal
 10    lake     Anderson       Lake      751   lake   sal
 11    lake     Anderson       Lake      752   lake   rad
```

```
 12     lake      Anderson        Lake       752   lake   sal
 13     lake      Anderson        Lake       752   lake temp
 14     lake      Anderson        Lake       837   lake   rad
 15     lake      Anderson        Lake       837   lake   sal
 16      roe    Valentina    Roerich       752    roe   sal
 17      roe    Valentina    Roerich       837    roe   sal
 18      roe    Valentina    Roerich       844    roe   rad
```

```
print(vs)
```

```
        ident      site          dated      taken person quant
 0        619      DR-1    1927-02-08        619   dyer   rad
 1        619      DR-1    1927-02-08        619   dyer   sal
 2        622      DR-1    1927-02-10        622   dyer   rad
 3        622      DR-1    1927-02-10        622   dyer   sal
 4        734      DR-3    1939-01-07        734     pb   rad
 5        734      DR-3    1939-01-07        734   lake   sal
 6        734      DR-3    1939-01-07        734     pb temp
 7        735      DR-3    1930-01-12        735     pb   rad
 8        735      DR-3    1930-01-12        735    NaN   sal
 9        735      DR-3    1930-01-12        735    NaN temp
 10       751      DR-3    1930-02-26        751     pb   rad
 11       751      DR-3    1930-02-26        751     pb temp
 12       751      DR-3    1930-02-26        751   lake   sal
 13       752      DR-3           NaN        752   lake   rad
 14       752      DR-3           NaN        752   lake   sal
 15       752      DR-3           NaN        752   lake temp
 16       752      DR-3           NaN        752    roe   sal
 17       837     MSK-4    1932-01-14        837   lake   rad
 18       837     MSK-4    1932-01-14        837   lake   sal
 19       837     MSK-4    1932-01-14        837    roe   sal
 20       844      DR-1    1932-03-22        844    roe   rad
```

We can perform a many-to-many merge by passing the multiple columns to match on in a python list.

```
ps_vs = ps.merge(vs,
                 left_on=['ident',   'taken',   'quant',   'rea
                 right_on=['person',   'ident',   'quant',   'ı
```

If we just take a look at the first row of data:

```
print(ps_vs.ix[0,   ])
```

```
 ident_x           dyer
 personal       William
```

```
family              Dyer
taken_x              619
person_x            dyer
quant                rad
reading             9.82
ident_y              619
site                DR-1
dated         1927-02-08
taken_y              619
person_y            dyer
Name: 0, dtype: object
```

Pandas will automatically add a suffix to a column name if there are collisions in the name. the `jx` refers to values from the left dataframe, and the `_y` suffix comes from values in the right dataframe.

4.6 Summary

There will be times when you need to combine various parts or data or multiple datasets depending on the question you are trying to answer. One thing to keep in mind, the data you need for analysis, does not necessarily mean the best shape of data for storage.

The survey data used in the last example came in 4 separate parts that needed to be merged together. After we merged the tables together, you will notice a lot of redundant information across rows. From a data storage and entry point of view, each of these duplications can lead to errors and data inconsistency. This is what Hadley meant by "each type of observational unit forms a table".

Chapter 5. Missing Data

5.1 Introduction

Rarely will you be given a dataset without any missing values. There are many representations of missing data. In databases they are `NULL` values, Certain programming languages will use `NA`, and depending on where you get your data, missing values can be an empty string, `''` or even numeric values such as `88` or `99`.

Pandas has displays missing values as `NaN`.

Concept map

1. Prior knowledge

(a) importing libraries

(b) slicing and indexing data

(c) using functions and methods

(d) using function parameters

Objectives

This chapter will cover:

1. What is a missing value

2. How are missing values created

3. How to recode and make calculations with missing values

5.2 What is a NaN value

We can get the `NaN` value from `numpy`. You may see missing values in python used or displayed in a few ways: `NaN, NAN`, or `nan`. They are all equivalent.

```
# Just import the numpy missing values ## TODO SEE APPENDIX
from numpy import NaN, NAN, nan
```

Missing values are different than other types of data, in that they don't really equal anything. The data is missing, so there is no concept of equality. `NaN` is not be equivalent to `0` or an empty string, `''`.

We can illustrate this in python by testing it's equality.

```
print(NaN == True) print(NaN == False)print(NaN == 0)    print(N
```

```
 |False              |False             |False             |False
```

To illustrate the lack of equality, missing values are also not equal to misisng values.

```
print(NaN == NaN) print(NaN == nan) print(NaN == NAN) print(nan
```

```
 |False             |False             |False             |False
```

Pandas has built-in methods to test for a missing value.

```
import pandas as pd
```

```
print(pd.isnull(NaN))    print(pd.isnull(nan))    print(pd.isnul
```

```
 True                     True                     True
```

Pandas also has methods for testing non-missing values

```
print(pd.notnull(NaN))  print(pd.notnull(42))    print(pd.notnu
```

```
 False                    True                     True
```

5.3 Where do missing values come from?

We can get missing values from loading in data with missing values, or from the data munging process.

5.3.1 Load data

The survey data we used in Chapter 4 had a dataset, `visited`, which contained missing data. When we loaded the data, `pandas` automatically found the missing data cell, and gave us a `dataframe` with the `NaN` value in the appropriate cell. In the `read_csv` function, there are three parameters that relate to reading in missing values: `na_values, keep default_na`, and `na_filter`.

`na_values` allow you to specify additional missing or `NaN` values. You can either pass in a python `str` or list-like object for to be automatically coded as missing values when the file is read. There are already default missing values, such as `NA, NaN`, or nan, which is why this parameter is not always used. Some health data will code `99` as a missing value; an example of a value you would set in this field is `na_values=[99]`.

`keep_default_na` is a bool that allows you to specify whether any additional values need to be considered as missing. This parameter is `True` by default, meaning, any additional missing values specified with the `na_values` parameter will be appended to the list of missing values. However, `keep_default_na` can also be set to keep default na=False to only use the missing values specified in `na_values`

Lastly, `na_filter` is a bool that will specify whether or not any values will be read as missing. The default value of `na_filter =True` means that missing values will be coded as a `NaN`. If we assign `na_filter =False`, then nothing will be recoded as missing. This can by though of as a means to tun off all the parameters set for na values and `keep_default_na`, but it really is used when you want a performance boost loading in data without missing values.

```
# set the location for data
visited_file = '../data/survey_visited.csv'

# load data with default values
print(pd.read_csv(visited_file))
```

```
    ident    site      datedxs
 0    619    DR-1   1927-02-08
 1    622    DR-1   1927-02-10
 2    734    DR-3   1939-01-07
 3    735    DR-3   1930-01-12
 4    751    DR-3   1930-02-26
 5    752    DR-3          NaN
 6    837   MSK-4   1932-01-14
 7    844    DR-1   1932-03-22

# load data without default missing values
print(pd.read_csv(visited_file,
                  keep_default_na=False))

   ident    site       dated
0    619    DR-1  1927-02-08
1    622    DR-1  1927-02-10
2    734    DR-3  1939-01-07
3    735    DR-3  1930-01-12
4    751    DR-3  1930-02-26
5    752    DR-3
6    837   MSK-4  1932-01-14
7    844    DR-1  1932-03-22

# manually specify missing valu
print(pd.read_csv(visited_file,
                  na_values=[’’
                  keep_default_na=False))

   ident    site       dated
0    619    DR-1  1927-02-08
1    622    DR-1  1927-02-10
2    734    DR-3  1939-01-07
3    735    DR-3  1930-01-12
4    751    DR-3  1930-02-26
5    752    DR-3         NaN
6    837   MSK-4  1932-01-14
7    844    DR-1  1932-03-22
```

5.3.2 Merged data

Chapter 4 showed how to combine datasets. Some of the examples in the chapter showed missing values in the output. If we recreate the merged table from Section 4.5.3, we will see missing values in the merged output.

```
visited = pd.read_csv('../data/survey_visited.csv')
survey = pd.read_csv('../data/survey_survey.csv')

print(visited)

   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22

print(survey)

     taken    person    quant   reading
0      619      dyer      rad      9.82
1      619      dyer      sal      0.13
2      622      dyer      rad      7.80
3      622      dyer      sal      0.09
4      734        pb      rad      8.41
5      734      lake      sal      0.05
6      734        pb     temp    -21.50
7      735        pb      rad      7.22
8      735       NaN      sal      0.06
9      735       NaN     temp    -26.00
10      751       pb      rad      4.35
11      751       pb     temp    -18.50
12      751     lake      sal      0.10
13      752     lake      rad      2.19
14      752     lake      sal      0.09
15      752     lake     temp    -16.00
16      752      roe      sal     41.60
17      837     lake      rad      1.46
18      837     lake      sal      0.21
19      837      roe      sal     22.50
20      844      roe      rad     11.25

vs = visited.merge(survey, left_on='ident', right_on='taken')
print(vs)

     ident      site          dated    taken     person    quar
0      619      DR-1     1927-02-08      619       dyer      ra
1      619      DR-1     1927-02-08      619       dyer      sa
2      622      DR-1     1927-02-10      622       dyer      ra
3      622      DR-1     1927-02-10      622       dyer      sa
```

```
4       734      DR-3     1939-01-07      734       pb     ra
5       734      DR-3     1939-01-07      734     lake     sa
6       734      DR-3     1939-01-07      734       pb    tem
7       735      DR-3     1930-01-12      735       pb     ra
8       735      DR-3     1930-01-12      735      NaN     sa
9       735      DR-3     1930-01-12      735      NaN    tem
10      751      DR-3     1930-02-26      751       pb     ra
11      751      DR-3     1930-02-26      751       pb    tem
12      751      DR-3     1930-02-26      751     lake     sa
13      752      DR-3            NaN      752     lake     ra
14      752      DR-3            NaN      752     lake     sa
15      752      DR-3            NaN      752     lake    tem
16      752      DR-3            NaN      752      roe     sa
17      837     MSK-4     1932-01-14      837     lake     ra
18      837     MSK-4     1932-01-14      837     lake     sa
19      837     MSK-4     1932-01-14      837      roe     sa
20      844      DR-1     1932-03-22      844      roe     ra
```

5.3.3 User input values

Missing values could also be created by the user. This can come from creating a vector of values from a calculation or a manually curated vector. To build on the examples from Section 2.4, we can create our own data with missing values. `NaNs` are valid values for Series and `DataFrames`.

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
```

```
amoeba    NaN
goat      4.0
dtype: float64
```

```
# missing value in a dataframe
scientists = pd.DataFrame({

'Name': ['Rosaline Franklin', 'William Gosset'],
'Occupation': ['Chemist', 'Statistician'],
'Born': ['1920-07-25', '1876-06-13'],
'Died': ['1958-04-16', '1937-10-16'],
'missing': [NaN, nan]})

print(scientists)
```

```
         Born        Died                 Name    Occupation  mis
```

```
0  1920-07-25  1958-04-16  Rosaline Franklin       Chemist
1  1876-06-13  1937-10-16     William Gosset  Statistician
```

You can also assign a column of missing values to a dataframe directly.

```
# create a new dataframe
scientists = pd.DataFrame({

'Name': ['Rosaline Franklin', 'William Gosset'],
'Occupation': ['Chemist', 'Statistician'],
'Born': ['1920-07-25', '1876-06-13'],
'Died': ['1958-04-16', '1937-10-16']})

# assign a columns of missing values
scientists['missing'] = nan

print(scientists)
```

```
         Born        Died               Name    Occupation   mis
0  1920-07-25  1958-04-16  Rosaline Franklin       Chemist
1  1876-06-13  1937-10-16     William Gosset  Statistician
```

5.3.4 Re-indexing

Lastly, another way to introduce missing values into your data is to reindex your dataframe. This is useful when you want to add new indicies to your dataframe, but still want to retain its original values. A common useage is when your index represents some time interval, and you want to add more dates.

If we wanted to only look at the years from 2000 to 2010 from the gapminder plot in Section 1.7, we can perform the same grouped operations, subset the data and then re-index it.

```
gapminder = pd.read_csv('../data/gapminder.tsv', sep='\t')
life_exp = gapminder.\
    groupby(['year'])['lifeExp'].\
    mean()

print(life_exp)
```

```
year
1952      49.057620
1957      51.507401
```

```
1962      53.609249
1967      55.678290
1972      57.647386
1977      59.570157
1982      61.533197
1987      63.212613
1992      64.160338
1997      65.014676
2002      65.694923
2007      67.007423
Name:      lifeExp, dtype: float64
```

We can re-index by slicing the data (See Section 1.5)

```
# note you can continue to chain the 'ix' from the code above
print(life_exp.ix[range(2000, 2010), ])
```

```
year
2000            NaN
2001            NaN
2002      65.694923
2003            NaN
2004            NaN
2005            NaN
2006            NaN
2007      67.007423
2008            NaN
2009            NaN
Name:  lifeExp,  dtype: float64
```

Or subset the data separately, and use the reindex method.

```
# subset
y2000 = life_exp[life_exp.index > 2000]
print(y2000)
```

```
year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
# reindex
print(y2000.reindex(range(2000, 2010)))
```

```
year
2000            NaN
2001            NaN
```

```
2002      65.694923
2003            NaN
2004            NaN
2005            NaN
2006            NaN
2007      67.007423
2008            NaN
2009            NaN
Name:   lifeExp, dtype: float64
```

5.4 Working with missing data

Now that we know how missing values can be created, let's see how they behave when working with data.

5.4.1 Find and Count missing data

```
ebola = pd.read_csv('../data/country_timeseries.csv')
```

One way to look at the number of missing values is to count them.

```
# count the number of non-missing values
print(ebola.count())
```

```
Date                                  122
Day                                   122
Cases_Guinea                           93
Cases_Liberia                          83
Cases_SierraLeone                      87
Cases_Nigeria                          38
Cases_Senegal                          25
Cases_UnitedStates                     18
Cases_Spain                            16
Cases_Mali                             12
Deaths_Guinea                          92
Deaths_Liberia                         81
Deaths_SierraLeone                     87
Deaths_Nigeria                         38
Deaths_Senegal                         22
Deaths_UnitedStates                    18
Deaths_Spain                           16
Deaths_Mali                            12
dtype: int64
```

If we wanted, we can subtract the number of non-missing from the total number of rows.

```
num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print(num_missing)

Date                         0
Day                          0
Cases_Guinea                29
Cases_Liberia               39
Cases_SierraLeone           35
Cases_Nigeria               84
Cases_Senegal               97
Cases_UnitedStates         104
Cases_Spain                106
Cases_Mali                 110
Deaths_Guinea               30
Deaths_Liberia              41
Deaths_SierraLeone          35
Deaths_Nigeria              84
Deaths_Senegal             100
Deaths_UnitedStates        104
Deaths_Spain               106
Deaths_Mali                110
dtype: int64
```

If you wanted to count the total number of missing values in your data, or count the number of missing values for a particular columns, you can use the `count_nonzero` function from numpy in conjunction with the isnull method.

```
import numpy as np
print(np.count_nonzero(ebola.isnull()))
1214
print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))
29
```

Another way to get missing data counts is to use the `value_counts` method on a series. This will print a frequency table of values, if you use the `dropna` parameter, you can also get a missing value count.

```
# get the first 5 value counts from the Cases_Guinea column
print(ebola.Cases_Guinea.value_counts(dropna=False).head())

NaN      29
```

```
 86.0      3
 495.0     2
 390.0     2
 112.0     2
Name: Cases_Guinea, dtype: int64
```

5.4.2 Cleaning missing data

5.4.2.1 Recode/Replace

We Can use the fillna method to recode the missing values to another value. For example, if we wanted the missing values to be recoded as a 0.

```
print(ebola.fillna(0).ix[0:10, 0:5])
```

```
              Date      Day   Cases_Guinea   Cases_Liberia  Ca
  0       1/5/2015      289         2776.0             0.0
  1       1/4/2015      288         2775.0             0.0
  2       1/3/2015      287         2769.0          8166.0
  3       1/2/2015      286            0.0          8157.0
  4     12/31/2014      284         2730.0          8115.0
  5     12/28/2014      281         2706.0          8018.0
  6     12/27/2014      280         2695.0             0.0
  7     12/24/2014      277         2630.0          7977.0
  8     12/21/2014      273         2597.0             0.0
  9     12/20/2014      272         2571.0          7862.0
  10    12/18/2014      271            0.0          7830.0
```

You can see if we use fillna , we can recode the values to a specific value. If you look into the documentation, `fillna` , like many other pandas functions, have a parameter for `inplace`. This simply means, the underlying data will be automatically changed without creating a new copy with the changes. This is a parameter you will want to use when your data gets larger and you want to be more memory efficient.

5.4.2.2 Fill Forwards

We can use built-in methods to fill forwards or backwards. When we fill data forwards, it means take the last known value, and use that value for the next missing value. This way, missing values are replaced with the last known/recorded value.

```
print(ebola.fillna(method='ffill').ix[0:10, 0:5])

              Date      Day   Cases_Guinea       Cases_Liberia
0         1/5/2015      289         2776.0                 NaN
1         1/4/2015      288         2775.0                 NaN
2         1/3/2015      287         2769.0              8166.0
3         1/2/2015      286         2769.0              8157.0
4       12/31/2014      284         2730.0              8115.0
5       12/28/2014      281         2706.0              8018.0
6       12/27/2014      280         2695.0              8018.0
7       12/24/2014      277         2630.0              7977.0
8       12/21/2014      273         2597.0              7977.0
9       12/20/2014      272         2571.0              7862.0
10      12/18/2014      271         2571.0              7830.0
```

If a column begins with a missing value, then it will remain missing because there is no previous value to fill in.

5.4.2.3 Fill Backwards

We can also have pandas fill data backwards. When we fill data backwards, the newest value is used to replace missing. This way, missing values are replaced with the newest value.

```
print(ebola.fillna(method='bfill').ix[:, 0:5].tail())

          Date     Day     Cases_Guinea     Cases_Liberia     Ca
117  3/27/2014       5            103.0               8.0
118  3/26/2014       4             86.0               NaN
119  3/25/2014       3             86.0               NaN
120  3/24/2014       2             86.0               NaN
121  3/22/2014       0             49.0               NaN
```

If a column ends with a missing value, then it will remain missing because there is no new value to fill in.

5.4.2.4 interpolate

Interpolation is a small mini chapter on its own (TODO CHAPTER?). The general gist is, you can have pandas use existing values to fill in missing values.

```
print(ebola.interpolate().ix[0:10, 0:5])

            Date    Day   Cases_Guinea     Cases_Liberia
 0      1/5/2015    289         2776.0               NaN
 1      1/4/2015    288         2775.0               NaN
 2      1/3/2015    287         2769.0            8166.0
 3      1/2/2015    286         2749.5            8157.0
 4    12/31/2014    284         2730.0            8115.0
 5    12/28/2014    281         2706.0            8018.0
 6    12/27/2014    280         2695.0            7997.5
 7    12/24/2014    277         2630.0            7977.0
 8    12/21/2014    273         2597.0            7919.5
 9    12/20/2014    272         2571.0            7862.0
 10   12/18/2014    271         2493.5            7830.0
```

The interpolate method has a method parameter that can change the interpolation method.

5.4.2.5 Drop Missing values

The last way to work with missing data is to drop observations or variables with missing data. Depending on how much data is missing, only keeping complete case data can leave you with a useless dataset. Either the missing data is not random, and dropping missing values will leave you with a biased dataset, or keeping only complete data will leave you with not enough data to run your analysis.

We can use the dropna method to drop missing data. There are a few ways we can control how data can be dropped. The *dropna* method has a `how` parameter that lets you specify whether a row (or column) is dropped when ' `any` ' or ' `all` ' the data is missing.

The thresh parameter lets you specify how many non-NA values you have before dropping the row or column.

```
print(ebola.shape)
(122, 18)
```

If we only keep complete cases in our ebola dataset, we are only left with 1 row of data.

```
ebola_dropna = ebola.dropna()
print(ebola_dropna.shape)
(1, 18)
print(ebola_dropna)

             Date  Day       Cases_Guinea      Cases_Liberia
19     11/18/2014  241             2047.0             7082.0

       Cases_Nigeria      Cases_Senegal    Cases_UnitedStates
19              20.0                1.0                   4.0

       Deaths_Guinea       Deaths_Liberia   Deaths_SierraLeone
19            1214.0               2963.0               1267.0

       Deaths_Senegal       Deaths_UnitedStates    aths_Spain
19                0.0                       1.0           0.0
```

5.4.3 Calculations with missing data

Let’s say we wanted to look at the case counts for multiple regions. We can add multiple regions together to get a new columns of case counts.

```
ebola['Cases_multiple'] = ebola['Cases_Guinea'] + \
                          ebola['Cases_Liberia'] + \
                          ebola['Cases_SierraLeone']
```

We can look at the results by looking at the first 10 lines of the calculation.

```
ebola_subset = ebola.ix[:, ['Cases_Guinea', 'Cases_Liberia',
                            'Cases_SierraLeone', 'Cases_multip

print(ebola_subset.head(n=10))

      Cases_Guinea    Cases_Liberia    Cases_SierraLeone      (
  0         2776.0              NaN              10030.0
  1         2775.0              NaN               9780.0
  2         2769.0           8166.0               9722.0
  3            NaN           8157.0                  NaN
  4         2730.0           8115.0               9633.0
  5         2706.0           8018.0               9446.0
  6         2695.0              NaN               9409.0
  7         2630.0           7977.0               9203.0
  8         2597.0              NaN               9004.0
  9         2571.0           7862.0               8939.0
```

You can see that the only times a value for `Cases_multiple` was calculated, was when there was no missing value for `Cases_Guinea`, `Cases_Liberia` , and `Cases_SierraLeone`. Calculations with missing values will typically return a missing value, unless the function or method called has a means to ignore missing values in its calculations.

An example of a built-in method that can ignore missing values is mean or sum. These functions will typically have a skipna parameter that will still calculate a value by skipping over the missing values.

```
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))
```

```
 84729.0
```

```
print(ebola.Cases_Guinea.sum(skipna = False))
```

```
 nan
```

Summary

It is rare to have a dataset without any missing values. It is important to know how to work with missing values because even when you are working with data that is complete, missing values can still arise from your own data munging. Here I began some of the basic methods of the data analysis process that pertains to data validity. By looking at your data, and tabulating missing values, you can start the process of assessing if the data you are given is of enough quality for making decisions and inferences from your data.

Chapter 6. Tidy Data by Reshaping

6.1 Introduction

Hadley Wickham [1], one of the more prominent members in the R community, talks about *tidy* data in a paper[2] in the *Journal of Statistical Software*. Tidy data is a framework to structure datasets so they can be easily analyzed and visualized. It can be thought of as a goal one should aim for when cleaning data. Once you understand what tidy data is, it will make your data analysis, visualization, and collection much easier.

What is *tidy* data? Hadley Wickham's paper defines it as such:

• each row is an observation

• each column is a variable

• each type of observational unit forms a table

This chapter will go through the various ways to tidy data from the *Tidy Data* paper.

Concept Map

Prior knowledge:

1. function and method calls

2. subsetting data

3. loops

4. list comprehension

This Chapter:

• reshaping data

1. unpivot/melt/gather

2. pivot/cast/spread

3. subsetting

4. combining

(a) globbing

(b) concatenation

[1] http://hadley.nz/

[2] http://vita.had.co.nz/papers/tidy-data.pdf

Objectives

This chapter will cover:

1. unpivot/melt/gather columns into rows

2. pivot/cast/spread rows into columns

3. normalize data by separating a dataframe into multiple tables

4. assembling data from multiple parts

6.2 Columns contain values, not variables

Data can have columns that contain values instead of variables. This is usually a convenient format for data collection and presentation.

6.2.1 Keep 1 column fixed

We can use the data on income and religion in the United States from the Pew Research Center to illustrate this example.

```
import pandas as pd
pew = pd.read_csv('../data/tidy-data/data/pew_raw.csv')
```

If we look at the data, we can see that not every column is a variable. The values that relate to income are spread across multiple columns. The format shown is great when presenting data in a table, but for data analytics, the table needs to be reshaped such that we have a religion, income, and count variables.

```
# only show the first few columns
print(pew.ix[:, 0:6])
                        religion        <$10k    $10-20k     $20-
 0                      Agnostic           27         34
 1                       Atheist           12         27
 2                      Buddhist           27         21
 3                      Catholic          418        617
 4            Dont know/refused          15         14
 5              Evangelical Prot          575        869
 6                         Hindu            1          9
 7     Historically Black Prot          228        244
 8            Jehovah's Witness           20         27
 9                        Jewish           19         19
 10                 Mainline Prot          289        495
 11                        Mormon           29         40
 12                        Muslim            6          7
 13                     Orthodox           13         17
 14              Other Christian            9          7
 15                 Other Faiths           20         33
 16        Other World Religions            5          2
 17                 Unaffiliated          217        299
```

This view of the data is also known as 'wide' data. In order to turn it into the 'long' tidy data format, we will have to unpivot/melt/gather (depending on which statistical programming language you use) our dataframe.

Pandas has a function called `melt` that will reshape the dataframe into a tidy format. `melt` takes a few parameters:

- `id_vars` is a container (list, tuple, ndarray) that represents the variables that will remain as-is

• `value_vars` are the columns you want to melt down (or unpivot) By default it will melt all the columns not specified in the `id_vars` parameter

• `var_name` is a string for the new column name when the `value_vars` is melted down. By defualt it will be called `variable`

• `value_name` is a string for the new column name that represents the values for the `var_name`. By default it will be called `value`

```
#  we do not need to specify a value_vars since we want to piv
#  all the columns except for the 'religion' column
pew_long = pd.melt(pew, id_vars='religion')

print(pew_long.head())

             religion variable  value
  0          Agnostic    <$10k     27
  1           Atheist    <$10k     12
  2          Buddhist    <$10k     27
  3          Catholic    <$10k    418
  4  Dont know/refused    <$10k    15

print(pew_long.tail())

                   religion                variable  value
  175              Orthodox  Don't know/refused     73
  176       Other Christian  Don't know/refused     18
  177          Other Faiths  Don't know/refused     71
  178  Other World Religions  Don't know/refused      8
  179          Unaffiliated  Don't know/refused    597
```

We can change the defaults so that the melted/unpivoted columns are named.

```
pew_long = pd.melt(pew,
                   id_vars='religion',
                   var_name='income',
                   value_name='count')

print(pew_long.head())

              religion income  count
 0            Agnostic <$10k     27
 1             Atheist <$10k     12
 2            Buddhist <$10k     27
 3            Catholic <$10k    418
```

```
4  Dont know/refused <$10k      15
```

```
print(pew_long.tail())
                  religion                     income  count
175              Orthodox     Don't  know/refused     73
176       Other Christian     Don't  know/refused     18
177          Other Faiths     Don't  know/refused     71
178  Other World Religions     Don't  know/refused      8
179          Unaffiliated     Don't  know/refused    597
```

6.2.2 Keep multiple columns fixed

Not every dataset will have one column to hold still while you unpivot the rest. If you look at the Billboard dataset:

```
billboard = pd.read_csv('../data/tidy-data/data/billboard-raw.c

# look at the first few rows and columns
print(billboard.ix[0:5, 0:7])

    year        artist                        track  time date.ente
 0  2000        2Ge+her  The Hardest Part Of ...  3:15   2000-09
 1  2000          2 Pac           Baby Don't Cry  4:22   2000-02
 2  2000  3 Doors Down               Kryptonite  3:53   2000-04
 3  2000  3 Doors Down                    Loser  4:24   2000-10
 4  2000      504 Boyz            Wobble Wobble  3:35   2000-04
 5  2000           98?  Give Me Just One Nig...  3:24   2000-08
```

You can see here that each week is it's own column. Again, there is nothing nothing *wrong* with this form of data. It maybe easy to enter the data in this form, and it is much quicker to understand when presented in a table. However, there may be a time when you will need to melt the data. An example would be when plotting weekly ratings in a faceted plot, since the facet variable needs to be a columns in the dataframe.

```
billboard_long = pd.melt(
    billboard,
    id_vars=['year', 'artist', 'track', 'time', 'date.entered']
    var_name='week',
    value_name='rating')

print(billboard_long.head())
             year        artist                          track   time
```

```
   0  2000       2Ge+her  The Hardest Part Of ...   3:15
   1  2000          2 Pac          Baby Don't Cry   4:22
   2  2000  3 Doors Down              Kryptonite   3:53
   3  2000  3 Doors Down                   Loser   4:24
   4  2000      504 Boyz           Wobble Wobble   3:35
```

```
print(billboard_long.tail())
       year            artist                    track   time da
24087  2000     Wright, Chely                   It Was   3:51
24088  2000       Yankee Grey    Another Nine Minutes   3:10
24089  2000  Yearwood, Trisha          Real Live Woman   3:55
24090  2000   Ying Yang Twins Whistle While You Tw...   4:19
24091  2000     Zombie Nation            Kernkraft 400   3:30
```

6.3 Columns contain multiple variables

There will be times when the columns represent multiple variables. This is something that is common when working with health data. To illustrate this, let’s look at the Ebola dataset.

```
ebola = pd.read_csv('../data/ebola_country_timeseries.csv')

print(ebola.columns)
          Index(['Date', 'Day', 'Cases_Guinea', 'Cases_Liberia'
          'Cases_SierraLeone',
                 'Cases_Nigeria', 'Cases_Senegal', 'Cases_Unite
          'Cases_Spain',
                 'Cases_Mali', 'Deaths_Guinea', 'Deaths_Liberia
          'Deaths_SierraLeone',
                 'Deaths_Nigeria', 'Deaths_Senegal', 'Deaths_Un
                 'Deaths_Spain', 'Deaths_Mali'],
                dtype='object')

# print select rows
print(ebola.ix[:5, [0, 1, 2, 3, 10, 11]])
         Date  Day   Cases_Guinea  Cases_Liberia  Deaths_Guinea
0    1/5/2015  289         2776.0            NaN         1786.0
1    1/4/2015  288         2775.0            NaN         1781.0
2    1/3/2015  287         2769.0         8166.0         1767.0
3    1/2/2015  286            NaN         8157.0            NaN
4  12/31/2014  284         2730.0         8115.0         1739.0
5  12/28/2014  281         2706.0         8018.0         1708.0
```

The column names `Cases_Guinea` and `Deaths_Guinea` actually contain 2 variables. The individual status, cases and deaths, and the county, Guinea. The data is also in wide format that needs to be unpivoted.

```
ebola_long = pd.melt(ebola, id_vars=['Date', 'Day'])

print(ebola_long.head())

            Date  Day       variable       value
 0      1/5/2015  289   Cases_Guinea      2776.0
 1      1/4/2015  288   Cases_Guinea      2775.0
 2      1/3/2015  287   Cases_Guinea      2769.0
 3      1/2/2015  286   Cases_Guinea         NaN
 4    12/31/2014  284   Cases_Guinea      2730.0

print(ebola_long.tail())

              Date  Day       variable     value
 1947     3/27/2014  5    Deaths_Mali       NaN
 1948     3/26/2014  4    Deaths_Mali       NaN
 1949     3/25/2014  3    Deaths_Mali       NaN
 1950     3/24/2014  2    Deaths_Mali       NaN
 1951     3/22/2014  0    Deaths_Mali       NaN
```

6.3.1 Split and add columns individually (simple method)

Conceptually, the column of interest can be split by the underscore (_)). The first part will be the new `status` column, and the second part will be the new `country` column. This will require some string parsing and splitting in Python. In Python, a string is an object, similar to how `Pandas` has a `Series` and `DataFrame` object. Chapter ?? showed how `Series` can have various methods, such as `mean`, and `DataFrames` have methods such as `to_csv`. Strings have methods as well, in this case we will use the `split` method that takes a string and will split the string up by a given delimiter. By default `split` will split the string by a space, but we can pass in the underscore, , in our example. In order to get access to the string methods, we need to use the `str` attribute.

```
# get the variable column
# access the string methods
# and split the column by a delimiter
variable_split = ebola_long.variable.str.split('_')

print(variable_split[:5])                        print(variable_sp
```

```
0       [Cases, Guinea]                          1947        [Deat
1       [Cases, Guinea]                          1948        [Deat
2       [Cases, Guinea]                          1949        [Deat
3       [Cases, Guinea]                          1950        [Deat
4       [Cases, Guinea]                          1951        [Deat
Name: variable, dtype: object                    Name: variable,
```

We can see that after we `split` on the underscore, the values are returned in a list. We know it's a list because that's how the `split` method works[3], but the visual cue is that the results are surrounded by square brackets.

[3] https://docs.python.org/2/library/stdtypes.html#str.split

```
# the entire container
print(type(variable_split))

class 'pandas.core.series.Series'>

# the first element in the container
print(type(variable_split[0]))

class 'list'>
```

Now that we have column split into the various pieces, the next step is to assign them to a new column. But first, we need to extract all the 0 index elements for the `status` column and the 1 index elements for the `country` column. To do so, we need to access the string methods again, and then use the `get` method to get the index we want for each row.

```
status_values = variable_split.str.get(0)
country_values = variable_split.str.get(1)
```

```
print(status_values[:5])                    print(status

0     Cases                                 1947     Deat
1     Cases                                 1948     Deat
2     Cases                                 1949     Deat
3     Cases                                 1950     Deat
4     Cases                                 1951     Deat
Name: variable, dtype: object               Name: variak

print(status_values[:5])                    print(status
```

```
0    Guinea                                              1947    Mal:
1    Guinea                                              1948    Mal:
2    Guinea                                              1949    Mal:
3    Guinea                                              1950    Mal:
4    Guinea                                              1951    Mal:
Name: variable, dtype: object                            Name: variak
```

Now that we have the vectors we want, we can add them to our dataframe

```
ebola_long['status'] = status_values
ebola_long['country'] = country_values

print(ebola_long.head())
             Date  Day      variable   value status country
  0      1/5/2015  289   Cases_Guinea  2776.0  Cases Guinea
  1      1/4/2015  288   Cases_Guinea  2775.0  Cases Guinea
  2      1/3/2015  287   Cases_Guinea  2769.0  Cases Guinea
  3      1/2/2015  286   Cases_Guinea     NaN  Cases Guinea
  4    12/31/2014  284   Cases_Guinea  2730.0  Cases Guinea
```

6.3.2 Split and combine in a single step (simple method)

We can do the same thing as before, and exploit the fact that the vector returned is in the same order as our data. We can concatenate (Chapter 4) the new vector or our original data.

```
variable_split = ebola_long.variable.str.split('_',  expand=T
variable_split.columns =  ['status',  'country']
ebola_parsed = pd.concat([ebola_long,  variable_split],  axis=

print(ebola_parsed.head())

          Date  Day     variable   value status country
 0    1/5/2015  289  Cases_Guinea  2776.0 Cases Guinea
 1    1/4/2015  288  Cases_Guinea  2775.0 Cases Guinea
 2    1/3/2015  287  Cases_Guinea  2769.0 Cases Guinea
 3    1/2/2015  286  Cases_Guinea     NaN Cases Guinea
 4  12/31/2014  284  Cases_Guinea  2730.0 Cases Guinea

print(ebola_parsed.tail())

            Date   Day     variable  value  status country
 1947  3/27/2014     5  Deaths_Mali    NaN  Deaths    Mali
 1948  3/26/2014     4  Deaths_Mali    NaN  Deaths    Mali
 1949  3/25/2014     3  Deaths_Mali    NaN  Deaths    Mali
```

```
 1950  3/24/2014     2   Deaths_Mali    NaN  Deaths    Mali
 1951  3/22/2014     0   Deaths_Mali    NaN  Deaths    Mali
```

6.3.3 Split and combine in a single step (more complicated method)

We can accomplish the same result in a single step by taking advantage of the fact that the split results return a list of 2 elements, where each element will be a new column. We can combine the list of split items with the built-in `zip` function (TODO APPENDIX).

`zip` takes a set of iterators (lists, tuples, etc.) and creates a new container that is made of the input iterators, but each new container created is the same index from the input containers.

For example, if we have 2 lists of values:

```
constants =   ['pi',   'e']
values =   ['3.14',   '2.718']
```

we can `zip` the values together as such:

```
# we have to call list on the zip function
# to show the contents of the zip object
# this is because in Python 3 zip returns an iterator.
print(list(zip(constants,  values)))
```

```
 [('pi',   '3.14'),   ('e',   '2.718')]
```

Each element now has the constant matched with its corresponding value. Conceptually, each container is like a side of a zipper. When we `zip` the containers, the indices are matched up and returned.

Another way to visualize what `zip` is doing is taking each container passed into zip and stacking them on top of each other (think row wise concatenation in Section 4.4.1) creating a dataframe of sorts. `zip` then returns the values column-by-column in a tuple.

We can use the same `ebolaJong . variable . str. split (' _')` to split the values in the column. However, since the result is already a container (a `Series` object), we need to unpack it such that it is the contents of the container (each status-country list) not the container itself (the series)

The asterisk, *, in python is used to unpack containers[4]. When we zip the unpacked containers, it is the same as creating the `status_values` and `country .values` above. We can then assign the vectors to the columns simultaneously using multiple assignment (TODO APPENDIX MULTIPLE ASSIGNMENT).

```
# note we can also use:
# ebola_long['status'],   ebola_long['country']  =
zip(*ebola_long['variable']str.split('_'))
ebola_long['status'],   ebola_long['country']   =
zip(*ebola_long.variable.str.split('_'))

print(ebola_long head())

         Date Day        variable     value status country
 0   1/5/2015 289   Cases_Guinea    2776.0  Cases  Guinea
 1   1/4/2015 288   Cases_Guinea    2775.0  Cases  Guinea
 2   1/3/2015 287   Cases_Guinea    2769.0  Cases  Guinea
 3   1/2/2015 286   Cases_Guinea       NaN  Cases  Guinea
 4 12/31/2014 284   Cases_Guinea    2730.0  Cases  Guinea
```

6.4 Variables in both rows and columns

At times data will be in a shape where variables are in both rows and columns. That is, some combination of the previous sections of this chapter. Most of the methods to tidy up the data have already been presented. What is left to show is what happens if a column of data actually holds 2 variables instead of 1. In this case, we will have to pivot or cast the variable into separate columns.

[4] https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists

```
weather = pd.read_csv('../data/tidy-data/data/weather-raw.csv')
print(weather.ix[:5,   :12])
            id     year    month element d1            d2       d3
 0     MX17004     2010        1     tmax NaN          NaN      NaN
 1     MX17004     2010        1     tmin NaN          NaN      NaN
 2     MX17004     2010        2     tmax NaN    27.3     24.1     Na
 3     MX17004     2010        2     tmin NaN    14.4     14.4     Na
```

```
4      MX17004    2010          3     tmax NaN     NaN      NaN    Na
5      MX17004    2010          3     tmin NaN     NaN      NaN    Na
```

In the weather data, there are minimum and maximum (`tmin` and `tmax` values in the `element` column, respectively) temperatures recorded for each day `(d1, d2, d31)` of the month `(month)`. The `element` column contains variables that need to be casted/pivoted to become new columns, and the day variables, need to be melted into row vales. Again, there is nothing wrong with the data in the current format. It is simply not in a shape for analysis, but can be helpful when presenting data in reports.

Let's first melt/unpivot the day values

```
weather_melt = pd.melt(weather,
                       id_vars=['id',   'year',   'month',   'e
                       var_name = 'day' ,
                       value_name='temp')

print(weather_melt.head())

          id  year  month element  day   temp
  0  MX17004  2010      1    tmax   d1    NaN
  1  MX17004  2010      1    tmin   d1    NaN
  2  MX17004  2010      2    tmax   d1    NaN
  3  MX17004  2010      2    tmin   d1    NaN
  4  MX17004  2010      3    tmax   d1    NaN

print(weather_melt.tail())

             id  year   month element   day   temp
  677   MX17004  2010     10     tmin   d31    NaN
  678   MX17004  2010     11     tmax   d31    NaN
  679   MX17004  2010     11     tmin   d31    NaN
  680   MX17004  2010     12     tmax   d31    NaN
  681   MX17004  2010     12     tmin   d31    NaN
```

The next, we need to pivot up the variables stored in the `element` column. This is also refereed to as casting or spreading in other statistical languages.

One of the main differences from `pivot_table` and `melt`, is that `melt` is a function within `pands` and `pivot_table` is a method we call on a `DataFrame` object.

```
weather_tidy = weather_melt.pivot_table(
    index=['id',   'year',   'month',   'day'],
    columns = 'element' ,
    values='temp'
```

If we look at the pivoted table, we will notice that each value in the `element` column is now a separate column. We can leave it in its current state, but we can also flatten the hierarchical columns

```
weather_tidy_flat = weather_tidy.reset_index()

print(weather_tidy_flat head())

 element            id      year  month   day   tmax  tmin
 0             MX17004      2010      1    d1    NaN   NaN
 1             MX17004      2010      1   d10    NaN   NaN
 2             MX17004      2010      1   d11    NaN   NaN
 3             MX17004      2010      1   d12    NaN   NaN
 4             MX17004      2010      1   d13    NaN   NaN
```

likewise, we can perform those methods without the intermediate dataframe as such:

```
weather_tidy = weather_melt \
    pivot_table(
        index=['id',   'year',   'month',   'day'],
        columns='element',
        values='temp').\
reset_index()

print(weather_tidy head())

 element            id      year    month      day   tmax    tmin
 0             MX17004      2010        1       d1    NaN     NaN
 1             MX17004      2010        1      d10    NaN     NaN
 2             MX17004      2010        1      d11    NaN     NaN
 3             MX17004      2010        1      d12    NaN     NaN
 4             MX17004      2010        1      d13    NaN     NaN
```

6.5 Multiple Observational Units in a table (Normalization)

One of the simplest ways of knowing if multiple observational units are represented in a table is by looking at each of the rows, and taking note of any cells or values that are being repeated from row to row. This is very common in government education administration data where student demographics are reported for each student for each year the student is enrolled.

If we look at the billboard data we cleaned in Section 6.2.2:

```
print(billboard_long head())
       year          artist                        track
 0     2000         2Ge+her       The Hardest Part Of ...
 1     2000           2 Pac                Baby Don't Cry
 2     2000    3 Doors Down                    Kryptonite
 3     2000    3 Doors Down                         Loser
 4     2000        504 Boyz                 Wobble Wobble
```

and if we subset (Section 2.6.1) on a particular track:

```
print(billboard_long[billboard_long.track == 'Loser'].head())

       year            artist  track    time date.entered week
 3     2000    3   Doors Down  Loser    4:24   2000-10-21  wk1
 320   2000    3   Doors Down  Loser    4:24   2000-10-21  wk2
 637   2000    3   Doors Down  Loser    4:24   2000-10-21  wk3
 954   2000    3   Doors Down  Loser    4:24   2000-10-21  wk4
 1271  2000    3   Doors Down  Loser    4:24   2000-10-21  wk5
```

We can see that this table actually holds 2 types of data: the track information and weekly ranking. It would be better to store the track information in a separate table. This way, the information stored in the `year`, `artist`, `track`, and `time` columns are not repeated in the dataset. This is particularly important if the data is manually entered. By repeating the same values over and over during data entry, one risks having inconsistent data.

What we should do in this case is to have the `year`, `artist`, `track`, `time`, and `date.entered` in a new dataframe and each unique set of values be assigned a unique ID. We can then use this unique ID in a second dataframe that represents a song, date, week number, and ranking. This entire process can be thought of as reversing the steps in concatenating and merging data in Chapter 4.

```
billboard_songs = billboard_long[['year',   'artist',   'track
print(billboard_songs.shape)
```

```
(24092,   4)
```

We know there are duplicate entries in this dataframe, so we need to drop the duplicate rows.

```
billboard_songs = billboard_songs.drop_duplicates() print(billk
(317,   4)
```

We can then assign a unique value to each row of data.

```
billboard_songs['id']   = range(len(billboard_songs))
print(billboard_songs.head(n=10))
```

```
   year           artist                     track      time
0  2000          2Ge+her   The Hardest Part Of ...      3:15
1  2000            2 Pac            Baby Don't Cry      4:22
2  2000     3 Doors Down                Kryptonite      3:53
3  2000     3 Doors Down                     Loser      4:24
4  2000         504 Boyz             Wobble Wobble      3:35
5  2000              98?   Give Me Just One Nig...      3:24
6  2000          Aaliyah             I Don't Wanna      4:15
7  2000          Aaliyah                 Try Again      4:03
8  2000   Adams, Yolanda             Open My Heart      5:30
9  2000    Adkins, Trace                      More      3:05
```

Now that we have a separate dataframe about songs, we can use the newly created `id` column to match a song to its weekly ranking.

```
# Merge the song dataframe to the original dataset
billboard_ratings = billboard_long.merge(billboard_songs,   on=
'artist', 'track', 'time'])
print(billboard_ratings shape)
```

```
(24092,   8)
```

```
print(billboard_ratings head())
```

```
   year  artist                           track   time date.ente
0  2000  2Ge+her  The   Hardest  Part  Of ...   3:15   2000-09
1  2000  2Ge+her  The   Hardest  Part  Of ...   3:15   2000-09
2  2000  2Ge+her  The   Hardest  Part  Of ...   3:15   2000-09
3  2000  2Ge+her  The   Hardest  Part  Of ...   3:15   2000-09
4  2000  2Ge+her  The   Hardest  Part  Of ...   3:15   2000-09
```

Finally, we subset the columns to the ones we want in our ratings dataframe.

```
billboard_ratings = billboard_ratings[['id', 'date.entered', 'w
print(billboard_ratings head())

    id date.entered week   rating
 0   0    2000-09-02 wk1     91.0
 1   0    2000-09-02 wk2     87.0
 2   0    2000-09-02 wk3     92.0
 3   0    2000-09-02 wk4      NaN
 4   0    2000-09-02 wk5      NaN
```

6.6 Observational units across multiple tables

The last bit of data tidying involves having the same type of data being spread across multiple datasets. This has already been covered in Chapter 4 when we discussed data concatenation and merging. A reason why data would be split across multiple files would be size. By splitting up data into various parts, each part would be smaller. This may be good to share data on the Internet or email since many services limit the size of a file that can be opened or shared. Another reason why a dataset would be split into multiple parts would be from the data collection process. For example, a separate data containing stock information could be created for each day.

I've already covered how to merge and concatenate data, but here I will show you ways we can quickly load multiple data sources and assemble them together.

The Unified New York City Taxi and Uber Data is a good example to show this. The entire dataset has over 1.3 billion taxi and Uber trips from New York City, and has over 140 files.

Here for illustration purposes, we only work with 5 of these data files. When the same data is broken into multiple parts, they typically have a structured naming pattern associated with it.

In the NYC Taxi example, all of the raw taxi trips have the pattern `fhv_tripdata_YYYY_XX.csv`, where `YYYY` represents the year (e.g., 2015), and `XX` represents the part number. We can use the a simple pattern matching function from the `glob` library in Python to get a list of all the filenames that match a particular pattern.

```
import glob

# get a list of the csv files from the nyc-taxi data folder
nyc_taxi_data = glob.glob('../data/nyc-taxi/*.csv')
print(nyc_taxi_data)
```

```
['../data/nyc-taxi/fhv_tripdata_2015-03.csv', '../data/nyc-
taxi/fhv_tripdata_2015-02.csv', '../data/nyc-
taxi/fhv_tripdata_2015-04.csv', '../data/nyc-
taxi/fhv_tripdata_2015-05.csv', '../data/nyc-
taxi/fhv_tripdata_2015-01.csv']
```

Now that we have a list of filenames we want to load, we can load each file into a dataframe.

We can choose to load each file individually like we have been doing so far.

```
taxi1 = pd.read_csv(nyc_taxi_data[0])
taxi2 = pd.read_csv(nyc_taxi_data[1])
taxi3 = pd.read_csv(nyc_taxi_data[2])
taxi4 = pd.read_csv(nyc_taxi_data[3])
taxi5 = pd.read_csv(nyc_taxi_data[4])
```

We can look at our data and see how they can be nicely stacked (concatenated) on top of each other.

```
print(taxi1.head(n=2))
print(taxi2.head(n=2))
print(taxi3.head(n=2))
print(taxi4.head(n=2))
print(taxi5.head(n=2))
     Dispatching_base_num          Pickup_date  locationID
 0                 B00029  2015-03-01 00:02:00       213.0
 1                 B00029  2015-03-01 00:03:00        51.0
     Dispatching_base_num          Pickup_date  locationID
 0                 B00013  2015-02-01 00:00:00         NaN
 1                 B00013  2015-02-01 00:01:00         NaN
     Dispatching_base_num          Pickup_date  locationID
 0                 B00001  2015-04-01 04:30:00         NaN
 1                 B00001  2015-04-01 06:00:00         NaN
     Dispatching_base_num          Pickup_date  locationID
 0                 B00001  2015-05-01 04:30:00         NaN
 1                 B00001  2015-05-01 05:00:00         NaN
     Dispatching_base_num          Pickup_date  locationID
 0                 B00013  2015-01-01 00:30:00         NaN
```

```
 1                    B00013  2015-01-01 01:22:00         NaN
```

We can concatenate them just like in Chapter 4.

```
# shape of each dataframe
print(taxi1 shape)
print(taxi2 shape)
print(taxi3 shape)
print(taxi4 shape)
print(taxi5 shape)

(3281427, 3)
(3126401, 3)
(3917789, 3)
(4296067, 3)
(2746033, 3)

# concatenate the dataframes together
taxi = pd.concat([taxi1,  taxi2,  taxi3,  taxi4,  taxi5])

# shape of final concatenated taxi data
print(taxi shape)

(17367717, 3)
```

However, manually saving each dataframe will get tedious when there are many parts the data is split into. Instead we can automate the process using loops and list comprehensions

6.6.1 Load multiple files using a loop

The easier way is to first create an empty list, use a loop to iterate though each of the csv files, load the csv file into a pandas dataframe, and finally append the dataframe to the list.

The final type of data we want is a list of dataframes because the `concat` function takes a list of dataframes to concatenate.

```
# create an empty list  to append to list_taxi_df =   []
# loop though each csv filename
for csv_filename in nyc_taxi_data:
    # you can choose to print  the filename for debugging
```

```
    # print(csv_filename)

        # load the csv file into a dataframe
        df = pd.read_csv(csv_filename)

        # append the dataframe to the list  that will hold the d
        list_taxi_df append(df)

# print  the length of the dataframe
print(len(list_taxi_df))
# type of the first element
print(type(list_taxi_df[0]))
<class  'pandas.core.frame.DataFrame'>
# look at  the head of the first dataframe
print(list_taxi_df[0].head())
      Dispatching_base_num            Pickup_date  locationID
 0                  B00029   2015-03-01 00:02:00       213.0
 1                  B00029   2015-03-01 00:03:00        51.0
 2                  B00029   2015-03-01 00:11:00         3.0
 3                  B00029   2015-03-01 00:11:00       259.0
 4                  B00029   2015-03-01 00:13:00       174.0
```

Now that we have a list of dataframes, we can concatentate them.

```
taxi_loop_concat = pd.concat(list_taxi_df)
print(taxi_loop_concat shape)
 (17367717,   3)
# Did we get the same results as the manual laod and concatena
print(taxi.equals(taxi_loop_concat))
 True
```

6.6.2 Load multiple files using a list comprehension

Python has an idiom for looping though something and adding it to a list. It is called a list comprehension.

The loop above which, I will show again without the comments, can be written in a list comprehension (TODO APPENDIX).

```
# the loop code without comments

list_taxi_df =   []
for csv_filename in nyc_taxi_data:
    df = pd.read_csv(csv_filename)
```

```
    list_taxi_df append(df)

# same code in a list comprehension
list_taxi_df_comp =   [pd.read_csv(csv_filename)   for csv_file
```

The result from our list comprehension is a list, just like the loop example above.

```
print(type(list_taxi_df_comp))
 <class  'list'>
```

Finally, we can concatenate the results just like before.

```
taxi_loop_concat_comp = pd.concat(list_taxi_df_comp)

# are the concatenated dataframes the same?
print(taxi_loop_concat_comp equals(taxi_loop_concat))
 True
```

6.7 Summary

Here I showed you how we can reshape data to a format that is conducive for data analysis, visualization, and collection. We followed Hadley Wickham's *Tidy Data* paper to show the various functions and methods to reshape our data. This is an important skill since various functions will need data in a certain shape, tidy or not, in order to work. Knowing how to reshape your data will be an important still as a data scientist and analyst.