

Understanding and Visualizing Data with Python

Nina Dombrowski

Table of contents

2	Lecture notes	6
2.1	Definitions	6
2.2	Standard Score (Empirical Rule)	6
2.3	Multivariate data	7
2.3.1	Scatter plots	7
2.3.2	Outliers	8
2.4	Simpson's Paradox	8
2.5	Populations and samples	8
2.5.1	Sampling from populations intro	8
2.5.2	Simple random sampling (SRS)	9
2.5.3	Working with complex samples	10
3	What is Jupyter Notebooks?	11
3.0.1	Jupyter Notebook Features	11
3.0.2	What is Markdown?	11
4	H1	12
4.1	H2	12
4.1.1	H3	12
4.1.2	Kernels, Variables, and Environment	13
4.1.3	Command vs. Edit Mode & Shortcuts	14
4.1.4	How do you install Jupyter Notebooks?	16
5	Data Types in Python	17
5.0.1	Numerical or Quantitative (taking the mean makes sense)	17
5.0.2	Categorical or Qualitative	19
6	Python Libraries	24
7	Documentation	25
7.0.1	Importing Libraries	25
7.0.2	Utilizing Library Functions	25

8	Data Management	27
8.0.1	Importing Data	27
8.0.2	Viewing Data	28
8.0.3	.loc()	29
8.0.4	.iloc()	33
8.1	Explore what datatypes we work with using dtypes	33
8.2	Print unique values	34
8.3	Summarizing multiple columns using groupby	35
9	Using Python to read data files and explore their contents	37
9.0.1	Importing libraries	37
9.0.2	Reading a data file	38
9.0.3	Exploring the contents of a data set	39
9.0.4	Slicing a data set	41
9.0.5	Missing values	43
10	Python Resources	44
10.0.1	The Python Documentation	44
10.0.2	Python Programming Introductions	44
10.0.3	Cheatsheets and References	45
10.0.4	Python Style Guides	45
11	Python Libraries	51
11.1	NumPy	51
11.1.1	Numpy Array	51
11.1.2	Array Indexing	55
11.1.3	Datatypes in Arrays	56
11.1.4	Array Math	57
11.1.5	Descriptive statistics with numpy	59
11.2	SciPy	60
11.2.1	SciPy.Stats	61
11.3	Matplotlib	64
11.4	Seaborn	67
12	Visualizing Data in Python	78
13	Univariate data analyses - NHANES case study	93
13.0.1	Frequency tables	94
13.0.2	Numerical summaries	97
13.0.3	Graphical summaries	99
13.0.4	Stratification	102

14 Practice notebook for univariate analysis using NHANES data	109
14.1 Question 1	110
14.2 Question 2	113
14.3 Question 3	114
14.4 Question 4	118
14.5 Question 5	121
14.6 Question 6	122
15 How to select dataframe subsets from multivariate data	125
15.0.1 Selecting columns	125
15.0.2 Selection by conditions	128
15.0.3 Common errors and how to read them	131
15.0.4 Problem	131
15.0.5 Problem	132
15.0.6 Problem	132
15.0.7 Change values inside a df	133
16 Plot Multivariate Distributions in Python	136
17 Unit Testing	140
17.0.1 Goal	140
18 Analysis of multivariate data - NHANES case study	144
18.0.1 Quantitative bivariate data	145
18.0.2 Heterogeneity and stratification	151
18.0.3 Categorical bivariate data	153
18.0.4 Create count table with pd.crosstab	154
18.0.5 Mixed categorical and quantitative data	158
19 Practice notebook for multivariate analysis using NHANES data	160
19.1 Question 1	161
19.2 Question 2	163
19.3 Question 3	163
19.4 Question 4	164
19.5 Question 5	165

1

2 Lecture notes

A great resource that you can explore is the [This is Statistics website](#), created by the American Statistical Association. This insightful and motivating campaign has countless links, videos, and resources to raise awareness of the wide variety of fascinating careers within statistics.

2.1 Definitions

- The **mean (average)** of a data set is found by adding all numbers in the data set and then dividing by the number of values in the set. Its highly affected by outliers.
- The **median** is the middle value when a data set is ordered from least to greatest.
- The mode is the number that occurs most often in a data set.
- I.i.d. data = Independent and identically distributed data. Here each random variable has the same probability distribution as the others and all are mutually independent
- **Range**: the difference between the highest and lowest values.
- **Interquartile range**: the range of the middle half of a distribution. Q3-Q1
- **Standard deviation**: average distance from the mean.

2.2 Standard Score (Empirical Rule)

In statistics, the **standard score** is the number of standard deviations by which the value of a raw score (i.e., an observed value or data point) is above or below the mean value of what is being observed or measured.

It is calculated by subtracting the population mean from an individual raw score and then dividing the difference by the population standard deviation.

Standard scores are most commonly called **z-scores**.

A bell-shaped or normal distributions is sometimes referred to as the **68-95-99.7 rule**: 68% of the observations are within 1 standard deviation of the mean. 95% of the observations are within 2 standard deviation of the mean. 99.7% of the observations are within 3 standard deviation of the mean.

2.3 Multivariate data

Univariate statistics summarize only one variable at a time. Bivariate statistics compare two variables. Multivariate statistics compare more than two variables.

A **mosaic plot** is a special type of stacked bar chart that shows percentages of data in groups. The plot is a graphical representation of a contingency table. These plots are a way to display qualitative multivariate data.

2.3.1 Scatter plots

Scatter plots are a good way to display multivariate quantitative data

Looking at association in scatter plots:

- Linear: The pattern in a plot is a line
- Quadratic association: The pattern is parabolic, i.e. the pattern goes up at the beginning and goes back down latter
- No association: there is no pattern

Looking at the direction of the association:

- Positive linear association: the pattern has a positive slope, i.e. when x increases, y decreases
- Negative linear association

Looking at the strength of the association:

- Weak linear association: the points are largely scattered along the line
- Moderate linear association: Partial scattering of points
- Strong linear association: Points are minimally scattered

Quantify the strength and direction via **correlation**:

Pearson correlation (R or p): number between -1 and 1 that indicates the strength and direction of association between two variables. The sign of the correlation indicates the

direction, i.e. negative \rightarrow negative linear association. The closer the number is to 1 or -1 the stronger the association is.

One caveat with correlations: Correlation does NOT imply causation. I.e. age might not be the reason to why blood pressure is increasing even though we might see a positive correlation.

2.3.2 Outliers

Outliers are extreme data points that deviate from patterns observed in the rest of the data.

2.4 Simpson's Paradox

A **confounding variable** is an outside influence that changes the relationship between the independent and the dependent variable (ie. a third variable in a study examining a potential cause-and-effect relationship). It oftentimes works by affecting the causal relationship between the primary independent variable and the dependent variable. This confounding variable confuses the relationship between two other variables; it may act by hiding, obscuring, or enhancing the existing relationship.

For example, suppose that you are interested in examining how activity level affects weight change. Other factors, like diet, age, and gender may also affect weight change and come into play when looking at the relationship between activity level and weight change. If one doesn't control for these factors, the relationship between activity level and weight change can be distorted.

2.5 Populations and samples

2.5.1 Sampling from populations intro

How to make inferential statements about a population?

1. Conduct a **census**, i.e. measure everyone in a population. So only doable for small populations and can get easily get expensive, so requires careful cost evaluation.

2. Selected a **probability sample from the population** and measure all units in that sample. Here, we construct a list of all units in a population, i.e. a sampling frame and then we determine a probability of selection for every unit on that list. Then select units from that list at random where sampling rates for different subgroups are determined by the probabilities of selection. Finally, measure those randomly selected units.
3. Select a **non-probability sample** from the population. This process doesn't involve random selection of individuals, according to probability of selection, so there is no statistical basis for making inferences about the target population → high potential for bias. Examples are opt-in web surveys, quota sampling, snowball sampling or convenience sampling.

Why probability sampling?

The known probabilities of selection for all units allow us to make unbiased statements about population features and the uncertainty in survey estimates. The random selection of units protects us against bias from the sample selection mechanisms → allows us to make population inferences based on sampling distributions.

2.5.2 Simple random sampling (SRS)

- We start with a known list of N population units (N = size of the population) and randomly select n units from the list (n = the size of our sample)
- Every unit has equal probability of selection of n/N
- All possible samples of size n are equally likely
- Estimates of means, proportions and totals based on SRS are unbiased (i.e. equal to the population averages on average)
- Can be with replacement or without replacement
 - **With replacement** means that when we select somebody from a larger list, we've replaced them in that list. And we give them a chance of being selected again in the sample.
 - More common is that simple random sampling is done **without replacement**. So once an individual unit is sampled from a given list, they can't be sampled again.
 - For both: The probability of selection for each unit is still n/N
- Rarely used in practice: Collecting data from n randomly sampled units in a large population can be expensive
- Example: We have 1000 email conversations and want to check 100 manually for some things, then we can design a random nr. generator to pull out a simple, random sample from our population of emails.

2.5.3 Working with complex samples

Key features of complex samples:

- Population is divided into different **strata**, and part of the sample is allocated to each stratum. This ensures the a sample representation from each stratum and reduces variance of survey estimates (this technique is known as **stratification**)
- **Clusters** of population units (e.g. counties) are randomly samples first (with a known probability) within strata, to save costs of data collection (i.e. we collect data from cases close to each other geographically for the county example)
- **Units** are randomly sampled from within clusters, according to some probability of selection, and measured

A unit's probability of selection is determined by:

- The number of clusters sampled from each stratum
- The total number of clusters in the population of each stratum
- The number of units ultimately sampled from within each randomly selected cluster
- The total number of units in the population in each cluster

An example for finding a unit's probability of selection:

- We want to select a out of A clusters at random in a given stratum
- Then we select b out of B units at random from within each selected cluster
- The probability of selection is: $(a/A)(b/B)$

3 What is Jupyter Notebooks?

Jupyter is a web-based interactive development environment that supports multiple programming languages, however most commonly used with the Python programming language.

The interactive environment that Jupyter provides enables students, scientists, and researchers to create reproducible analysis and formulate a story within a single document.

Lets take a look at an example of a completed Jupyter Notebook: [Example Notebook](#)

3.0.1 Jupyter Notebook Features

- File Browser
- Markdown Cells & Syntax
- Kernels, Variables, & Environment
- Command vs. Edit Mode & Shortcuts

3.0.2 What is Markdown?

Markdown is a markup language that uses plain text formatting syntax. This means that we can modify the formatting our text with the use of various symbols on our keyboard as indicators.

Some examples include:

- Headers
- Text modifications such as italics and bold
- Ordered and Unordered lists
- Links
- Tables
- Images
- Etc.

Now I'll showcase some examples of how this formatting is done:

Headers:

4 H1

4.1 H2

4.1.1 H3

4.1.1.1 H4

4.1.1.1.1 H5

4.1.1.1.1.1 H6

Text modifications:

Emphasis, aka italics, with *asterisks* or *underscores*.

Strong emphasis, aka bold, with **asterisks** or **underscores**.

Combined emphasis with ***asterisks and underscores***.

Strikethrough uses two tildes. ~~Scratch this.~~

Lists:

1. First ordered list item
 2. Another item
- Unordered sub-list.
1. Actual numbers don't matter, just that it's a number
 2. Ordered sub-list
 3. And another item.
- Unordered list can use asterisks
 - Or minuses
 - Or pluses

Links:

<http://www.umich.edu>

<http://www.umich.edu>

[The University of Michigan's Homepage](#)

To look into more examples of Markdown syntax and features such as tables, images, etc. head to the following link: [Markdown Reference](#)

4.1.2 Kernels, Variables, and Environment

A notebook kernel is a “computational engine” that executes the code contained in a Notebook document. There are kernels for various programming languages, however we are solely using the python kernel which executes python code.

When a notebook is opened, the associated kernel is automatically launched for our convenience.

```
### This is python
print("This is a python code cell")
```

This is a python code cell

A kernel is the back-end of our notebook which not only executes our python code, but stores our initialized variables.

```
### For example, lets initialize variable x

x = 1738

print("x has been set to " + str(x))
```

x has been set to 1738

```
### Print x

print(x)
```

1738

Issues arise when we restart our kernel and attempt to run code with variables that have not been reinitialized.

If the kernel is reset, make sure to rerun code where variables are initialized.

```
## We can also run code that accepts input

name = input("What is your name? ")

print("The name you entered is " + name)
```

It is important to note that Jupyter Notebooks have in-line cell execution. This means that a prior executing cell must complete its operations prior to another cell being executed. A cell still being executing is indicated by the `[*]` on the left-hand side of the cell.

```
print("This won't print until all prior cells have finished
↪   executing.")
```

4.1.3 Command vs. Edit Mode & Shortcuts

There is an edit and a command mode for jupyter notebooks. The mode is easily identifiable by the color of the left border of the cell.

Blue = Command Mode.

Green = Edit Mode.

Command Mode can be toggled by pressing **esc** on your keyboard.

Commands can be used to execute notebook functions. For example, changing the format of a markdown cell or adding line numbers.

Let's toggle line numbers while in command mode by pressing **L**.

4.1.3.1 Additional Shortcuts

There are a lot of shortcuts that can be used to improve productivity while using Jupyter Notebooks.

Here is a list:

Command Mode (press Esc to enable)		Edit Mode (press Enter to enable)	
Enter	enter edit mode	Tab	code completion or indent
Shift-Enter	run cell, select below	Shift-Tab	tooltip
Ctrl-Enter	run cell	Ctrl-]	indent
Alt-Enter	run cell, insert below	Ctrl-[dedent
Y	to code	Ctrl-A	select all
M	to markdown	Ctrl-Z	undo
R	to raw	Ctrl-Shift-Z	redo
1	to heading 1	Ctrl-Y	redo
2,3,4,5,6	to heading 2,3,4,5,6	Ctrl-Home	go to cell start
Up/K	select cell above	Ctrl-Up	go to cell start
Down/J	select cell below	Ctrl-End	go to cell end
A/B	insert cell above/below	Ctrl-Down	go to cell end
X	cut selected cell	Ctrl-Left	go one word left
C	copy selected cell	Ctrl-Right	go one word right
Shift-V	paste cell above	Ctrl-Backspace	delete word before
V	paste cell below	Ctrl-Delete	delete word after
Z	undo last cell deletion	Esc	command mode
D,D	delete selected cell	Ctrl-M	command mode
Shift-M	merge cell below	Shift-Enter	run cell, select below
Ctrl-S	Save and Checkpoint	Ctrl-Enter	run cell
L	toggle line numbers	Alt-Enter	run cell, insert below
O	toggle output	Ctrl-Shift-Subtract	split cell
Shift-O	toggle output scrolling	Ctrl-Shift--	split cell
Esc	close pager	Ctrl-S	Save and Checkpoint
H	show keyboard shortcut help dialog	Up	move cursor up or previous cell
I,I	interrupt kernel	Down	move cursor down or next cell
0,0	restart kernel	Ctrl-/	toggle comment on current or selected lines
Space	scroll down		
Shift-Space	scroll up		
Shift	ignore		

Figure 4.1: Jupyter Notebook Shortcuts

4.1.4 How do you install Jupyter Notebooks?

Note: *Coursera provides embedded jupyter notebooks within the course, thus the download is not a requirement unless you wish to explore jupyter further on your own computer.*

Official Installation Guide: <https://jupyter.readthedocs.io/en/latest/install.html>

Jupyter recommends utilizing Anaconda, which is a platform compatible with Windows, macOS, and Linux systems.

Anaconda Download: <https://www.anaconda.com/download/#macos>

5 Data Types in Python

The following data types can be used in base python: * **boolean** * **integer** * **float** * **string** * **list** * **None** * complex * object * set * dictionary

We will only focus on the **bolded** ones

Let's connect these data types to the the variable types we learned from the [Variable Types video](#).

5.0.1 Numerical or Quantitative (taking the mean makes sense)

- Discrete
 - Integer (int) #Stored exactly, i.e. a whole number
- Continuous
 - Float (float) #Stored similarly to scientific notation. Allows for decimal places but loses precision.

```
import math
```

```
#the type function tells us with what data type we are working  
type(4)
```

int

```
type(0)
```

int

```
type(-3)
```

int

```
#try taking the mean
numbers = [2, 3, 4, 5]
print(sum(numbers)/len(numbers))
type(sum(numbers)/len(numbers)) #In Python 3 returns float, but in
↳ Python 2 would return int
```

3.5

float

Floats

```
3/5
```

0.6

```
6*10**(-1)
```

0.6000000000000001

```
type(3/5)
```

float

```
type(math.pi)
```

float

```
type(4.0)
```

float

```
# Try taking the mean
numbers = [math.pi, 3/5, 4.1]
type(sum(numbers)/len(numbers))
```

float

5.0.2 Categorical or Qualitative

- Nominal
 - Boolean (bool)
 - String (str)
 - None (NoneType)
- Ordinal
 - Only defined by how you use the data
 - Often important when creating visuals
 - Lists can hold ordinal information because they have indices

Boolean

Booleans essentially are stored as True or False. I.e. below we see that True is a reserved word in python.

```
# Boolean
type(True)
```

bool

We also can make our own booleans:

```
type(bool('yes'))
```

bool

We can also use booleans in if statements, i.e. If the below is true, print something.

```
# Boolean
if 4 < 5:
    print("Yes!")
```

Yes!

```
myList = [True, 6<5, 1==3, None is None]
for element in myList:
    print(type(element))
```

```
<class 'bool'>
<class 'bool'>
<class 'bool'>
<class 'bool'>
```

For booleans, True equals to the value of 1 and False to the value of 0. This is why we can do math with booleans. Below we get a value of 0.5, since half of the statements above are true.

```
print(sum(myList)/len(myList))
type(sum(myList)/len(myList))
```

0.5

float

String

```
type("This sentence makes sense")
```

str

```
type("Makes sentence this sense")
```

str

```
type("math.pi")
```

str

```
strList = ['dog', 'koala', 'goose']

#the code below gives an error because we can not calculate the mean
↪ on a string
#sum(strList)/len(strList)
```

Nonetype

```
# None
type(None)
```

NoneType

```
# None
x = None
type(x)
```

NoneType

```
noneList = [None]*5

##the code below gives an error because we can not calculate the mean
↪ on a NoneType
#sum(noneList)/len(noneList)
```

Lists

A list can hold many types and can also be used to store ordinal information.

```
# List
myList = [1, 1.1, "This is a sentence", None]

for element in myList:
    print(type(element))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'NoneType'>
```

```
#this would give an error because the list contains categorical data
#sum(myList)/len(myList)
```

```
# List
myList = [1, 2, 3]

for element in myList:
    print(type(element))

sum(myList)/len(myList) # note that this outputs a float
```

```
<class 'int'>
<class 'int'>
<class 'int'>
```

2.0

While we would see the order in the list below, by default Python does not see this as an ordinal category:

```
myList = ['third', 'first', 'medium', 'small', 'large']

#use an index to access data
myList[0]
```

```
'third'
```

```
myList.sort()  
myList
```

```
['first', 'large', 'medium', 'small', 'third']
```

There are more datatypes available when using different libraries such as Pandas and Numpy, which we will introduce to you as we use them.

6 Python Libraries

Python, like other programming languages, has an abundance of additional modules or libraries that augment the base framework and functionality of the language.

Think of a library as a collection of functions that can be accessed to complete certain programming tasks without having to write your own algorithm.

For this course, we will focus primarily on the following libraries:

- **Numpy** is a library for working with arrays of data.
- **Pandas** provides high-performance, easy-to-use data structures and data analysis tools.
- **Scipy** is a library of techniques for numerical and scientific computing.
- **Matplotlib** is a library for making graphs.
- **Seaborn** is a higher-level interface to Matplotlib that can be used to simplify many graphing tasks.
- **Statsmodels** is a library that implements many statistical techniques.

7 Documentation

Reliable and accesible documentation is an absolute necessity when it comes to knowledge transfer of programming languages. Luckily, python provides a significant amount of detailed documentation that explains the ins and outs of the language syntax, libraries, and more.

Understanding how to read documentation is crucial for any programmer as it will serve as a fantastic resource when learning the intricacies of python.

Here is the link to the documentation of the python standard library: [Python Standard Library](#)

7.0.1 Importing Libraries

When using Python, you must always begin your scripts by importing the libraries that you will be using.

The following statement imports the numpy and pandas library, and gives them abbreviated names:

```
import numpy as np
import pandas as pd
```

7.0.2 Utilizing Library Functions

After importing a library, its functions can then be called from your code by prepending the library name to the function name. For example, to use the `dot` function from the `numpy` library, you would enter `numpy.dot`. To avoid repeatedly having to type the library name in your scripts, it is conventional to define a two or three letter abbreviation for each library, e.g. `numpy` is usually abbreviated as `np`. This allows us to use `np.dot` instead of `numpy.dot`. Similarly, the Pandas library is typically abbreviated as `pd`.

The next cell shows how to call functions within an imported library:

```
a = np.array([0,1,2,3,4,5,6,7,8,9,10])  
np.mean(a)
```

5.0

As you can see, we used the `mean()` function within the numpy library to calculate the mean of the numpy 1-dimensional array.

8 Data Management

Data management is a crucial component to statistical analysis and data science work. The following code will show how to import data via the pandas library, view your data, and transform your data.

The main data structure that Pandas works with is called a **Data Frame**. This is a two-dimensional table of data in which the rows typically represent cases (e.g. Cartwheel Contest Participants), and the columns represent variables. Pandas also has a one-dimensional data structure called a **Series** that we will encounter when accessing a single column of a Data Frame.

Pandas has a variety of functions named ‘`read_xxx`’ for reading data in different formats. Right now we will focus on reading ‘`csv`’ files, which stands for comma-separated values. However the other file formats include excel, json, and sql just to name a few.

This is a link to the `.csv` that we will be exploring in this tutorial: [Cartwheel Data](#) (Link goes to the dataset section of the Resources for this course)

There are many other options to ‘`read_csv`’ that are very useful. For example, you would use the option `sep='\t'` instead of the default `sep=','` if the fields of your data file are delimited by tabs instead of commas. See [here](#) for the full documentation for ‘`read_csv`’.

8.0.1 Importing Data

```
# Store the url string that hosts our .csv file (note that this is a
↪ different url than in the video)
url = "../data/Cartwheeldata.csv"

# Read the .csv file and store it as a pandas Data Frame
df = pd.read_csv(url)

# Output object type
type(df)
```

```
pandas.core.frame.DataFrame
```

8.0.2 Viewing Data

```
# We can view our Data Frame by calling the head() function
df.head()
```

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Comp
0	1	56	F	1	Y	1	62.0	61.0	79	Y
1	2	26	F	1	Y	1	62.0	60.0	70	Y
2	3	33	F	1	Y	1	66.0	64.0	85	Y
3	4	39	F	1	N	0	64.0	63.0	87	Y
4	5	27	M	2	N	0	73.0	75.0	72	N

The `head()` function simply shows the first 5 rows of our Data Frame. If we wanted to show the entire Data Frame we would simply write the following:

```
# Output entire Data Frame
df
```

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Comp
0	1	56	F	1	Y	1	62.00	61.0	79	Y
1	2	26	F	1	Y	1	62.00	60.0	70	Y
2	3	33	F	1	Y	1	66.00	64.0	85	Y
3	4	39	F	1	N	0	64.00	63.0	87	Y
4	5	27	M	2	N	0	73.00	75.0	72	N
5	6	24	M	2	N	0	75.00	71.0	81	N
6	7	28	M	2	N	0	75.00	76.0	107	Y
7	8	22	F	1	N	0	65.00	62.0	98	Y
8	9	29	M	2	Y	1	74.00	73.0	106	N
9	10	33	F	1	Y	1	63.00	60.0	65	Y
10	11	30	M	2	Y	1	69.50	66.0	96	Y
11	12	28	F	1	Y	1	62.75	58.0	79	Y
12	13	25	F	1	Y	1	65.00	64.5	92	Y
13	14	23	F	1	N	0	61.50	57.5	66	Y
14	15	31	M	2	Y	1	73.00	74.0	72	Y

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Com
15	16	26	M	2	Y	1	71.00	72.0	115	Y
16	17	26	F	1	N	0	61.50	59.5	90	N
17	18	27	M	2	N	0	66.00	66.0	74	Y
18	19	23	M	2	Y	1	70.00	69.0	64	Y
19	20	24	F	1	Y	1	68.00	66.0	85	Y
20	21	23	M	2	Y	1	69.00	67.0	66	N
21	22	29	M	2	N	0	71.00	70.0	101	Y
22	23	25	M	2	N	0	70.00	68.0	82	Y
23	24	26	M	2	N	0	69.00	71.0	63	Y
24	25	23	F	1	Y	1	65.00	63.0	67	N

As you can see, we have a 2-Dimensional object where each row is an independent observation of our cartwheel data.

To gather more information regarding the data, we can view the column names and data types of each column with the following functions:

```
df.columns
```

```
Index([u'ID', u'Age', u'Gender', u'GenderGroup', u'Glasses', u'GlassesGroup',
       u'Height', u'Wingspan', u'CWDistance', u'Complete', u'CompleteGroup',
       u'Score'],
      dtype='object')
```

Lets say we would like to splice our data frame and select only specific portions of our data. There are three different ways of doing so.

1. `.loc()`
2. `.iloc()`
3. `.ix()`

We will cover the `.loc()` and `.iloc()` splicing functions.

8.0.3 `.loc()`

`.loc()` takes two single/list/range operator separated by `','`. The first one indicates the row and the second one indicates columns.

```
# Return all observations of CWDistance
df.loc[:, "CWDistance"]
```

```
0    79
1    70
2    85
3    87
4    72
5    81
6   107
7    98
8   106
9    65
10   96
11   79
12   92
13   66
14   72
15  115
16   90
17   74
18   64
19   85
20   66
21  101
22   82
23   63
24   67
```

Name: CWDistance, dtype: int64

```
# Select all rows for multiple columns, ["CWDistance", "Height",
↪ "Wingspan"]
df.loc[:, ["CWDistance", "Height", "Wingspan"]]
```

	CWDistance	Height	Wingspan
0	79	62.00	61.0
1	70	62.00	60.0

	CWDistance	Height	Wingspan
2	85	66.00	64.0
3	87	64.00	63.0
4	72	73.00	75.0
5	81	75.00	71.0
6	107	75.00	76.0
7	98	65.00	62.0
8	106	74.00	73.0
9	65	63.00	60.0
10	96	69.50	66.0
11	79	62.75	58.0
12	92	65.00	64.5
13	66	61.50	57.5
14	72	73.00	74.0
15	115	71.00	72.0
16	90	61.50	59.5
17	74	66.00	66.0
18	64	70.00	69.0
19	85	68.00	66.0
20	66	69.00	67.0
21	101	71.00	70.0
22	82	70.00	68.0
23	63	69.00	71.0
24	67	65.00	63.0

```
# Select few rows for multiple columns, ["CWDistance", "Height",
↪ "Wingspan"]
df.loc[:9, ["CWDistance", "Height", "Wingspan"]]
```

	CWDistance	Height	Wingspan
0	79	62.0	61.0
1	70	62.0	60.0
2	85	66.0	64.0
3	87	64.0	63.0
4	72	73.0	75.0
5	81	75.0	71.0
6	107	75.0	76.0

	CWDistance	Height	Wingspan
7	98	65.0	62.0
8	106	74.0	73.0
9	65	63.0	60.0

```
# Select range of rows for all columns
df.loc[10:15]
```

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Com
10	11	30	M	2	Y	1	69.50	66.0	96	Y
11	12	28	F	1	Y	1	62.75	58.0	79	Y
12	13	25	F	1	Y	1	65.00	64.5	92	Y
13	14	23	F	1	N	0	61.50	57.5	66	Y
14	15	31	M	2	Y	1	73.00	74.0	72	Y
15	16	26	M	2	Y	1	71.00	72.0	115	Y

The `.loc()` function requires two arguments, the indices of the rows and the column names you wish to observe.

In the above case `:` specifies all rows, and our column is **CWDistance**. `df.loc[:, "CWDistance"]`

Now, let's say we only want to return the first 10 observations:

```
df.loc[:9, "CWDistance"]
```

```
0    79
1    70
2    85
3    87
4    72
5    81
6   107
7    98
8   106
9    65
Name: CWDistance, dtype: int64
```


8.0.4 .iloc()

.iloc() is integer based slicing, whereas .loc() used labels/column names. Here are some examples:

```
#return the first 4 rows up to but not including index 4:  
df.iloc[:4]
```

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Comp
0	1	56	F	1	Y	1	62.0	61.0	79	Y
1	2	26	F	1	Y	1	62.0	60.0	70	Y
2	3	33	F	1	Y	1	66.0	64.0	85	Y
3	4	39	F	1	N	0	64.0	63.0	87	Y

```
df.iloc[1:5, 2:4]
```

	Gender	GenderGroup
1	F	1
2	F	1
3	F	1
4	M	2

```
#loc does not allow for labels, so below gives an error:  
#df.iloc[1:5, ["Gender", "GenderGroup"]]
```

8.1 Explore what datatypes we work with using dtypes

We can view the data types of our data frame columns with by calling .dtypes on our data frame:

```
df.dtypes
```

```
ID                int64
Age               int64
Gender            object
GenderGroup       int64
Glasses          object
GlassesGroup     int64
Height           float64
Wingspan         float64
CWDistance       int64
Complete         object
CompleteGroup    int64
Score            int64
dtype: object
```

The output indicates we have integers, floats, and objects with our Data Frame.

8.2 Print unique values

We may also want to observe the different unique values within a specific column, lets do this for Gender:

```
# List unique values in the df['Gender'] column
df.Gender.unique()
```

```
array(['F', 'M'], dtype=object)
```

```
# Lets explore df["GenderGroup"] as well
df.GenderGroup.unique()
```

```
array([1, 2])
```

It seems that these fields may serve the same purpose, which is to specify male vs. female. Lets check this quickly by observing only these two columns:

```
# Use .loc() to specify a list of multiple column names
df.loc[:,["Gender", "GenderGroup"]]
```

	Gender	GenderGroup
0	F	1
1	F	1
2	F	1
3	F	1
4	M	2
5	M	2
6	M	2
7	F	1
8	M	2
9	F	1
10	M	2
11	F	1
12	F	1
13	F	1
14	M	2
15	M	2
16	F	1
17	M	2
18	M	2
19	F	1
20	M	2
21	M	2
22	M	2
23	M	2
24	F	1

8.3 Summarizing multiple columns using groupby

From eyeballing the output, it seems to check out. We can streamline this by utilizing the `groupby()` and `size()` functions.

```
df.groupby(['Gender', 'GenderGroup']).size()
```

```
Gender  GenderGroup
F        1          12
M        2          13
dtype: int64
```

This output indicates that we have two types of combinations.

- Case 1: Gender = F & Gender Group = 1
- Case 2: Gender = M & GenderGroup = 2.

This validates our initial assumption that these two fields essentially portray the same information.

9 Using Python to read data files and explore their contents

This notebook demonstrates using the [Pandas](#) data processing library to read a dataset into Python, and obtain a basic understanding of its contents.

Note that Python by itself is a general-purpose programming language and does not provide high-level data processing capabilities. The Pandas library was developed to meet this need. Pandas is the most popular Python library for data manipulation, and we will use it extensively in this course.

In addition to Pandas, we will also make use of the following Python libraries

- [Numpy](#) is a library for working with arrays of data
- [Matplotlib](#) is a library for making graphs
- [Seaborn](#) is a higher-level interface to Matplotlib that can be used to simplify many graphing tasks
- [Statsmodels](#) is a library that implements many statistical techniques
- [Scipy](#) is a library of techniques for numerical and scientific computing

9.0.1 Importing libraries

When using Python, you must always begin your scripts by importing the libraries that you will be using. After importing a library, its functions can then be called from your code by prepending the library name to the function name. For example, to use the `dot` function from the `numpy` library, you would enter `numpy.dot`. To avoid repeatedly having to type the library name in your scripts, it is conventional to define a two or three letter abbreviation for each library, e.g. `numpy` is usually abbreviated as `np`. This allows us to use `np.dot` instead of `numpy.dot`. Similarly, the Pandas library is typically abbreviated as `pd`.

The following statement imports the Pandas library, and gives it the abbreviated name `pd`.

```
import pandas as pd
```

9.0.2 Reading a data file

We will be working with the NHANES (National Health and Nutrition Examination Survey) data from the 2015-2016 wave, which has been discussed earlier in this course. The raw data for this study are available here:

<https://wwwn.cdc.gov/nchs/nhanes/Default.aspx>

As in many large studies, the NHANES data are spread across multiple files. The NHANES files are stored in [SAS transport](#) (Xport) format. This is a somewhat obscure format, and while Pandas is perfectly capable of reading the NHANES data directly from the xport files, accomplishing this task is a more advanced topic than we want to get into here. Therefore, for this course we have prepared some merged datasets in text/csv format.

Pandas is a large and powerful library. Here we will only use a few of its basic features. The main data structure that Pandas works with is called a “data frame”. This is a two-dimensional table of data in which the rows typically represent cases (e.g. NHANES subjects), and the columns represent variables. Pandas also has a one-dimensional data structure called a **Series** that we will encounter occasionally.

Pandas has a variety of functions named with the pattern ‘`read_xxx`’ for reading data in different formats into Python. Right now we will focus on reading ‘`csv`’ files, so we are using the ‘`read_csv`’ function, which can read csv (and “tsv”) format files that are exported from spreadsheet software like Excel. The ‘`read_csv`’ function by default expects the first row of the data file to contain column names.

Using ‘`read_csv`’ in its default mode is fairly straightforward. There are many options to ‘`read_csv`’ that are useful for handling less-common situations. For example, you would use the option `sep='\t'` instead of the default `sep=','` if the fields of your data file are delimited by tabs instead of commas. See [here](#) for the full documentation for ‘`read_csv`’.

Pandas can read a data file over the internet when provided with a URL, which is what we will do below. In the Python script we will name the data set ‘`da`’, i.e. this is the name of the Python variable that will hold the data frame after we have loaded it.

The variable ‘`url`’ holds a string (text) value, which is the internet URL where the data are located. If you have the data file in your local filesystem, you can also use ‘`read_csv`’ to read the data from this file. In this case you would pass a file path instead of a URL, e.g. `pd.read_csv("my_file.csv")` would read a file named `my_file.csv` that is located in your current working directory.

```
url = "../data/nhanes_2015_2016.csv"
da = pd.read_csv(url)
```

To confirm that we have actually obtained the data the we are expecting, we can display the shape (number of rows and columns) of the data frame in the notebook. Note that the final expression in any Jupyter notebook cell is automatically printed, but you can force other expressions to be printed by using the ‘`print`’ function, e.g. ‘`print(da.shape)`’.

Based on what we see below, the data set being read here has 5735 rows, corresponding to 5735 people in this wave of the NHANES study, and 28 columns, corresponding to 28 variables in this particular data file. Note that NHANES collects thousands of variables on each study subject, but here we are working with a reduced file that contains a limited number of variables.

```
da.shape
```

```
(5735, 28)
```

9.0.3 Exploring the contents of a data set

Pandas has a number of basic ways to understand what is in a data set. For example, above we used the ‘`shape`’ method to determine the numbers of rows and columns in a data set. The columns in a Pandas data frame have names, to see the names, use the ‘`columns`’ method:

```
da.columns
```

```
Index(['SEQN', 'ALQ101', 'ALQ110', 'ALQ130', 'SMQ020', 'RIAGENDR', 'RIDAGEYR',
       'RIDRETH1', 'DMDCITZN', 'DMDEDUC2', 'DMDMARTL', 'DMDHHSIZ', 'WTINT2YR',
       'SDMVPSU', 'SDMVSTRA', 'INDFMPIR', 'BPXSY1', 'BPXDI1', 'BPXSY2',
       'BPXDI2', 'BMXWT', 'BMXHT', 'BMXBMI', 'BMXLEG', 'BMXARML', 'BMXARMC',
       'BMXWAIST', 'HIQ210'],
      dtype='object')
```

These names correspond to variables in the NHANES study. For example, `SEQN` is a unique identifier for one person, and `BMXWT` is the subject’s weight in kilograms (“`BMX`” is the NHANES prefix for body measurements). The variables in the NHANES data set are

documented in a set of “codebooks” that are available on-line. The codebooks for the 2015-2016 wave of NHANES can be found by following the links at the following page:

<https://wwwn.cdc.gov/nchs/nhanes/continuousnhanes/default.aspx?BeginYear=2015>

For convenience, direct links to some of the code books are included below:

- [Demographics code book](#)
- [Body measures code book](#)
- [Blood pressure code book](#)
- [Alcohol questionnaire code book](#)
- [Smoking questionnaire code book](#)

Every variable in a Pandas data frame has a data type. There are many different data types, but most commonly you will encounter floating point values (real numbers), integers, strings (text), and date/time values. When Pandas reads a text/csv file, it guesses the data types based on what it sees in the first few rows of the data file. Usually it selects an appropriate type, but occasionally it does not. To confirm that the data types are consistent with what the variables represent, inspect the ‘`dtypes`’ attribute of the data frame.

```
da.dtypes
```

```
SEQN          int64
ALQ101        float64
ALQ110        float64
ALQ130        float64
SMQ020        int64
RIAGENDR      int64
RIDAGEYR      int64
RIDRETH1      int64
DMDCITZN      float64
DMDEDUC2      float64
DMDMARTL      float64
DMDHHSIZ      int64
WTINT2YR      float64
SDMVPSU       int64
SDMVSTRA      int64
INDFMPIR      float64
BPXSY1        float64
```



```

BPXDI1      float64
BPXSY2      float64
BPXDI2      float64
BMXWT       float64
BMXHT       float64
BMXBMI      float64
BMXLEG      float64
BMXARML     float64
BMXARMC     float64
BMXWAIST    float64
HIQ210      float64
dtype: object

```

As we see here, most of the variables have floating point or integer data type. Unlike many data sets, NHANES does not use any text values in its data. For example, while many datasets would use text labels like “F” or “M” to denote a subject’s gender, this information is represented in NHANES with integer codes. The actual meanings of these codes can be determined from the codebooks. For example, the variable `RIAGENDR` contains each subject’s gender, with male gender coded as 1 and female gender coded as 2. The `RIAGENDR` variable is part of the demographics component of NHANES, so this coding can be found in the demographics codebook.

Variables like `BMXWT` which represent a quantitative measurement will typically be stored as floating point data values.

9.0.4 Slicing a data set

As discussed above, a Pandas data frame is a rectangular data table, in which the rows represent cases and the columns represent variables. One common manipulation of a data frame is to extract the data for one case or for one variable. There are several ways to do this, as shown below.

To extract all the values for one variable, the following three approaches are equivalent (“`DMDEDUC2`” here is an NHANES variable containing a person’s educational attainment). In these four lines of code, we are assigning the data from one column of the data frame `da` into new variables `w`, `x`, `y`, and `z`. The first three approaches access the variable by name. The fourth approach accesses the variable by position (note that `DMDEDUC2` is in position 9 of the `da.columns` array shown above – remember that Python counts starting at position zero).

```
w = da["DMDEDUC2"]
x = da.loc[:, "DMDEDUC2"]
y = da.DMDEDUC2
z = da.iloc[:, 9] # DMDEDUC2 is in column 9
```

Another reason to slice a variable out of a data frame is so that we can then pass it into a function. For example, we can find the maximum value over all DMDEDUC2 values using any one of the following four lines of code:

```
print(da["DMDEDUC2"].max())
print(da.loc[:, "DMDEDUC2"].max())
print(da.DMDEDUC2.max())
print(da.iloc[:, 9].max())
```

```
9.0
9.0
9.0
9.0
```

Every value in a Python program has a type, and the type information can be obtained using Python's 'type' function. This can be useful, for example, if you are looking for the documentation associated with some value, but you do not know what the value's type is.

Here we see that the variable `da` has type 'DataFrame', while one column of `da` has type 'Series'. As noted above, a Series is a Pandas data structure for holding a single column (or row) of data.

```
print(type(da)) # The type of the variable
print(type(da.DMDEDUC2)) # The type of one column of the data frame
print(type(da.iloc[2,:])) # The type of one row of the data frame
```

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

It may also be useful to slice a row (case) out of a data frame. Just as a data frame's columns have names, the rows also have names, which are called the "index". However many data sets do not have meaningful row names, so it is more common to extract a row

of a data frame using its position. The `iloc` method slices rows or columns from a data frame by position (counting from 0). The following line of code extracts row 3 from the data set (which is the fourth row, counting from zero).

```
x = da.iloc[3, :]
```

Another important data frame manipulation is to extract a contiguous block of rows or columns from the data set. Below we slice by position, in the first case taking row positions 3 and 4 (counting from 0, which are rows 4 and 5 counting from 1), and in the second case taking columns 2, 3, and 4 (columns 3, 4, 5 if counting from 1).

```
x = da.iloc[3:5, :]  
y = da.iloc[:, 2:5]
```

9.0.5 Missing values

When reading a dataset using Pandas, there is a set of values including ‘NA’, ‘NULL’, and ‘NaN’ that are taken by default to represent a missing value. The full list of default missing value codes is in the ‘`read_csv`’ documentation [here](#). This document also explains how to change the way that ‘`read_csv`’ decides whether a variable’s value is missing.

Pandas has functions called `isnull` and `notnull` that can be used to identify where the missing and non-missing values are located in a data frame. Below we use these functions to count the number of missing and non-missing `DMDEDUC2` values.

```
print(pd.isnull(da.DMDEDUC2).sum())  
print(pd.notnull(da.DMDEDUC2).sum())
```

```
261  
5474
```

As an aside, note that there may be a variety of distinct forms of missingness in a variable, and in some cases it is important to keep these values distinct. For example, in case of the `DMDEDUC2` variable, in addition to the blank or NA values that Pandas considers to be missing, three people responded “don’t know” (code value 9). In many analyses, the “don’t know” values will also be treated as missing, but at this point we are considering “don’t know” to be a distinct category of observed response.

10 Python Resources

The purpose of this document is to direct you to resources that you may find useful if you decide to do a deeper dive into Python. This course is not meant to be an introduction to programming, nor an introduction to Python, but if you find yourself interested in exploring Python further, or feel as if this is a useful skill, this document aims to direct you to resources that you may find useful. If you have a background in Python or programming, a style guides are included below to show how Python may differ from other programming languages or give you a launching point for diving deeper into more advanced packages. This course does not endorse the use or non-use of any particular resource, but the author has found these resources useful in their exploration of programming and Python in particular

10.0.1 The Python Documentation

Any reference that does not begin with the Python documentation would not be complete. The authors of the language, as well as the community that supports it, have developed a great set of tutorials, documentation, and references around Python. When in doubt, this is often the first place that you should look if you run into a scary error or would like to learn more about a specific function. The documentation can be found here: [Python Documentation](#)

10.0.2 Python Programming Introductions

Below are resources to help you along your way in learning Python. While it is great to consume material, in programming there is no substitute for actually writing code. For every hour that you spend learning, you should spend about twice that amount of time writing code for cool problems or working out examples. Coding is best learned through actually coding!

- [Coursera](#) has several offerings for Python that you can take in addition to this course. These courses will go into depth into Python programming and how to use it in an applied setting

- [Code Academy](#) is another resources that is great for learning Python (and other programming languages). While not as focused as Cousera, this is a quick way to get up-and-running with Python
- YouTube is another great resource for online learning and there are several “courses” for learning Python. We recommend trying several sets of videos to see which you like best and using multiple video series to learn since each will present the material in a slightly different way
- There are tens of books on programming in Python that are great if you prefer to read. More so than the other resources, be sure to code what you learn. It is easy to read about coding, but you really learn to code by coding!
- If you have a background in coding, the authors have found the tutorial at [Tutorials Point](#) to be useful in getting started with Python. This tutorial assumes that you have some background in coding in another language

10.0.3 Cheatsheets and References

There are a variety of one-pagers and cheat-sheets available for Python that summarize the language in a few simple pages. These resources tend to be more aimed at someone who knows the language, or has experience in the language, but would like a refresher course in how the language works.

- [Cheatsheet for Numpy](#)
- [Cheatsheet for Datawrangling](#)
- [Cheatsheet for Pandas](#)
- [Cheatsheet for SciPy](#)
- [Cheatsheet for Matplotlib](#)

10.0.4 Python Style Guides

As you learn to code, you will find that you will begin to develop your own style. Sometimes this is good. Most times, this can be detrimental to your code readability and, worse, can hinder you from finding bugs in your own code in extreme cases.

It is best to learn good coding habits from the beginning and the [Google Style Guide](#) is a great place to start. We will mention some of these best practices here.

10.0.4.1 Consistent Indenting

Python will generally ‘yell’ at you if your indenting is incorrect. It is good to use an editor that takes care of this for you. In general, four spaces are preferred for indenting and you should not mix tabs and spaces.

```
# Good Indenting - four spaces are standard but consistency is key
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
print (result)

# Bad indenting
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
print (result)
```

10.0.4.2 Commenting

Comments seem weird when you first begin programming - why would I include ‘code’ that doesn’t run? Comments are probably some of the most important aspects of code. They help other read code that is difficult for them to understand, and they, more importantly, are helpful for yourself if you look at the code in a few weeks and need clarity on why you did something. Always comment and comment well.

```
#####
#
↳ #
#                               Good Commenting
↳ #
#
↳ #
#####
```

```
##### Bad Commenting
↪ #####

# My loop
for x in range(10):
    print (x)

##### Better Commenting
↪ #####

# Looping from zero to ten
for x in range(10):
    print (x)

##### Preferred Commenting
↪ #####

# Print out the numbers from zero to ten
for x in range(10):
    print (x)
```

```
#####
#
↪ #
#                               Mixing Commenting Strategies
↪ #
#
↪ #
#####

# Try not to mix commenting styles in the same blocks - just be
↪ consistient

##### Bad - mixing doc-strings commenting and line commenting
↪ #####

''' Printing one to five, a six, and then six to nine'''
```

```

for x in range(10):
    # If x > 5, then print the value
    if x > 5:
        print (x)
    else:
        print (x + 1)

##### Good - no mixing of comment types
↪ #####

# Printing one to five, a six, and then six to nine
for x in range(10):
    # If x > 5, then print the value
    if x > 5:
        print (x)
    else:
        print (x + 1)

```

10.0.4.3 Line Length

Try to avoid excessively long lines. Standard practice is to keep lines to no longer than 80 characters. While this is not a hard rule, it is a good practice to follow for readability

```

##### Bad - This code is too long
↪ #####

my_random_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6,
↪ 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]

##### Good - this code is wrapped to avoid excessive length
↪ #####

my_random_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6,
↪ 7, 8, 9,
                    10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5,
↪ 6, 7, 8,
                    9, 10]

```


10.0.4.4 White Space

Utilizing Whitespace is a great way to improve the way that your code looks. In general the following can be helpful to improve the look of your code

- Try to space out your code and introduce whitespace to improve readability
- Use spacing to separate function arguments
- Do not over-do spacing. Too many spaces between code blocks makes it difficult to organize code well

```
##### Bad - this code has bad whitespace management
↪ #####

my_player = player()
player_attributes = get_player_attributes(my_player,height,weight,
↪ birthday)

player_attributes[0]*=12 # convert from feet to inches

player.shoot_ball()

##### Good whitespace management
↪ #####

my_player = player()
player_attributes = get_player_attributes(my_player, height, weight,
↪ birthday)

# convert from feet to inches
player_attributes[0] *= 12

player.shoot_ball()
```

10.0.4.5 The tip of the iceberg

Take a look at code out in the wild if you are really curious. How are they coding specific things? How do they manage spacing in loops? How do they manage the whitespace in argument list?

You will learn to code by coding, and you will develop your own style but starting out with good habits ensures that your code is easy to read by others and, most importantly, yourself. Good luck!

11 Python Libraries

For this tutorial, we are going to outline the most common uses for each of the following libraries:

- **Numpy** is a library for working with arrays of data.
- **Scipy** is a library of techniques for numerical and scientific computing.
- **Matplotlib** is a library for making visualizations.
- **Seaborn** is a higher-level interface to Matplotlib that can be used to simplify many visualization tasks.

***Important:** While this tutorial provides insight into the basics of these libraries, I recommend digging into the documentation that is available online.*

11.1 NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

We will focus on the numpy array object.

11.1.1 Numpy Array

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

```
import numpy as np
```

```
### Create a 3x1 numpy array  
a = np.array([1,2,3])  
print(a)  
  
### Print object type  
print(type(a))
```

```
[1 2 3]  
<class 'numpy.ndarray'>
```

```
### Print shape  
#we have a one dimensional object with 3 values  
print(a.shape)
```

```
(3,)
```

```
### Print some values in a  
print(a[0], a[1], a[2])
```

```
1 2 3
```

```
### Create a 2x2 numpy array using a nested list  
b = np.array([[1,2],[3,4]])  
print(b)
```

```
[[1 2]  
 [3 4]]
```

```
### Print shape  
# we now have a two dimensional array (with 2 rows and 2 columns)
```

```
print(b.shape)
```

(2, 2)

```
#in row two, access the the first value  
#here we index, specifying the row first and then the column position  
#don't forget: in py we start counting at 0  
print(b[1,0])
```

3

```
## Print several values in b  
print(b[0,0], b[0,1], b[1,1])
```

1 2 4

```
### Create a 3x2 numpy array  
c = np.array([[1,2],[3,4],[5,6]])  
c
```

```
array([[1, 2],  
       [3, 4],  
       [5, 6]])
```

```
### Print shape  
print(c.shape)
```

(3, 2)
2 3 5 6

```
### Print some values in c
print(c[0,1], c[1,0], c[2,0], c[2,1])
```

2 3 5 6

11.1.1.1 Create numpy arrays with different automatic numberings

```
### create a 2x3 zero array with only 0s
d = np.zeros((2,3))

print(d)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

```
### 4x2 array of ones
e = np.ones((4,2))

print(e)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

```
### create 2x2 constant array with a specified value
#we first give the nr of rows and columns we want, followed by the
↪ constant value
f = np.full((2,2), 9)

print(f)
```

```
[[9 9]
 [9 9]]
```

```
### create a 3x3 random array with random nrs
g = np.random.random((3,3))

print(g)
```

```
[[0.75578673 0.52378499 0.68715926]
 [0.09153656 0.89729222 0.85334664]
 [0.34651959 0.06094491 0.15857276]]
```

11.1.2 Array Indexing

```
### Create 3x4 array
h = np.array([[1,2,3,4,], [5,6,7,8], [9,10,11,12]])

print(h)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
print(h[0,1])
```

2

```
### Slice array to make a 2x2 sub-array
#first we select rows with index 0 and 1 (so up to but not including
↪ 2)
#then we further select columns with index 1 and 2
i = h[:,2, 1:3]

print(i)
```

```
[[2 3]
 [6 7]]
```

```
### Modify something in the slice
i[0,0] = 1738
print(i)
```

```
[[1738   3]
 [   6   7]]
```

```
#notice how this value is also changed in our original array h!
print(h)
```

```
[[  1 1738   3   4]
 [  5   6   7   8]
 [  9  10  11  12]]
```

11.1.3 Datatypes in Arrays

```
### Integer
j = np.array([1, 2])
print(j)
print(j.dtype)
```

```
[1 2]
int64
```

```
### Float
k = np.array([1.2, 2.0])
print(k)
print(k.dtype)
```

```
[1.2 2. ]
float64
```



```

### Force Data Type
l = np.array([1.0, 2.0], dtype = np.int64)
print(l)
print(l.dtype)

```

```

[1 2]
int64

```

11.1.4 Array Math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```

x = np.array([[1,2],[3,4]], dtype = np.float64)
y = np.array([[5,6],[7,8]], dtype = np.float64)
print(x)

```

```

[[1. 2.]
 [3. 4.]]

```

```

print(y)

```

```

[[5. 6.]
 [7. 8.]]

```

```

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#   [10.0 12.0]]
print(x + y)

```

```

[[ 6.  8.]
 [10. 12.]]

```

```
print(np.add(x, y))
```

```
[[ 6.  8.]  
 [10. 12.]]
```

```
# Elementwise difference; both produce the array  
# [[-4.0 -4.0]  
#  [-4.0 -4.0]]  
print(x - y)
```

```
[[ -4. -4.]  
 [ -4. -4.]]
```

```
print(np.subtract(x, y))
```

```
[[ -4. -4.]  
 [ -4. -4.]]
```

```
# Elementwise product; both produce the array  
# [[ 5.0 12.0]  
#  [21.0 32.0]]  
print(x * y)
```

```
[[ 5. 12.]  
 [21. 32.]]
```

```
print(np.multiply(x, y))
```

```
[[ 5. 12.]  
 [21. 32.]]
```

```
# Elementwise division; both produce the array
# [[ 0.2          0.33333333]
#   [ 0.42857143  0.5         ]]
print(x / y)
```

```
[[0.2          0.33333333]
 [0.42857143  0.5         ]]
```

```
print(np.divide(x, y))
```

```
[[0.2          0.33333333]
 [0.42857143  0.5         ]]
```

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#   [ 1.73205081  2.         ]]
print(np.sqrt(x))
```

```
[[1.          1.41421356]
 [1.73205081  2.         ]]
```

11.1.5 Descriptive statistics with numpy

```
x = np.array([[1,2],[3,4]])
x
```

```
array([[1, 2],
       [3, 4]])
```

```
### Compute sum of all elements; prints "10"
print(np.sum(x))
```

10

```
### Compute sum of each column; prints "[4 6]"  
print(np.sum(x, axis=0))
```

[4 6]

```
### Compute sum of each row; prints "[3 7]"  
print(np.sum(x, axis=1))
```

[3 7]

```
### Compute mean of all elements; prints "2.5"  
print(np.mean(x))
```

2.5

```
### Compute mean of each column; prints "[2 3]"  
print(np.mean(x, axis=0))
```

[2. 3.]

```
### Compute mean of each row; prints "[1.5 3.5]"  
print(np.mean(x, axis=1))
```

[1.5 3.5]

11.2 SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

For this course, we will primarily be using the **SciPy.Stats** sub-library.

11.2.1 SciPy.Stats

The SciPy.Stats module contains a large number of probability distributions as well as a growing library of statistical functions such as:

- Continuous and Discrete Distributions (i.e Normal, Uniform, Binomial, etc.)
- Descriptive Statistics
- Statistical Tests (i.e T-Test)

```
from scipy import stats
import numpy as np
```

```
### Print 10 Normal Random Variables
print(stats.norm.rvs(size = 10))
```

```
[ 1.2273344 -1.18292396  0.06328786 -1.08124849 -0.6143745  0.703326
 1.11746254  1.06465073  0.64391558 -1.8944723 ]
```

```
from pylab import *

# Create some test data
dx = .01
X = np.arange(-2,2,dx)
Y = exp(-X**2)

#print(X)
#print(Y)
```

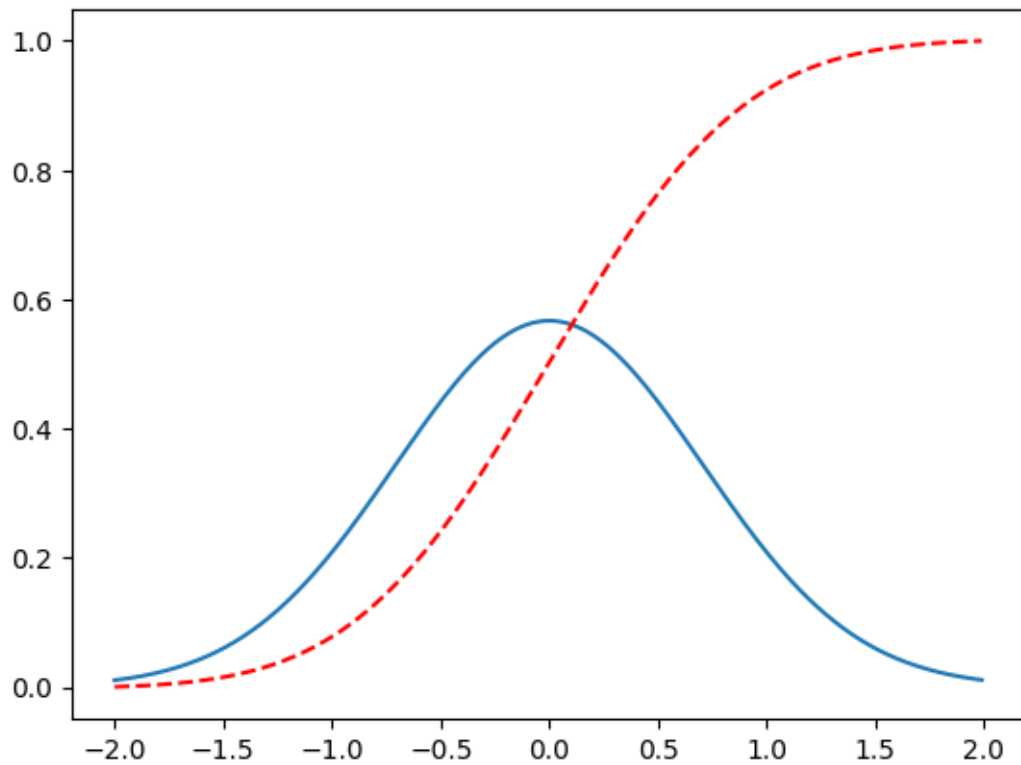
```
# Normalize the data to a proper PDF
Y /= (dx*Y).sum()

# Compute the CDF
CY = np.cumsum(Y*dx)

# Plot both
plot(X,Y)
```

```
plot(X,CY,'r--')

show()
```



```
### Compute the Normal CDF of certain values.
#this returns some probabilities based on the plot above
print(stats.norm.cdf(np.array([1,-1., 0, 1, 3, 4, -2, 6])))
```

```
[0.84134475 0.15865525 0.5          0.84134475 0.9986501  0.99996833
 0.02275013 1.          ]
```

11.2.1.1 Descriptive Statistics

```
np.random.seed(282629734)

# Generate 1000 Student's T continuous random variables.
x = stats.t.rvs(10, size=1000)
```

```
# Do some descriptive statistics
print(x.min()) # equivalent to np.min(x)
```

-3.7897557242248197

```
print(x.max()) # equivalent to np.max(x)
```

5.263277329807165

```
print(x.mean()) # equivalent to np.mean(x)
```

0.014061066398468422

```
print(x.var()) # equivalent to np.var(x))
```

1.288993862079285

```
stats.describe(x)
```

DescribeResult(nobs=1000, minmax=(-3.7897557242248197, 5.263277329807165), mean=0.014061066398468422, var=1.288993862079285, skew=0.0000000000000001, kurt=3.0000000000000004)

Later in the course, we will discuss distributions and statistical tests such as a T-Test. SciPy has built in functions for these operations.

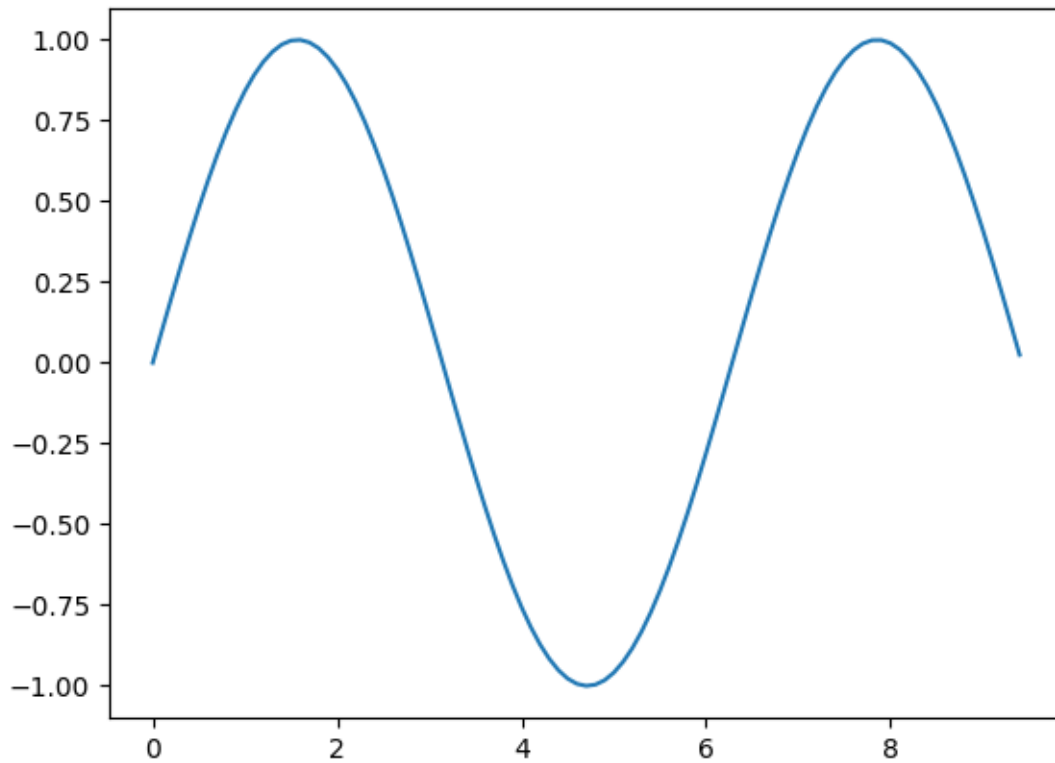
11.3 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the matplotlib.pyplot module.

```
import numpy as np
import matplotlib.pyplot as plt
```

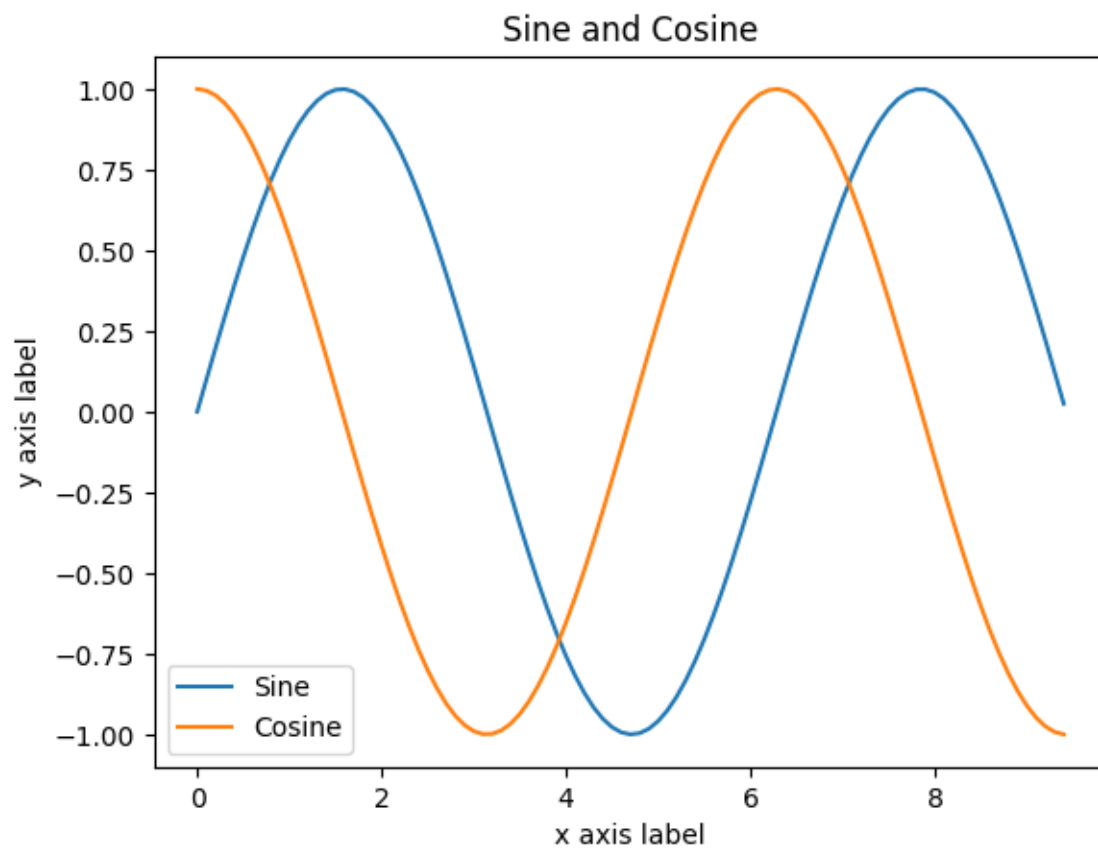
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```




```
# Compute the x and y coordinates for points on sine and cosine
↪ curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



11.3.0.1 Subplots

```
import numpy as np
import matplotlib.pyplot as plt

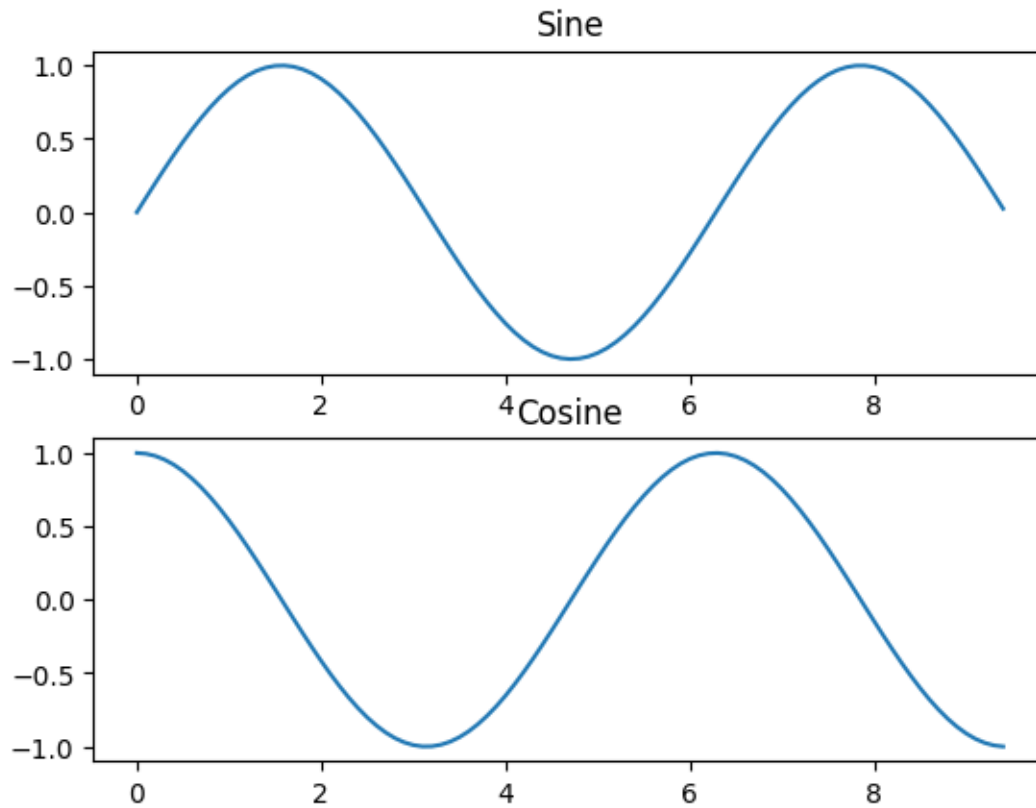
# Compute the x and y coordinates for points on sine and cosine
# ↪ curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



11.4 Seaborn

Seaborn is complimentary to Matplotlib and it specifically targets statistical data visualization. But it goes even further than that: Seaborn extends Matplotlib and makes generating visualizations convenient.

While Matplotlib is a robust solution for various problems, Seaborn utilizes more concise parameters for ease-of-use.

11.4.0.1 Scatterplots

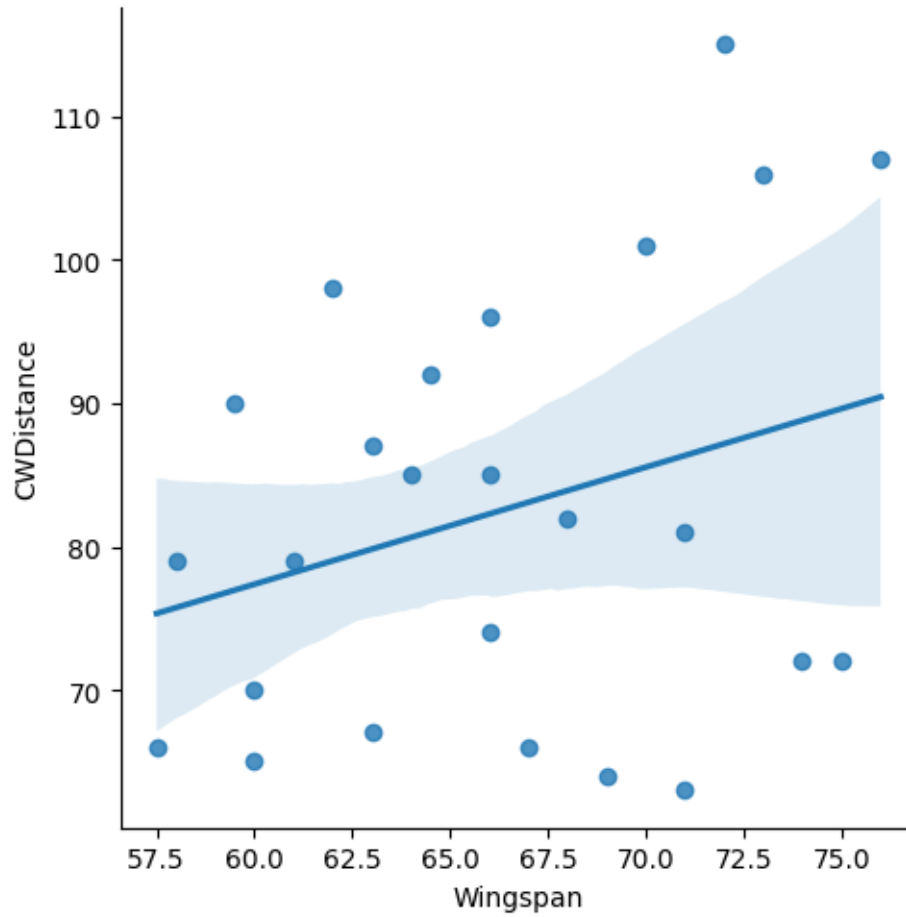
```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

```
# Store the url string that hosts our .csv file
url = "../data/Cartwheeldata.csv"

# Read the .csv file and store it as a pandas Data Frame
df = pd.read_csv(url)

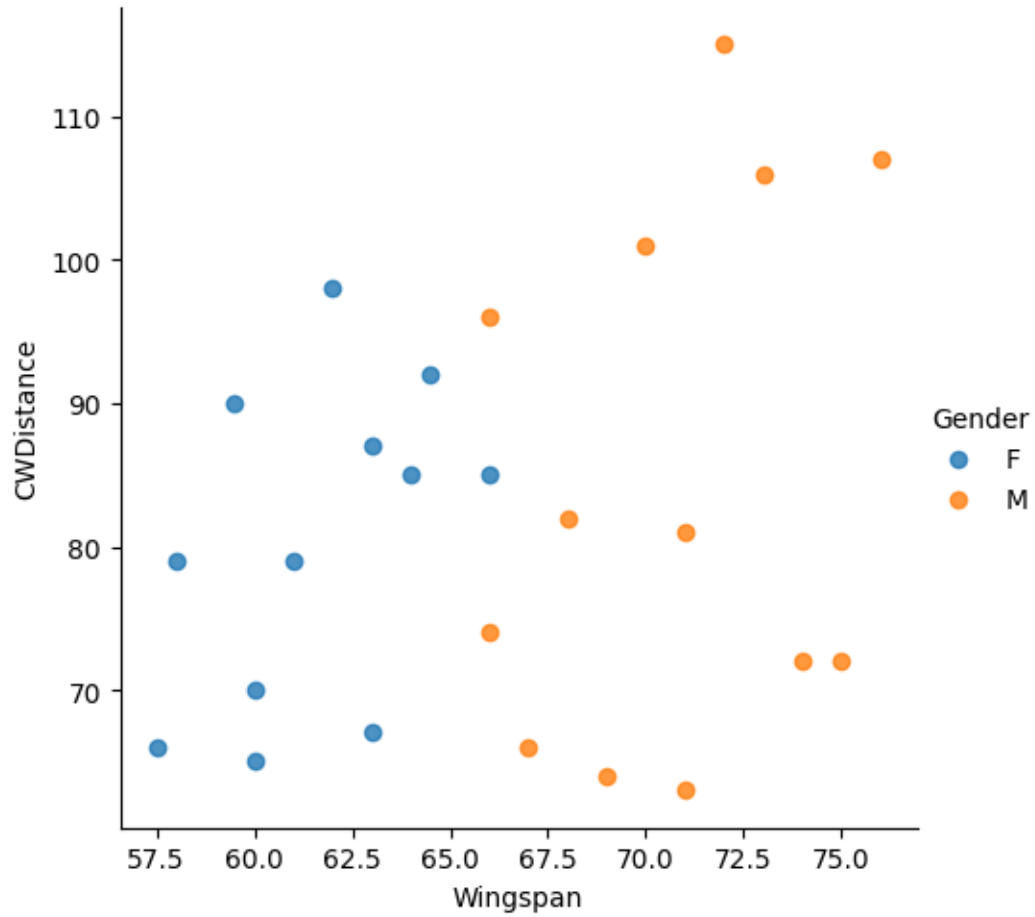
# Create Scatterplot
sns.lmplot(x='Wingspan', y='CWDistance', data=df)

plt.show()
```



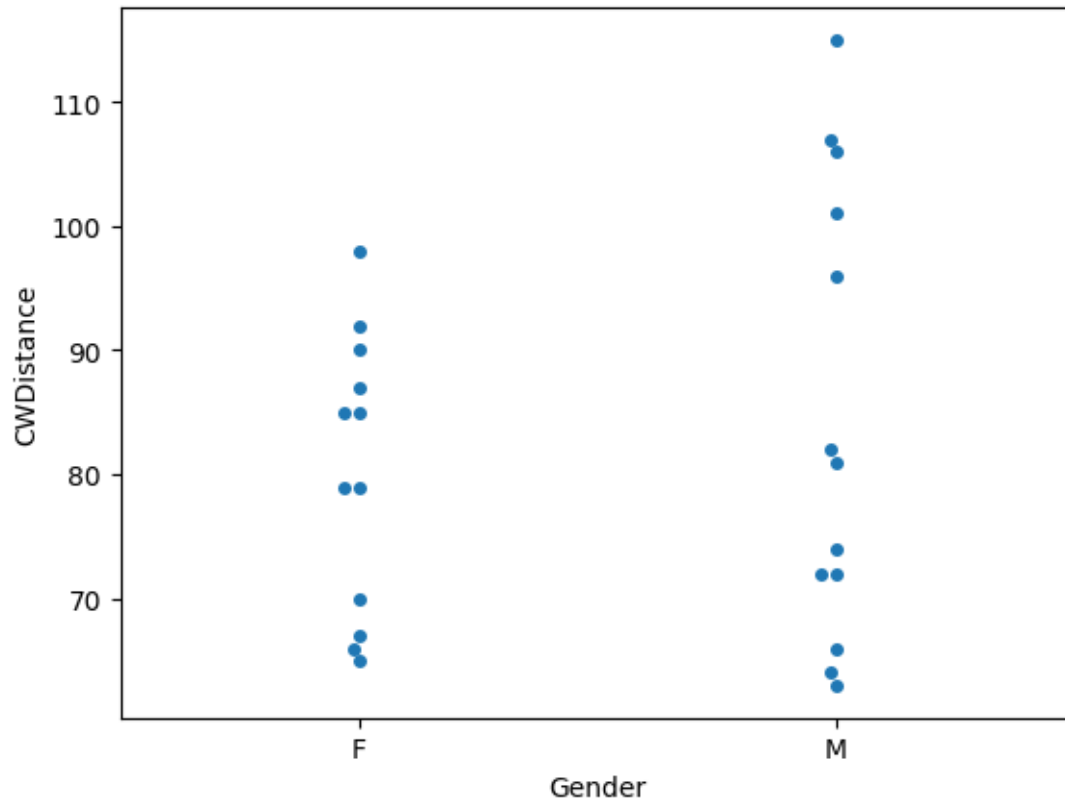
```
# Scatterplot arguments
sns.lmplot(x='Wingspan', y='CWDistance', data=df,
           fit_reg=False, # No regression line
           hue='Gender')  # Color by evolution stage

plt.show()
```



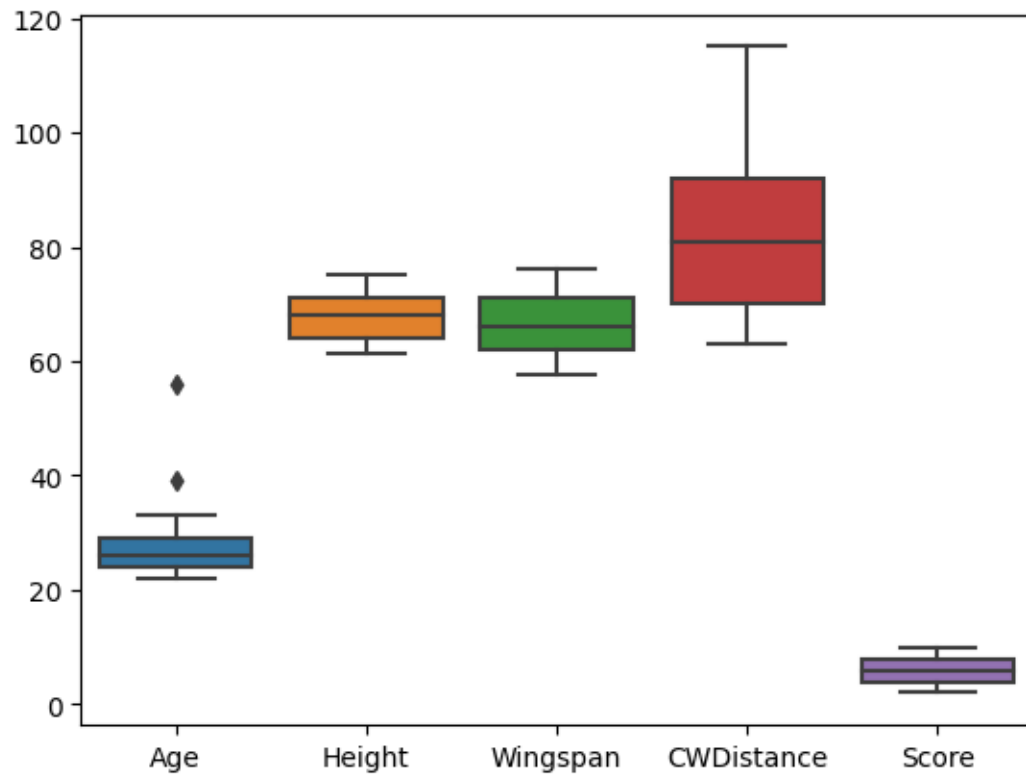
```
# Construct Cartwheel distance plot
sns.swarmplot(x="Gender", y="CWDistance", data=df)

plt.show()
```



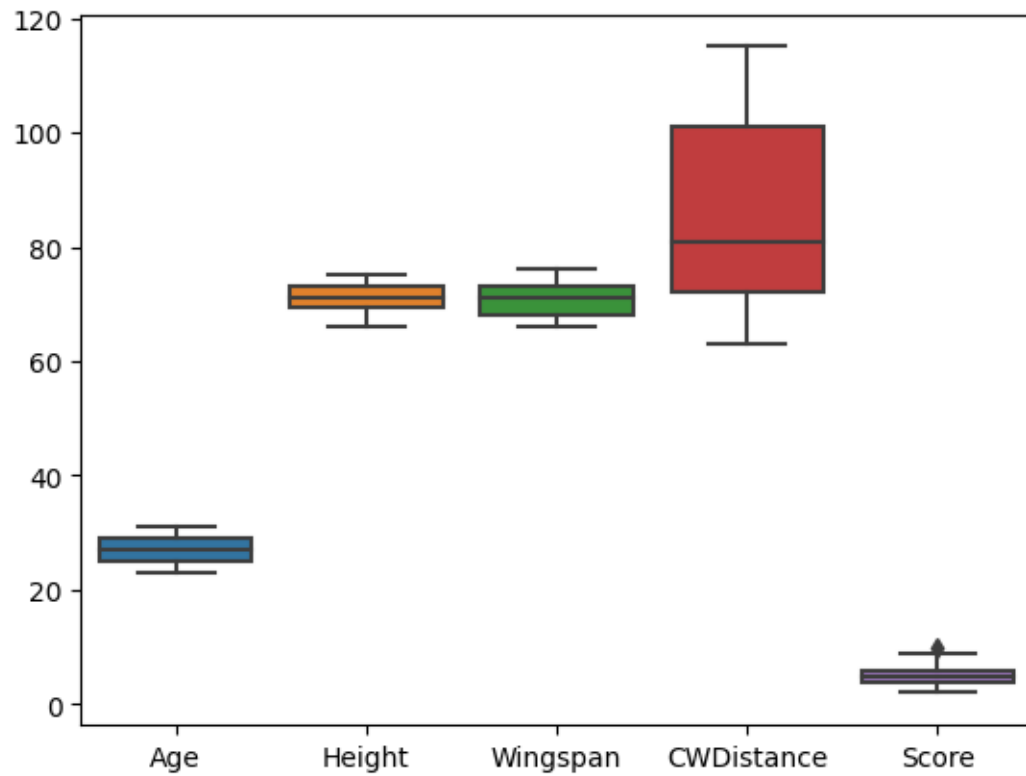
11.4.0.2 Boxplots

```
sns.boxplot(data=df.loc[:, ["Age", "Height", "Wingspan",  
↪ "CWDistance", "Score"]])  
  
plt.show()
```



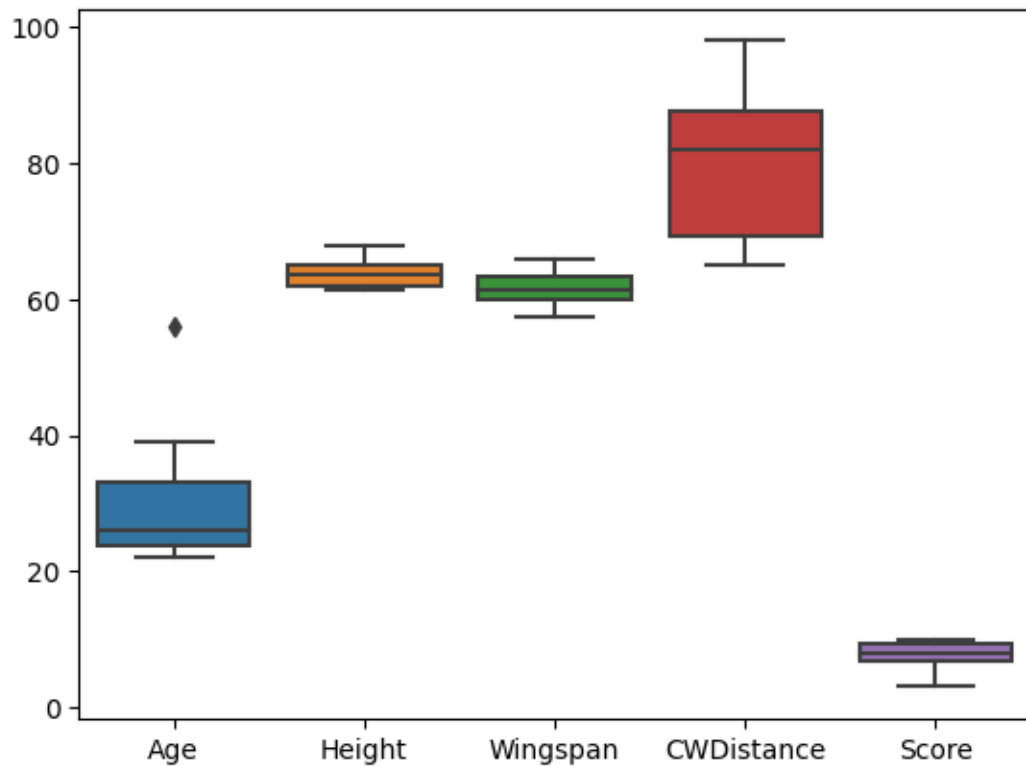
```
# Male Boxplot
sns.boxplot(data=df.loc[df['Gender'] == 'M', ["Age", "Height",
↪ "Wingspan", "CWDistance", "Score"]])

plt.show()
```

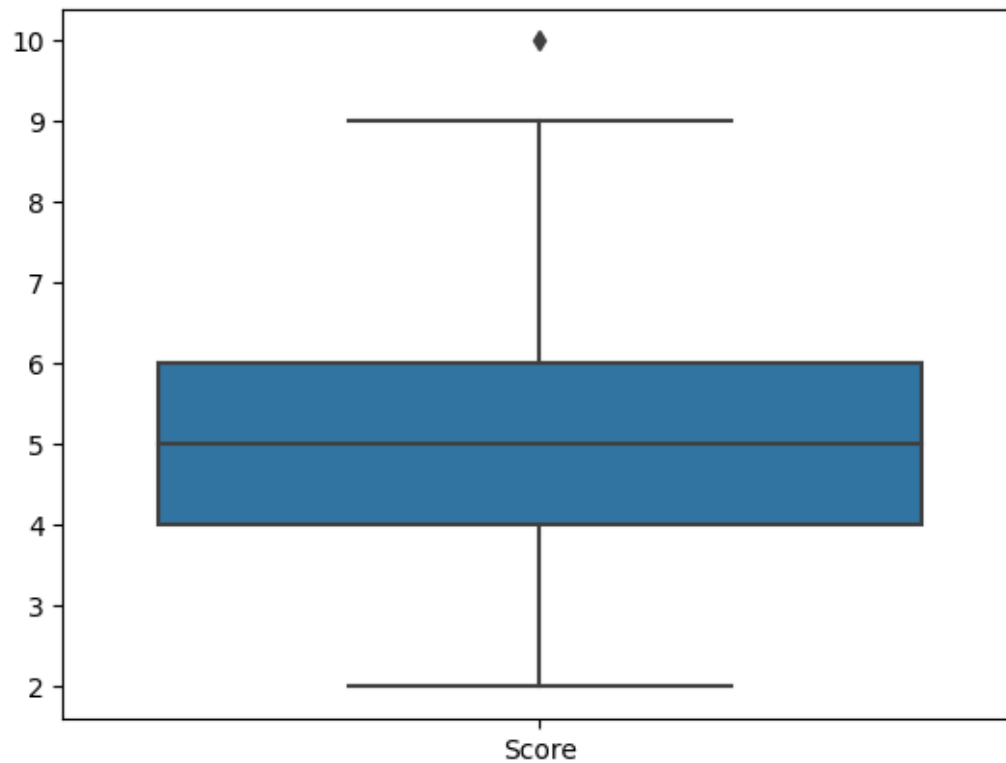



```
# Female Boxplot
sns.boxplot(data=df.loc[df['Gender'] == 'F', ["Age", "Height",
↪ "Wingspan", "CWDistance", "Score"]])

plt.show()
```



```
# Male Boxplot  
sns.boxplot(data=df.loc[df['Gender'] == 'M', ["Score"]])  
  
plt.show()
```



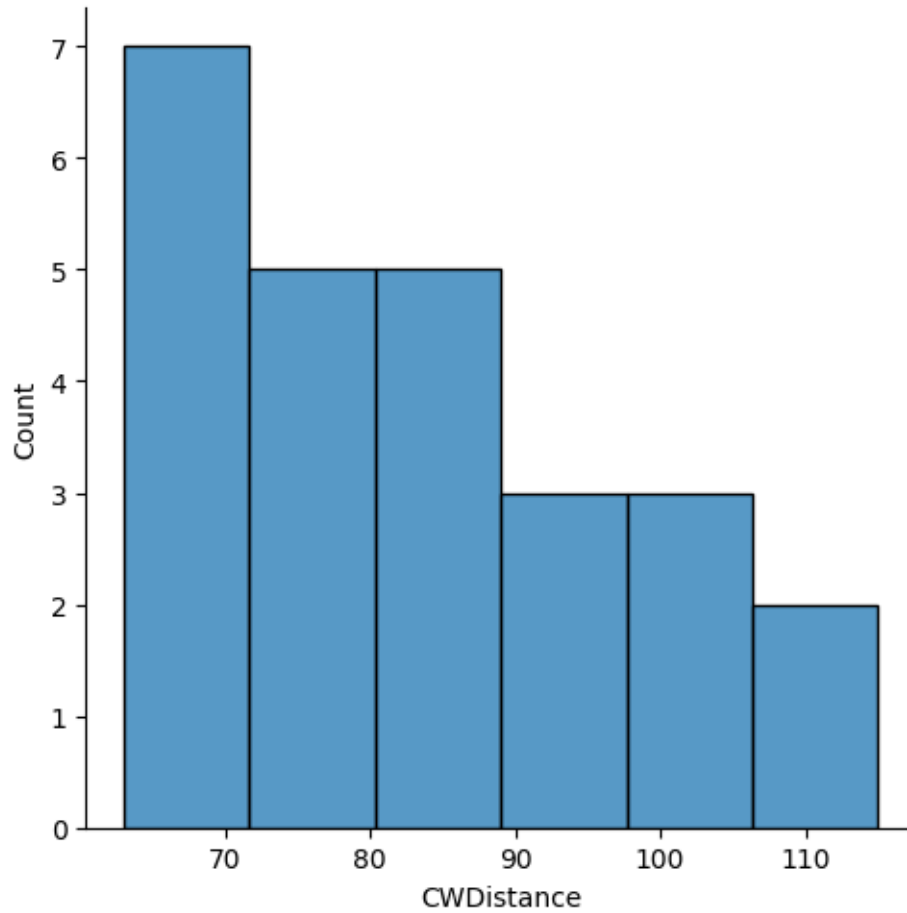
```
# Female Boxplot
sns.boxplot(data=df.loc[df['Gender'] == 'F', ["Score"]])

plt.show()
```

11.4.0.3 Histogram

```
# Distribution Plot (a.k.a. Histogram)
sns.displot(df.CWDistance)

plt.show()
```

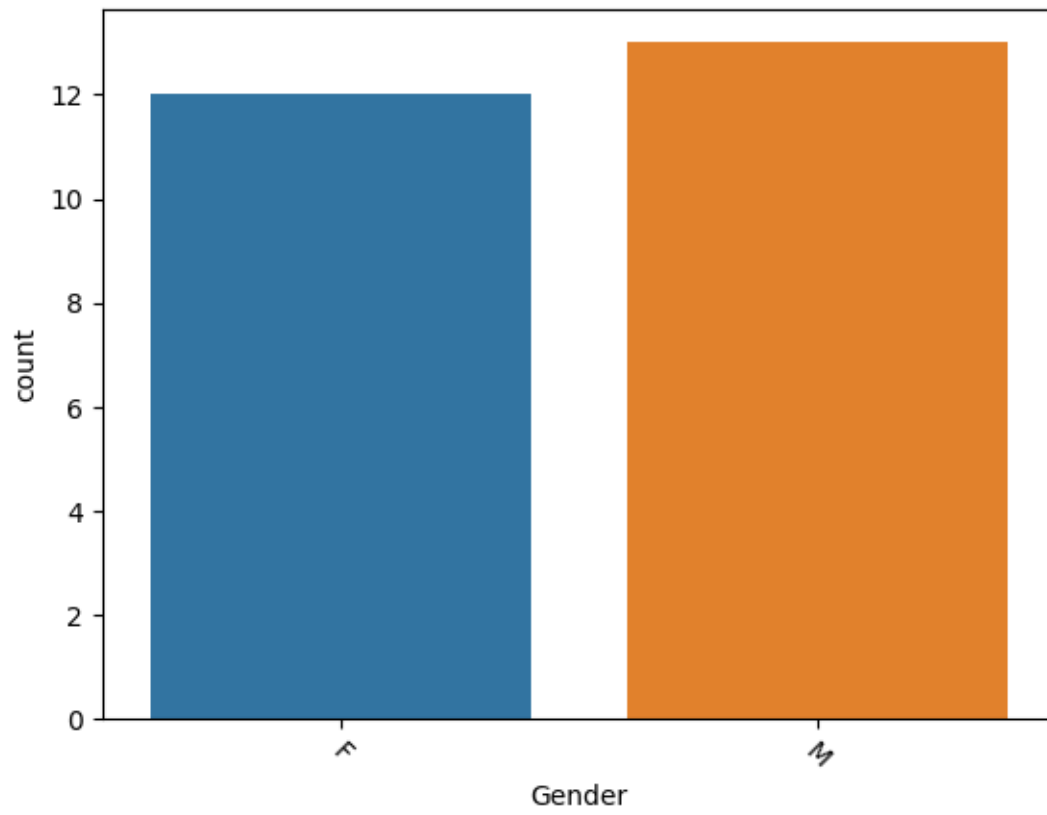


11.4.0.4 Count Plot

```
# Count Plot (a.k.a. Bar Plot)
sns.countplot(x='Gender', data=df)

plt.xticks(rotation=-45)

plt.show()
```



12 Visualizing Data in Python

12.0.0.1 Tables, Histograms, Boxplots, and Slicing for Statistics

When working with a new dataset, one of the most useful things to do is to begin to visualize the data. By using tables, histograms, box plots, and other visual tools, we can get a better idea of what the data may be trying to tell us, and we can gain insights into the data that we may have not discovered otherwise.

Today, we will be going over how to perform some basic visualisations in Python, and, most importantly, we will learn how to begin exploring data from a graphical perspective.

```
# We first need to import the packages that we will be using
import seaborn as sns # For plotting
import matplotlib.pyplot as plt # For showing plots

# Load in the data set
tips_data = sns.load_dataset("tips")
```

12.0.0.2 Visualizing the Data - Tables

When you begin working with a new data set, it is often best to print out the first few rows before you begin other analysis. This will show you what kind of data is in the dataset, what data types you are working with, and will serve as a reference for the other plots that we are about to make.

```
# Print out the first few rows of the data
tips_data.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

	total_bill	tip	sex	smoker	day	time	size
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

12.0.0.3 Describing Data

Summary statistics, which include things like the mean, min, and max of the data, can be useful to get a feel for how large some of the variables are and what variables may be the most important.

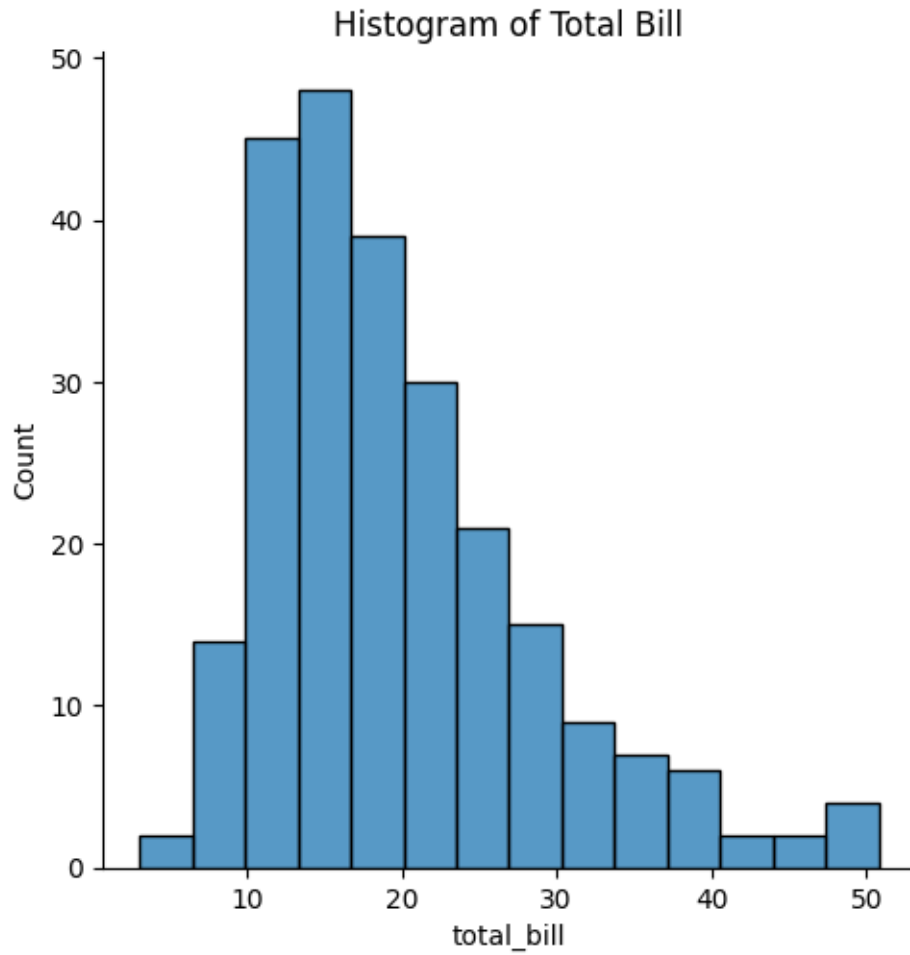
```
# Print out the summary statistics for the quantitative variables
tips_data.describe()
```

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

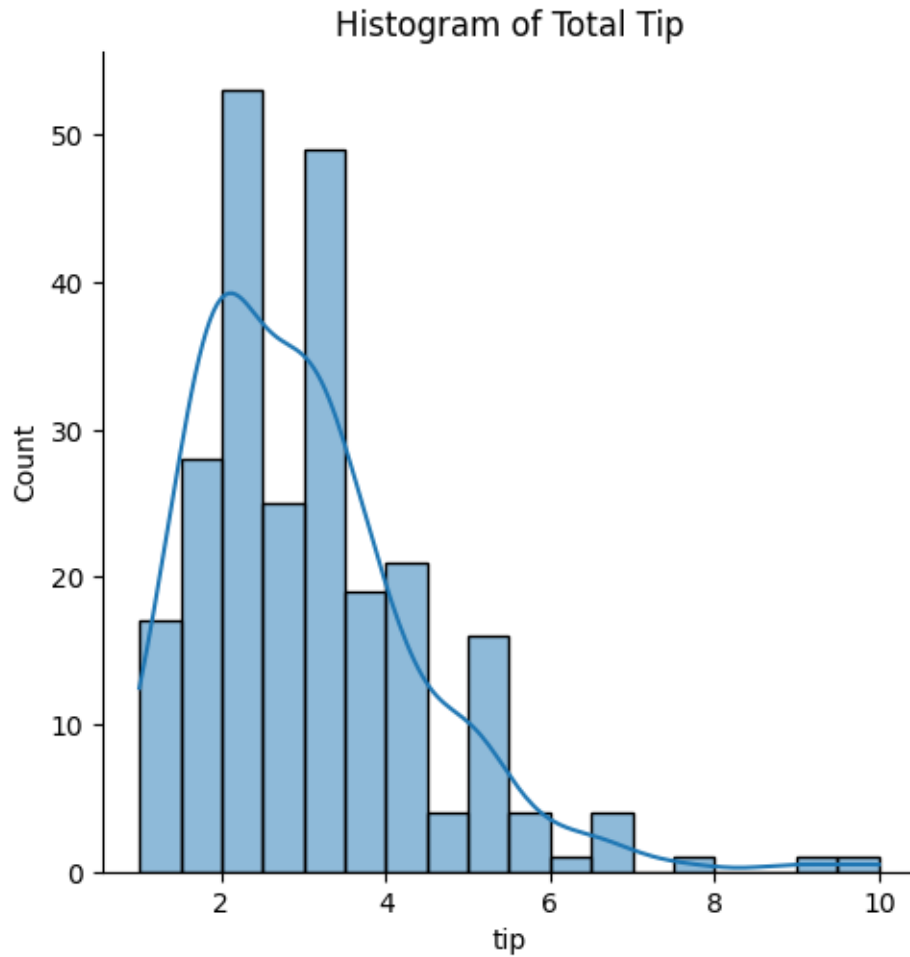
12.0.0.4 Creating a Histogram

After we have a general ‘feel’ for the data, it is often good to get a feel for the shape of the distribution of the data.

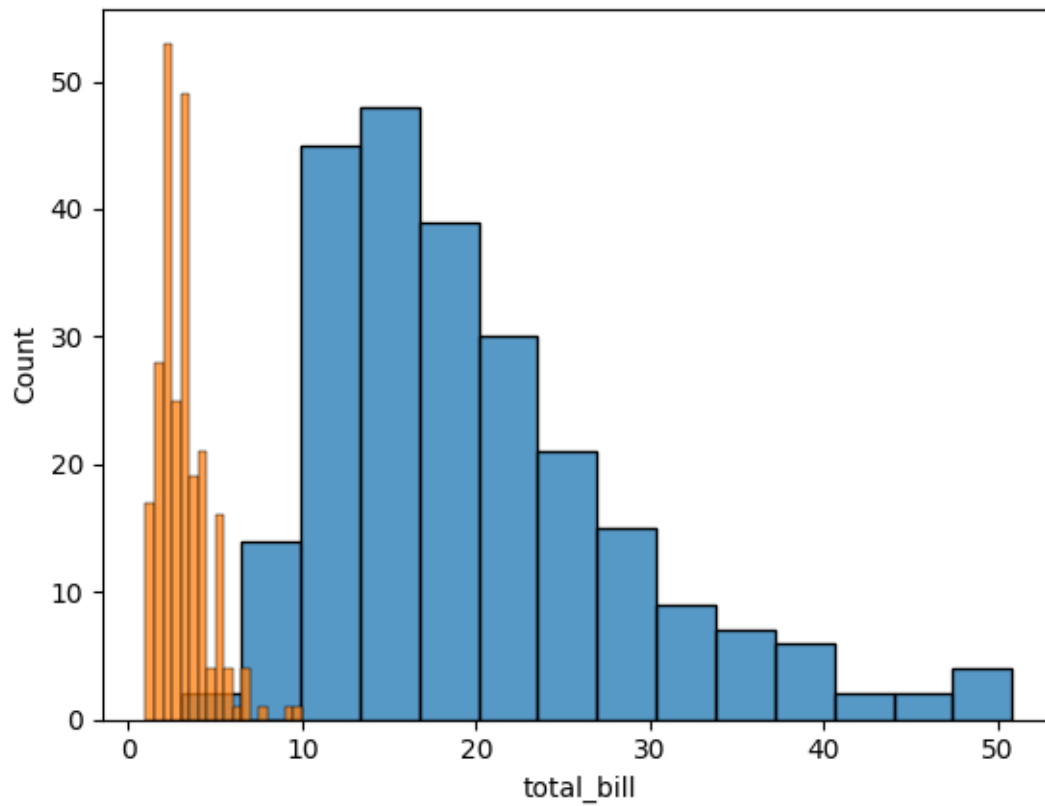
```
# Plot a histogram of the total bill
#kde --> whether or not to display a density plot
plot = sns.displot(tips_data["total_bill"], kde = False)
plt.title("Histogram of Total Bill")
plt.show()
```



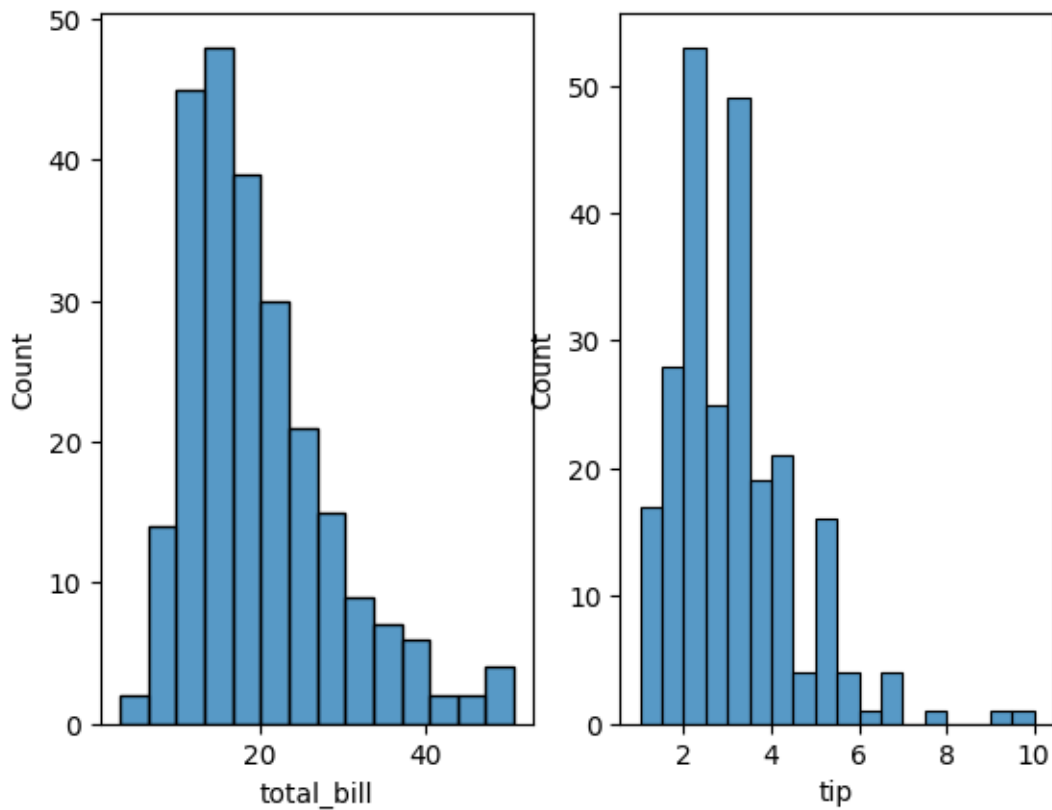
```
# Plot a histogram of the Tips only
sns.displot(tips_data["tip"], kde = True)
plt.title("Histogram of Total Tip")
plt.show()
```

```
# Plot a histogram of both the total bill and the tips'  
sns.histplot(tips_data["total_bill"], kde = False)  
sns.histplot(tips_data["tip"], kde = False)  
plt.show()
```



```
#alternative
fig, ax =plt.subplots(1,2)
sns.histplot(tips_data["total_bill"], kde = False, ax = ax[0])
sns.histplot(tips_data["tip"], kde = False, ax = ax[1])
plt.show()
```

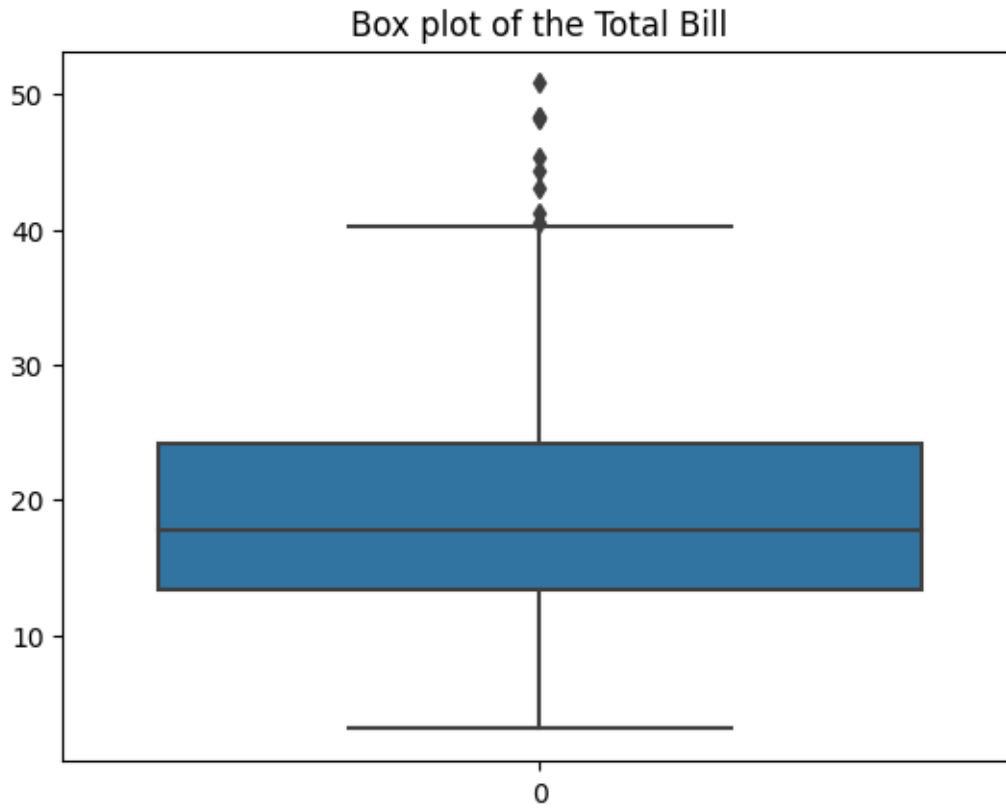


12.0.0.5 Creating a Boxplot

Boxplots do not show the shape of the distribution, but they can give us a better idea about the center and spread of the distribution as well as any potential outliers that may exist. Boxplots and Histograms often complement each other and help an analyst get more information about the data

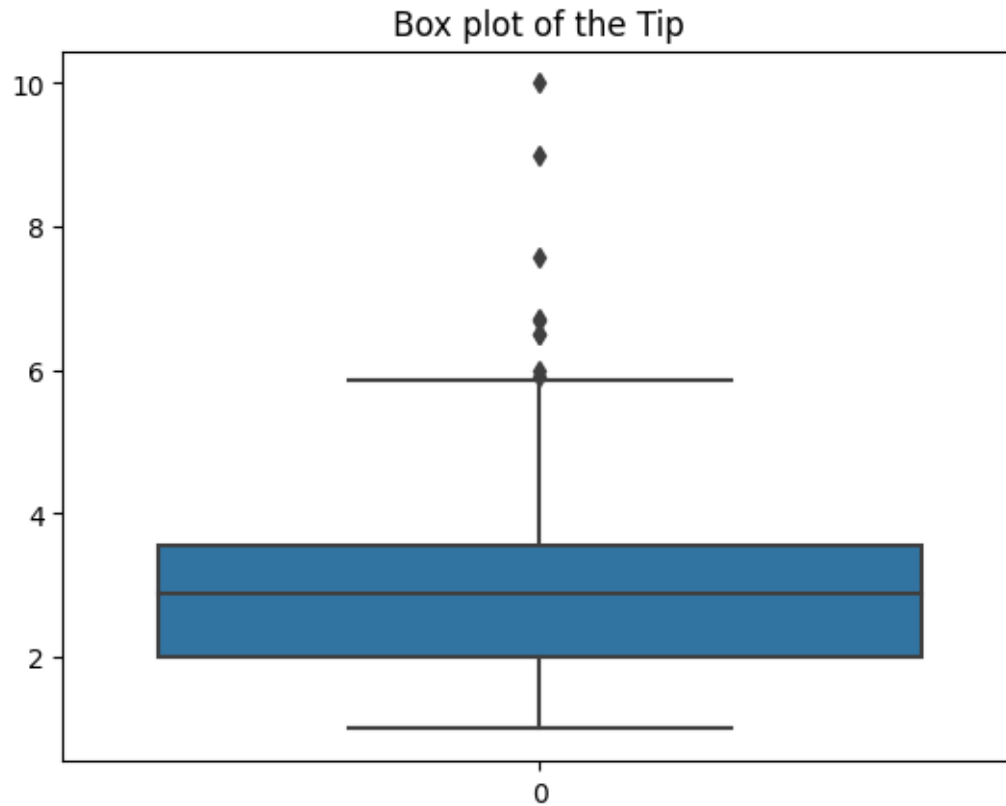
```
# Create a boxplot of the total bill amounts
sns.boxplot(tips_data["total_bill"])
plt.title("Box plot of the Total Bill")

plt.show()
```

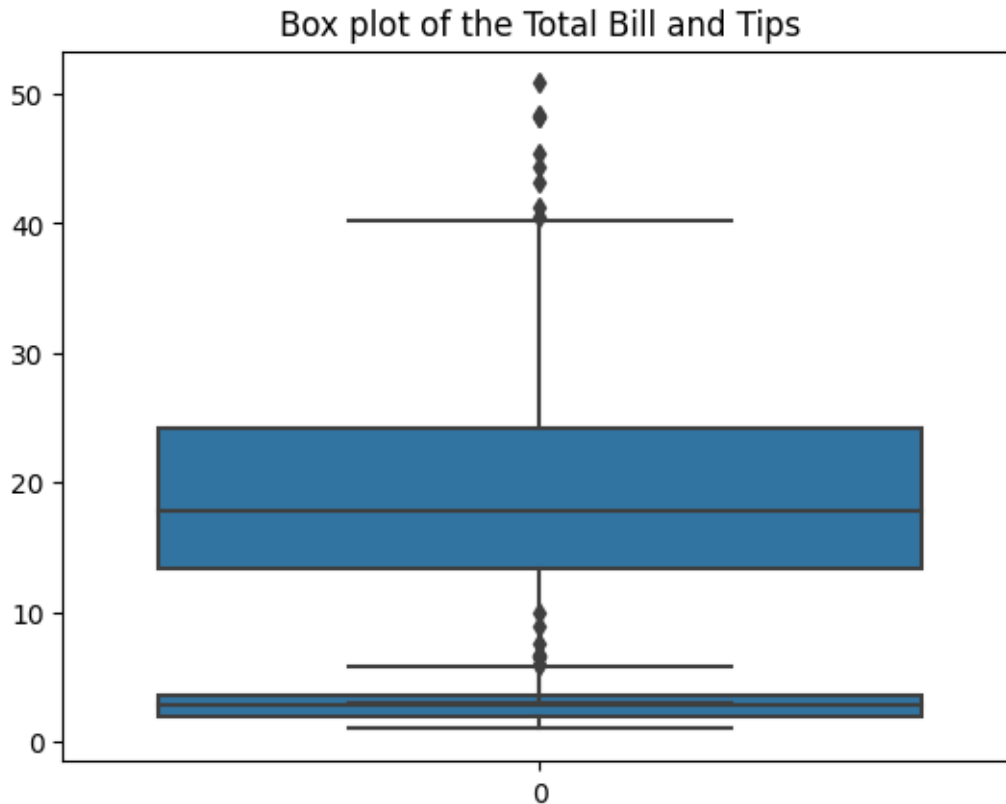


```
# Create a boxplot of the tips amounts
sns.boxplot(tips_data["tip"])
plt.title("Box plot of the Tip")

plt.show()
```



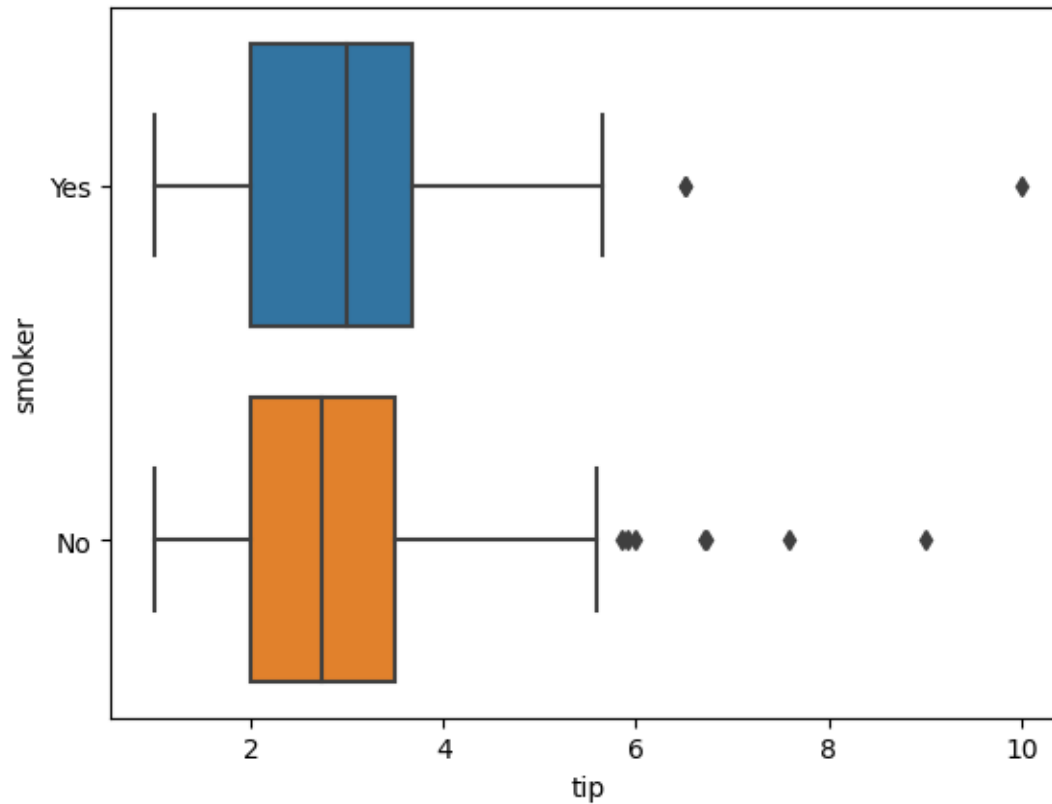
```
# Create a boxplot of the tips and total bill amounts - do not do it  
→ like this  
sns.boxplot(tips_data["total_bill"])  
plt.title("Box plot of the Total Bill and Tips")  
sns.boxplot(tips_data["tip"])  
  
plt.show()
```



12.0.0.6 Creating Histograms and Boxplots Plotted by Groups

While looking at a single variable is interesting, it is often useful to see how a variable changes in response to another. Using graphs, we can see if there is a difference between the tipping amounts of smokers vs. non-smokers, if tipping varies according to the time of the day, or we can explore other trends in the data as well.

```
# Create a boxplot and histogram of the tips grouped by smoking
↪ status
# x = what I am trying to plot
# y = what I am going to be grouping by
sns.boxplot(x = tips_data["tip"], y = tips_data["smoker"])
plt.show()
```

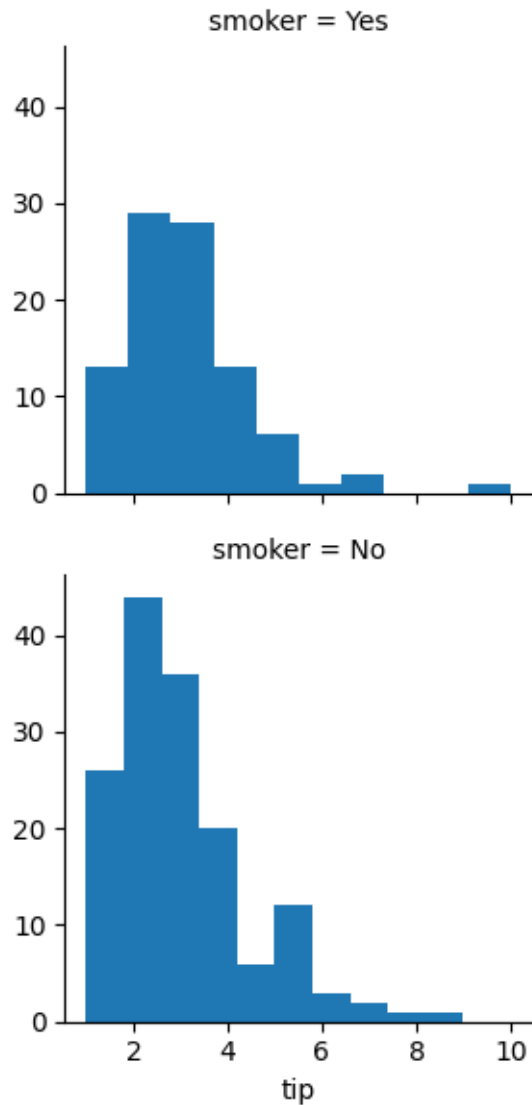


```
# Create histograms of the tips grouped by smoking status

#set up a facet grid by saying we want to have two similar boxes for
↳ our two smoking categories
g = sns.FacetGrid(tips_data, row = "smoker")

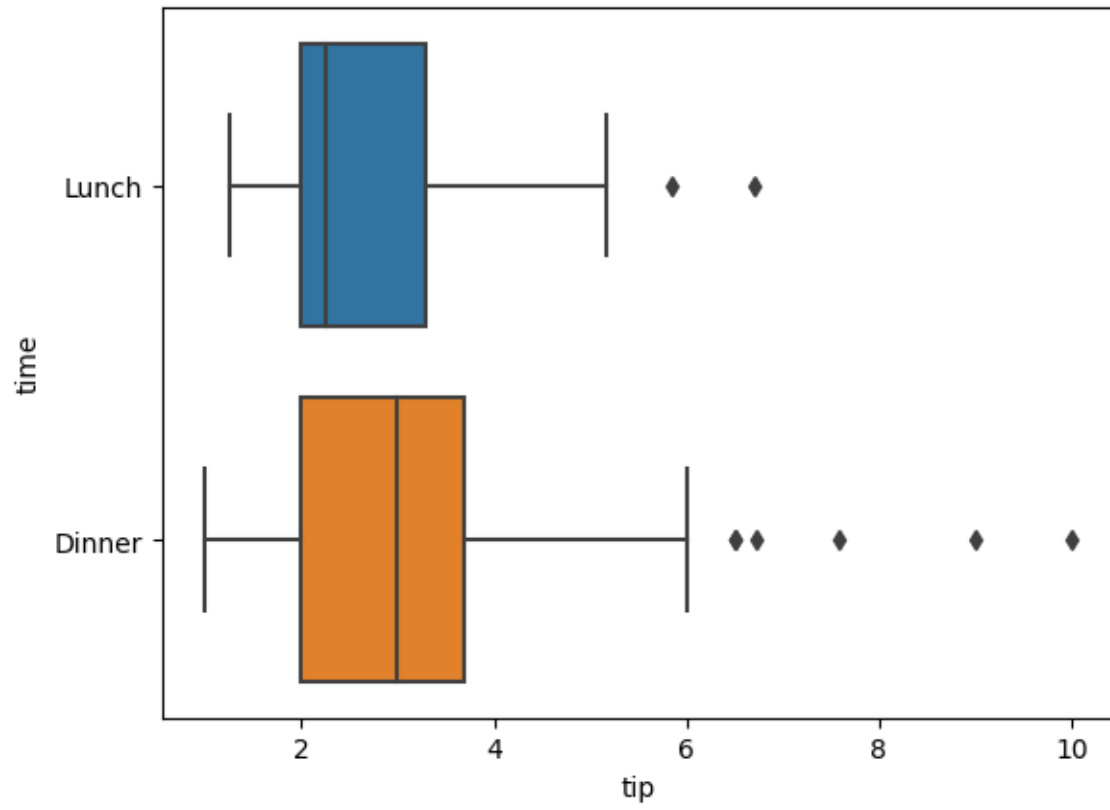
#the map fct allows us to take the histogram feature of plt and map
↳ it across both smoking groups at the same time
g = g.map(plt.hist, "tip")

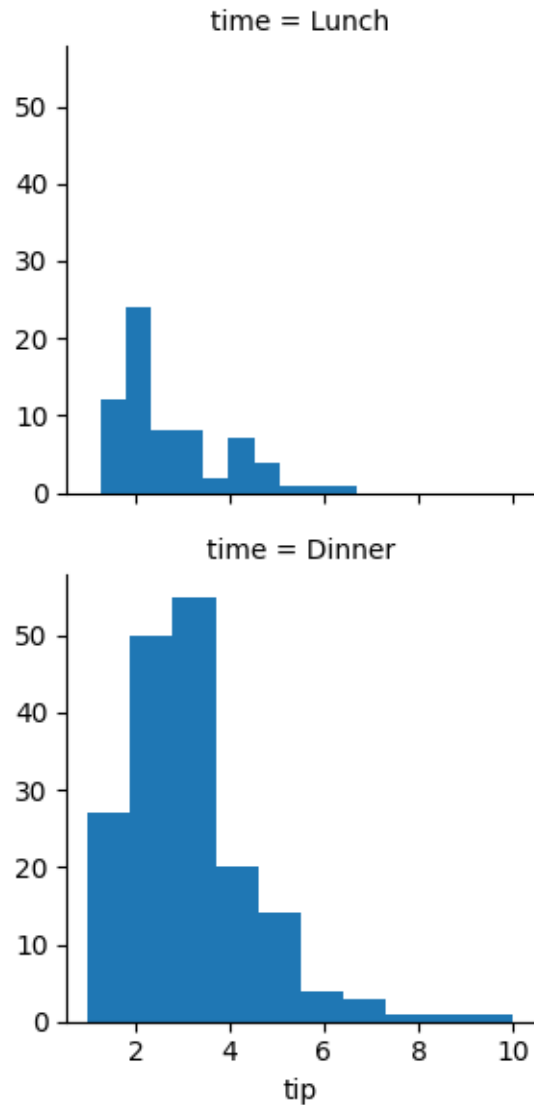
plt.show()
```



```
# Create a boxplot and histogram of the tips grouped by time of day
sns.boxplot(x = tips_data["tip"], y = tips_data["time"])

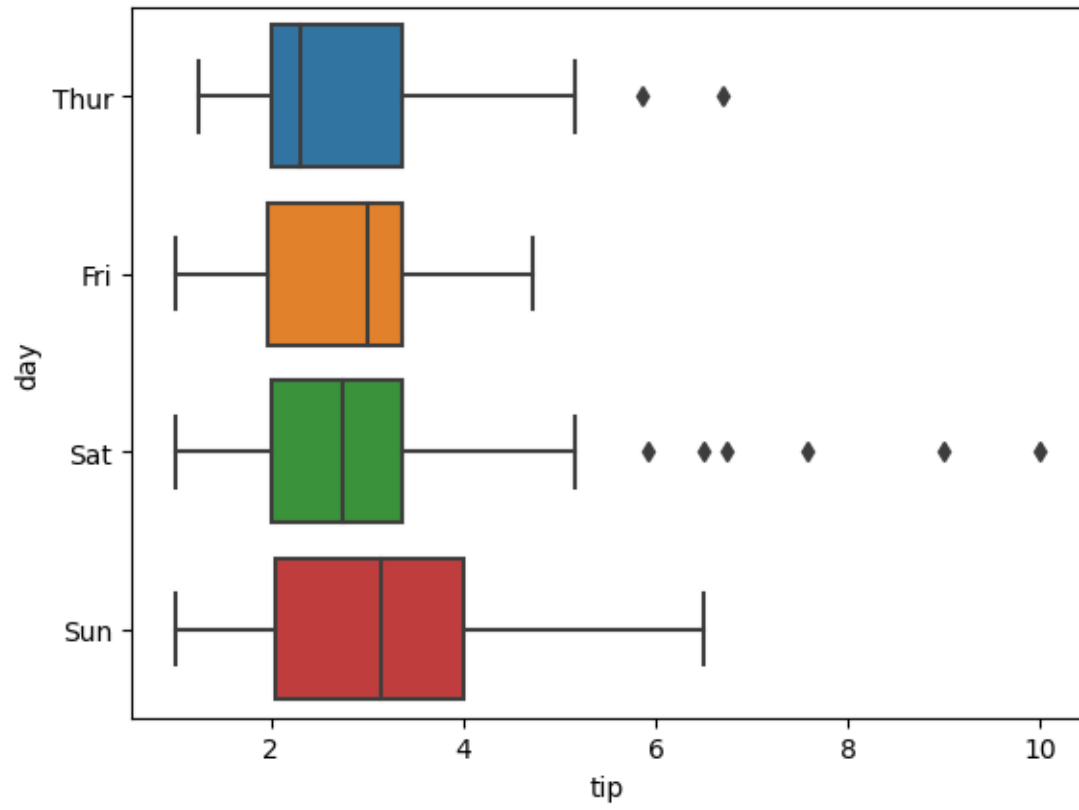
g = sns.FacetGrid(tips_data, row = "time")
g = g.map(plt.hist, "tip")
plt.show()
```

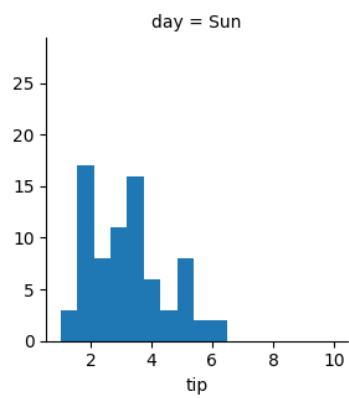
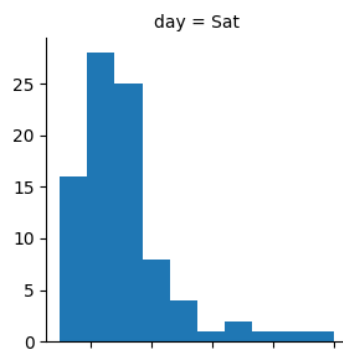
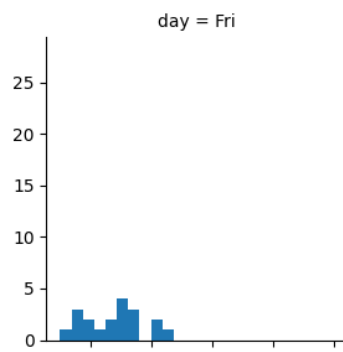
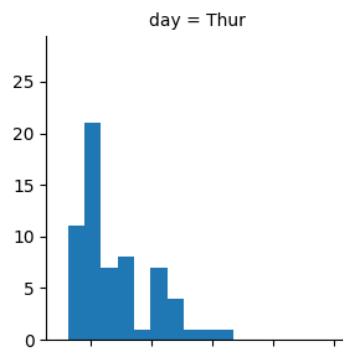





```
# Create a boxplot and histogram of the tips grouped by the day
sns.boxplot(x = tips_data["tip"], y = tips_data["day"])

g = sns.FacetGrid(tips_data, row = "day")
g = g.map(plt.hist, "tip")
plt.show()
```





13 Univariate data analyses - NHANES case study

Here we will demonstrate how to use Python and [Pandas](#) to perform some basic analyses with univariate data, using the 2015-2016 wave of the [NHANES](#) study to illustrate the techniques.

The following import statements make the libraries that we will need available. Note that in a Jupyter notebook, you should generally use the `%matplotlib inline` directive, which would not be used when running a script outside of the Jupyter environment.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

Next we will load the NHANES data from a file.

```
da = pd.read_csv("../data/nhanes_2015_2016.csv")
da.head()
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCH
0	83732	1.0	NaN	1.0	1	1	62	3	1.0
1	83733	1.0	NaN	6.0	1	1	53	3	2.0
2	83734	1.0	NaN	NaN	1	1	78	3	1.0
3	83735	2.0	1.0	1.0	2	2	56	3	1.0
4	83736	2.0	1.0	1.0	2	2	42	4	1.0

13.0.1 Frequency tables

The `value_counts` method can be used to determine the number of times that each distinct value of a variable occurs in a data set. In statistical terms, this is the “frequency distribution” of the variable. Below we show the frequency distribution of the `DMDEDUC2` variable, which is a variable that reflects a person’s level of educational attainment. The `value_counts` method produces a table with two columns. The first column contains all distinct observed values for the variable. The second column contains the number of times each of these values occurs. Note that the table returned by `value_counts` is actually a Pandas data frame, so can be further processed using any Pandas methods for working with data frames.

The numbers 1, 2, 3, 4, 5, 9 seen below are integer codes for the 6 possible non-missing values of the `DMDEDUC2` variable. The meaning of these codes is given in the NHANES codebook located [here](#), and will be discussed further below. This table shows, for example, that 1621 people in the data file have `DMDEDUC2`=4, which indicates that the person has completed some college, but has not graduated with a four-year degree.

```
da.DMDEDUC2.value_counts()
```

4.0	1621
5.0	1366
3.0	1186
1.0	655
2.0	643
9.0	3

Name: DMDEDUC2, dtype: int64

Note that **the `value_counts` method excludes missing values**. We confirm this below by adding up the number of observations with a `DMDEDUC2` value equal to 1, 2, 3, 4, 5, or 9 (there are 5474 such rows), and comparing this to the total number of rows in the data set, which is 5735. This tells us that there are $5735 - 5474 = 261$ missing values for this variable (other variables may have different numbers of missing values).

```
print(da.DMDEDUC2.value_counts().sum())
print(1621 + 1366 + 1186 + 655 + 643 + 3) # Manually sum the
↪ frequencies
print(da.shape)
```

```
5474
5474
(5735, 28)
```

Another way to obtain this result is to locate all the null (missing) values in the data set using the `isnull` Pandas function, and count the number of such locations.

```
pd.isnull(da.DMDEDUC2).sum()
```

```
261
```

13.0.1.1 Replace naming in a column

In some cases it is useful to `replace` integer codes with a text label that reflects the code's meaning. Below we create a new variable called 'DMDEDUC2x' that is recoded with text labels, then we generate its frequency distribution.

```
da["DMDEDUC2x"] = da.DMDEDUC2.replace({1: "<9", 2: "9-11", 3:
↪ "HS/GED", 4: "Some college/AA", 5: "College",
7: "Refused", 9: "Don't
↪ know"})

da.DMDEDUC2x.value_counts()
```

```
Some college/AA    1621
College            1366
HS/GED             1186
<9                 655
9-11               643
Don't know         3
Name: DMDEDUC2x, dtype: int64
```

We will also want to have a relabeled version of the gender variable, so we will construct that now as well. We will follow a convention here of appending an 'x' to the end of a categorical variable's name when it has been recoded from numeric to string (text) values.

```
da["RIAGENDRx"] = da.RIAGENDR.replace({1: "Male", 2: "Female"})

da["RIAGENDRx"].value_counts()
```

```
Female    2976
Male      2759
Name: RIAGENDRx, dtype: int64
```

For many purposes it is more relevant to consider the proportion of the sample with each of the possible category values, rather than the number of people in each category. We can do this as follows:

```
x = da.DMDEDUC2x.value_counts() # x is just a name to hold this
    ↪ value temporarily
x / x.sum() * 100
```

```
Some college/AA    29.612715
College            24.954330
HS/GED             21.666058
<9                 11.965656
9-11               11.746438
Don't know         0.054805
Name: DMDEDUC2x, dtype: float64
```

13.0.1.2 Replace NAs with another category

In some cases we will want to treat the missing response category as another category of observed response, rather than ignoring it when creating summaries. Below we create a new category called “Missing”, and assign all missing values to it using [fillna](#). Then we recalculate the frequency distribution. We see that 4.6% of the responses are missing.

```
da["DMDEDUC2x"] = da.DMDEDUC2x.fillna("Missing")
x = da.DMDEDUC2x.value_counts()
x / x.sum() * 100
```



```

Some college/AA    28.265039
College            23.818657
HS/GED            20.680035
<9                11.421099
9-11              11.211857
Missing           4.551003
Don't know        0.052310
Name: DMDDEDUC2x, dtype: float64

```

13.0.2 Numerical summaries

A quick way to get a set of numerical summaries for a quantitative variable is with the [describe](#) data frame method. Below we demonstrate how to do this using the body weight variable ([BMXWT](#)). As with many surveys, some data values are missing, so we explicitly drop the missing cases using the [dropna](#) method before generating the summaries.

```
da.BMXWT.dropna().describe()
```

```

count    5666.000000
mean      81.342676
std       21.764409
min       32.400000
25%       65.900000
50%       78.200000
75%       92.700000
max       198.900000
Name: BMXWT, dtype: float64

```

It's also possible to calculate individual summary statistics from one column of a data set. This can be done using Pandas methods, or with numpy functions:

```

x = da.BMXWT.dropna() # Extract all non-missing values of BMXWT into
↳ a variable called 'x'
print(x.mean()) # Pandas method
print(np.mean(x)) # Numpy function

print(x.median())
print(np.percentile(x, 50)) # 50th percentile, same as the median

```

```
print(np.percentile(x, 75)) # 75th percentile
print(x.quantile(0.75)) # Pandas method for quantiles, equivalent to
↪ 75th percentile
```

```
81.34267560889516
81.34267560889516
78.2
78.2
92.7
92.7
```

Next we look at frequencies for a systolic blood pressure measurement ([BPXSY1](#)). “BPX” here is the NHANES prefix for blood pressure measurements, “SY” stands for “systolic” blood pressure (blood pressure at the peak of a heartbeat cycle), and “1” indicates that this is the first of three systolic blood pressure measurements taken on a subject.

A person is generally considered to have pre-hypertension when their systolic blood pressure is between 120 and 139, or their diastolic blood pressure is between 80 and 89. Considering only the systolic condition, we can calculate the proportion of the NHANES sample who would be considered to have pre-hypertension.

```
np.mean((da.BPXSY1 >= 120) & (da.BPXSY2 <= 139)) # "&" means "and"
```

```
0.3741935483870968
```

Next we calculate the proportion of NHANES subjects who are pre-hypertensive based on diastolic blood pressure.

```
np.mean((da.BPXDI1 >= 80) & (da.BPXDI2 <= 89))
```

```
0.14803836094158676
```

Finally we calculate the proportion of NHANES subjects who are pre-hypertensive based on either systolic or diastolic blood pressure. Since some people are pre-hypertensive under both criteria, the proportion below is less than the sum of the two proportions calculated above.

Since the combined systolic and diastolic condition for pre-hypertension is somewhat complex, below we construct temporary variables ‘a’ and ‘b’ that hold the systolic and diastolic pre-hypertensive status separately, then combine them with a “logical or” to obtain the final status for each subject.

```
a = (da.BPXS1 >= 120) & (da.BPXS2 <= 139)
b = (da.BPXD1 >= 80) & (da.BPXD2 <= 89)
print(np.mean(a | b)) # "|" means "or"
```

0.43975588491717527

Blood pressure measurements are affected by a phenomenon called “white coat anxiety”, in which a subject’s blood pressure may be slightly elevated if they are nervous when interacting with health care providers. Typically this effect subsides if the blood pressure is measured several times in sequence. In NHANES, both systolic and diastolic blood pressure are measured three times for each subject (e.g. BPXS2 is the second measurement of systolic blood pressure). We can calculate the extent to which white coat anxiety is present in the NHANES data by looking at the mean difference between the first two systolic or diastolic blood pressure measurements.

```
print(np.mean(da.BPXS1 - da.BPXS2))
print(np.mean(da.BPXD1 - da.BPXD2))
```

0.6749860309182343

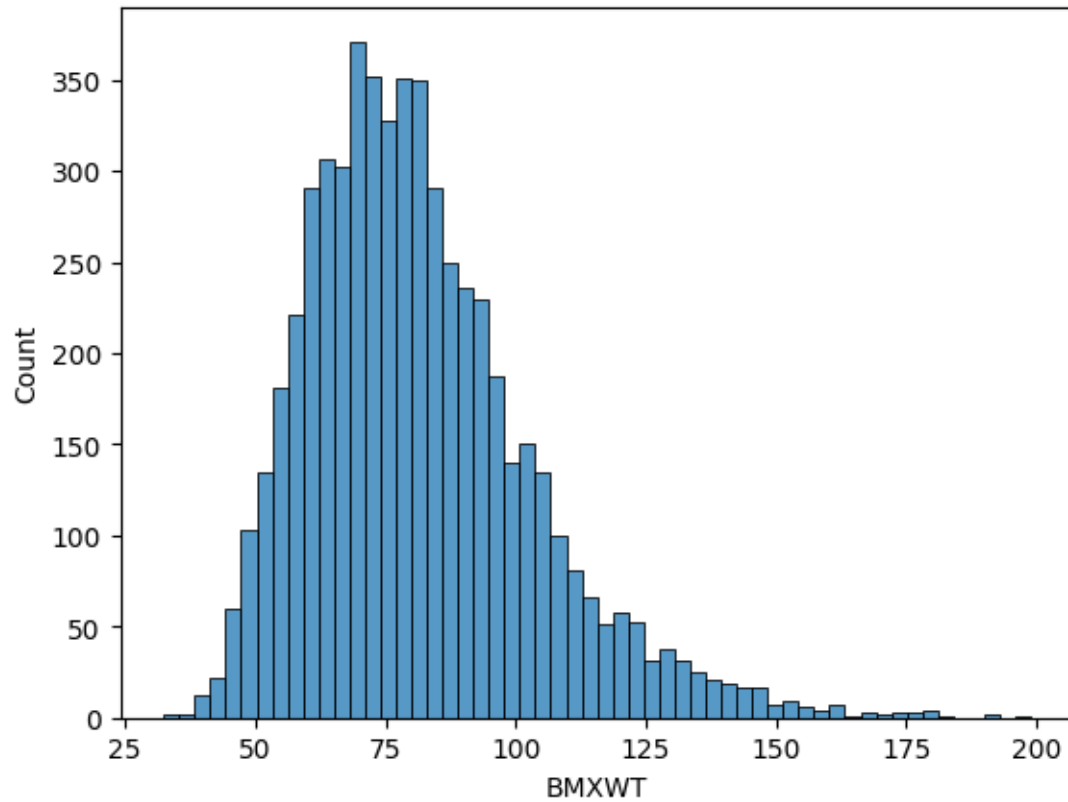
0.3490407897187558

13.0.3 Graphical summaries

Quantitative variables can be effectively summarized graphically. Below we see the distribution of body weight (in Kg), shown as a histogram. It is evidently right-skewed.

```
sns.histplot(da.BMXWT.dropna())
```

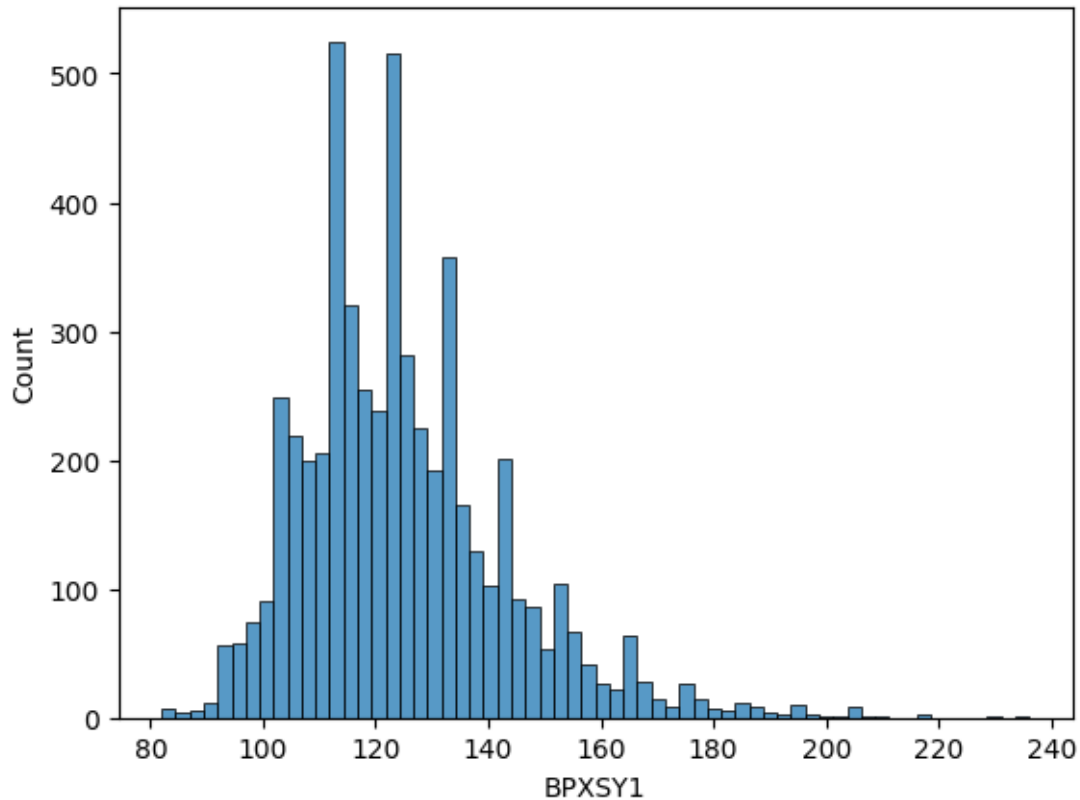
```
<AxesSubplot: xlabel='BMXWT', ylabel='Count'>
```



Next we look at the histogram of systolic blood pressure measurements. You can see that there is a tendency for the measurements to be rounded to the nearest 5 or 10 units.

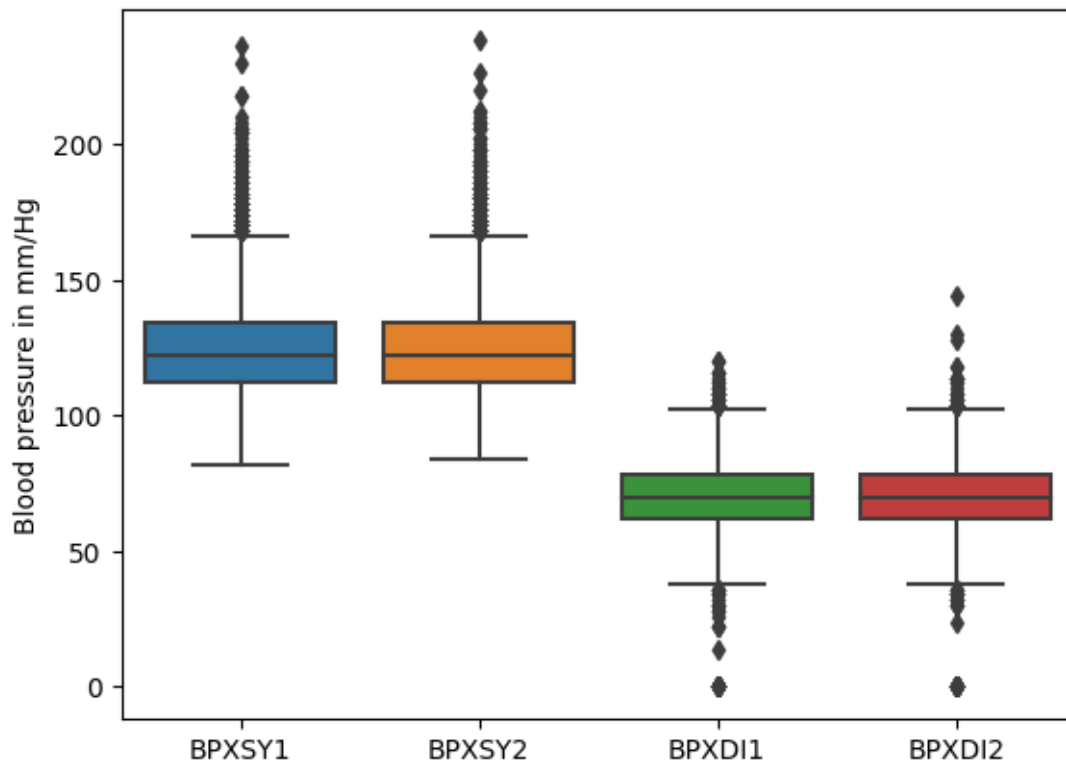
```
sns.histplot(da.BPXS1.dropna())
```

```
<AxesSubplot:xlabel='BPXS1', ylabel='Count'>
```



To compare several distributions, we can use side-by-side boxplots. Below we compare the distributions of the first and second systolic blood pressure measurements (BPXSY1, BPXSY2), and the first and second diastolic blood pressure measurements (BPXDI1, BPXDI2). As expected, diastolic measurements are substantially lower than systolic measurements. Above we saw that the second blood pressure reading on a subject tended on average to be slightly lower than the first measurement. This difference was less than 1 mm/Hg, so is not visible in the “marginal” distributions shown below.

```
bp = sns.boxplot(data=da[["BPXSY1", "BPXSY2", "BPXDI1", "BPXDI2"]])
_ = bp.set_ylabel("Blood pressure in mm/Hg")
```



13.0.4 Stratification

One of the most effective ways to get more information out of a dataset is to divide it into smaller, more uniform subsets, and analyze each of these “strata” on its own. We can then formally or informally compare the findings in the different strata. When working with human subjects, it is very common to stratify on demographic factors such as age, sex, and race.

To illustrate this technique, consider blood pressure, which is a value that tends to increase with age. To see this trend in the NHANES data, we can [partition](#) the data into age strata, and construct side-by-side boxplots of the systolic blood pressure (SBP) distribution within each stratum. Since age is a quantitative variable, we need to create a series of “bins” of similar SBP values in order to stratify the data. Each box in the figure below is a summary of univariate data within a specific population stratum (here defined by age).

```
da.RIDAGEYR.value_counts()
```

```

80    343
18    133
19    128
60    119
61    112
...
74     52
78     47
76     44
77     43
79     35

```

Name: RIDAGEYR, Length: 63, dtype: int64

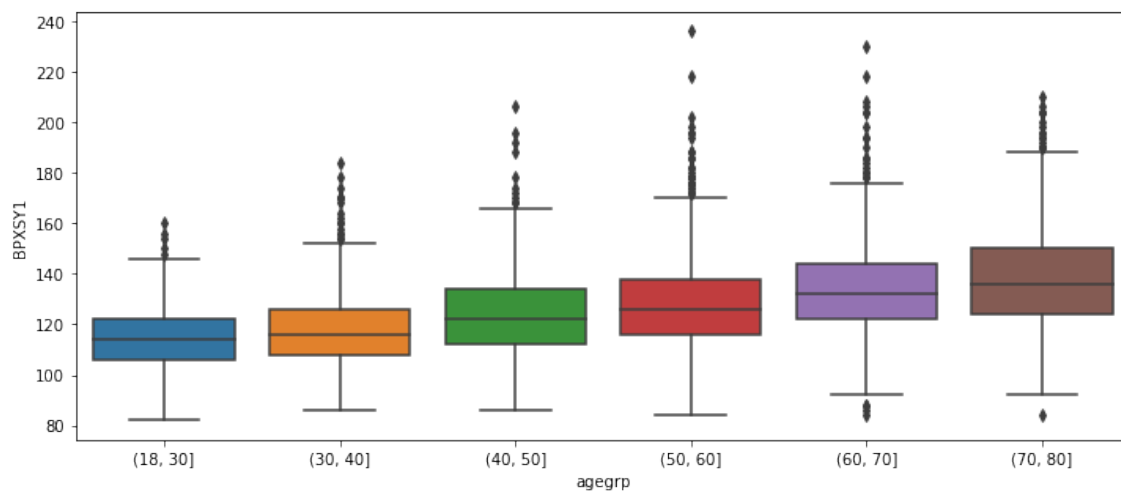
```

da["agegrp"] = pd.cut(da.RIDAGEYR, [18, 30, 40, 50, 60, 70, 80]) #
↳ Create age strata based on these cut points

plt.figure(figsize=(12, 5)) # Make the figure wider than default
↳ (12cm wide by 5cm tall)
sns.boxplot(x="agegrp", y="BPXSY1", data=da) # Make boxplot of
↳ BPXSY1 stratified by age group

```

<matplotlib.axes._subplots.AxesSubplot at 0x7f8388799ef0>



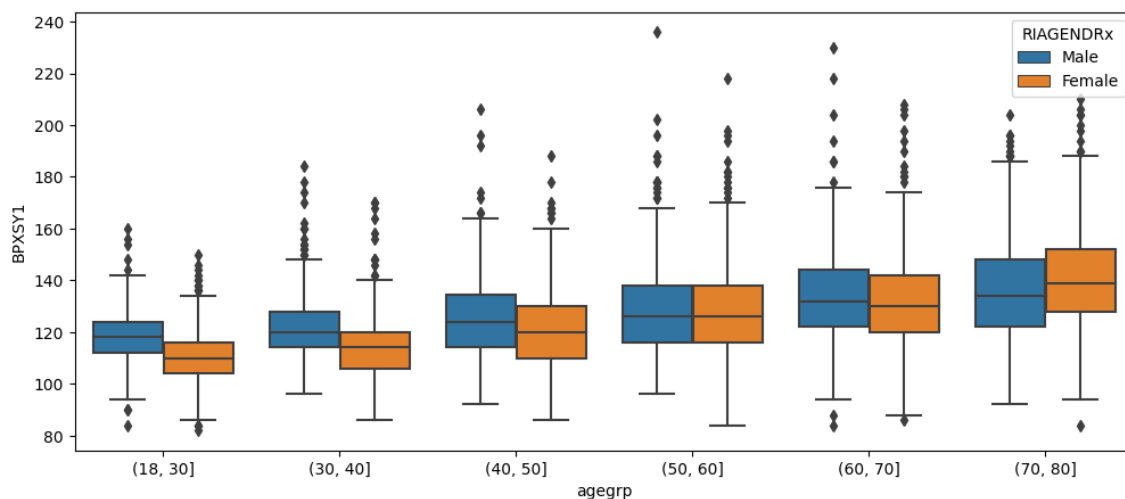
Taking this a step further, it is also the case that blood pressure tends to differ between women and men. While we could simply make two side-by-side boxplots to illustrate this contrast, it would be a bit odd to ignore age after already having established that it is strongly associated with blood pressure. Therefore, we will doubly stratify the data by gender and age.

We see from the figure below that within each gender, older people tend to have higher blood pressure than younger people. However within an age band, the relationship between gender and systolic blood pressure is somewhat complex – in younger people, men have substantially higher blood pressures than women of the same age. However for people older than 50, this relationship becomes much weaker, and among people older than 70 it appears to reverse. It is also notable that the variation of these distributions, reflected in the height of each box in the boxplot, increases with age.

```
da["agegrp"] = pd.cut(da.RIDAGEYR, [18, 30, 40, 50, 60, 70, 80])

plt.figure(figsize=(12, 5))
sns.boxplot(x="agegrp", y="BPXSY1", hue="RIAGENDRx", data = da)
```

<AxesSubplot:xlabel='agegrp', ylabel='BPXSY1'>



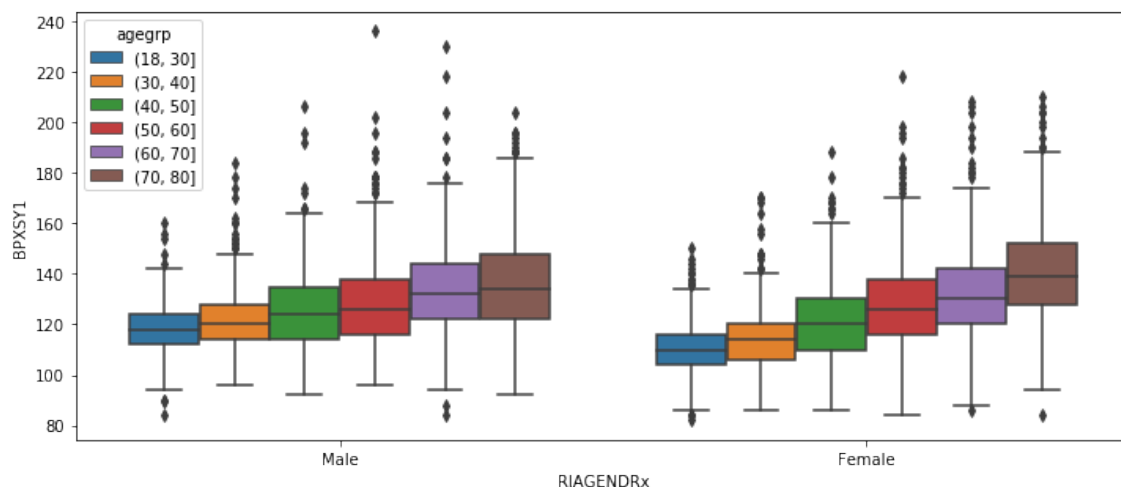
When stratifying on two factors (here age and gender), we can group the boxes first by age, and within age bands by gender, as above, or we can do the opposite – group first by

gender, and then within gender group by age bands. Each approach highlights a different aspect of the data.

```
da["agegrp"] = pd.cut(da.RIDAGEYR, [18, 30, 40, 50, 60, 70, 80])

plt.figure(figsize=(12, 5))
sns.boxplot(x="RIAGENDRx", y="BPXSY1", hue="agegrp", data=da)
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f838880ed68>



Stratification can also be useful when working with categorical variables. Below we look at the frequency distribution of educational attainment (“DMDEDUC2”) within 10-year age bands. While “some college” is the most common response in all age bands, up to around age 60 the second most common response is “college” (i.e. the person graduated from college with a four-year degree). However for people over 50, there are as many or more people with only high school or general equivalency diplomas (HS/GED) than there are college graduates.

Note on causality and confounding: An important role of statistics is to aid researchers in identifying causes underlying observed differences. Here we have seen differences in both blood pressure and educational attainment based on age. It is plausible that aging directly causes blood pressure to increase. But in the case of educational attainment, this is actually a “birth cohort effect”. NHANES is a cross sectional survey (all data for one wave were collected at a single point in time). People who were, say, 65 in 2015 (when these

data were collected), were college-aged around 1970, while people who were in their 20's in 2015 were college-aged in around 2010 or later. Over the last few decades, it has become much more common for people to at least begin a college degree than it was in the past. Therefore, younger people as a group have higher educational attainment than older people as a group. As these young people grow older, the cross sectional relationship between age and educational attainment will change.

```
da.groupby("agegrp")["DMDEDUC2x"].value_counts()
```

agegrp	DMDEDUC2x	
(18, 30]	Some college/AA	364
	College	278
	HS/GED	237
	Missing	128
	9-11	99
	<9	47
(30, 40]	Some college/AA	282
	College	264
	HS/GED	182
	9-11	111
	<9	93
(40, 50]	Some college/AA	262
	College	260
	HS/GED	171
	9-11	112
	<9	98
(50, 60]	Some college/AA	258
	College	220
	HS/GED	220
	9-11	122
	<9	104
(60, 70]	Some college/AA	238
	HS/GED	192
	College	188
	<9	149
(70, 80]	9-11	111
	Some college/AA	217
	HS/GED	184
	<9	164
	College	156

```

          9-11          88
          Don't know    3
Name: DMDEDUC2x, dtype: int64

```

We can also stratify jointly by age and gender to explore how educational attainment varies by both of these factors simultaneously. In doing this, it is easier to interpret the results if we [pivot](#) the education levels into the columns, and normalize the counts so that they sum to 1. After doing this, the results can be interpreted as proportions or probabilities. One notable observation from this table is that for people up to age around 60, women are more likely to have graduated from college than men, but for people over aged 60, this relationship reverses.

```

# Eliminate rare/missing values
dx = da.loc[~da.DMDEDUC2x.isin(["Don't know", "Missing"]), :]

#group data
dx = dx.groupby(["agegrp", "RIAGENDRx"])["DMDEDUC2x"]
dx = dx.value_counts()
dx.head()

```

```

agegrp    RIAGENDRx    DMDEDUC2x
(18, 30]  Female      Some college/AA    207
          Female      College            156
          Female      HS/GED            119
          Female      9-11              44
          Female      <9                27
Name: DMDEDUC2x, dtype: int64

```

```

dx = dx.unstack() # Restructure the results from 'long' to 'wide'
dx.head()

```

		DMDEDUC2x	9-11	<9	College	HS/GED	Some college/AA
agegrp	RIAGENDRx						
(18, 30]	Female		44	27	156	119	207
	Male		55	20	122	118	157
(30, 40]	Female		42	46	149	78	159
	Male		69	47	115	104	123

	DMDEDUC2x	9-11	<9	College	HS/GED	Some college/AA
agegrp	RIAGENDRx					
(40, 50]	Female	55	53	150	87	157

```
# Normalize within each stratum to get proportions
dx = dx.apply(lambda x: x/x.sum(), axis=1)
dx.head()
```

	DMDEDUC2x	9-11	<9	College	HS/GED	Some college/AA
agegrp	RIAGENDRx					
(18, 30]	Female	0.079566	0.048825	0.282098	0.215190	0.374322
	Male	0.116525	0.042373	0.258475	0.250000	0.332627
(30, 40]	Female	0.088608	0.097046	0.314346	0.164557	0.335443
	Male	0.150655	0.102620	0.251092	0.227074	0.268559
(40, 50]	Female	0.109562	0.105578	0.298805	0.173307	0.312749

```
print(dx.to_string(float_format="%.3f")) # Limit display to 3
↳ decimal places
```

DMDEDUC2x		9-11	<9	College	HS/GED	Some college/AA
agegrp	RIAGENDRx					
(18, 30]	Female	0.080	0.049	0.282	0.215	0.374
	Male	0.117	0.042	0.258	0.250	0.333
(30, 40]	Female	0.089	0.097	0.314	0.165	0.335
	Male	0.151	0.103	0.251	0.227	0.269
(40, 50]	Female	0.110	0.106	0.299	0.173	0.313
	Male	0.142	0.112	0.274	0.209	0.262
(50, 60]	Female	0.117	0.102	0.245	0.234	0.302
	Male	0.148	0.123	0.231	0.242	0.256
(60, 70]	Female	0.118	0.188	0.195	0.206	0.293
	Male	0.135	0.151	0.233	0.231	0.249
(70, 80]	Female	0.105	0.225	0.149	0.240	0.281
	Male	0.113	0.180	0.237	0.215	0.255

14 Practice notebook for univariate analysis using NHANES data

This notebook will give you the opportunity to perform some univariate analyses on your own using the NHANES. These analyses are similar to what was done in the week 2 NHANES case study notebook.

You can enter your code into the cells that say “enter your code here”, and you can type responses to the questions into the cells that say “Type Markdown and LaTeX”.

Note that most of the code that you will need to write below is very similar to code that appears in the case study notebook. You will need to edit code from that notebook in small ways to adapt it to the prompts below.

To get started, we will use the same module imports and read the data in the same way as we did in the case study:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import statsmodels.api as sm
import numpy as np

da = pd.read_csv("../data/nhanes_2015_2016.csv")
da.head()
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCH
0	83732	1.0	NaN	1.0	1	1	62	3	1.0
1	83733	1.0	NaN	6.0	1	1	53	3	2.0
2	83734	1.0	NaN	NaN	1	1	78	3	1.0
3	83735	2.0	1.0	1.0	2	2	56	3	1.0
4	83736	2.0	1.0	1.0	2	2	42	4	1.0

14.1 Question 1

Relabel the marital status variable `DMDMARTL` to have brief but informative character labels. Then construct a frequency table of these values for all people, then for women only, and for men only. Then construct these three frequency tables using only people whose age is between 30 and 40.

```
# relabel column
da['DMDMARTLx'] = da['DMDMARTL'].replace({1:'Married', 2:'Widowed',
↪ 3:'Divorced', 4:'Separated', 5:'Never married', 6:'Living with
↪ partner', 77:'Refused', 99:'Unknown' }).fillna('Missing')
da['RIAGENDRx'] = da['RIAGENDR'].replace({1:'Male', 2:'Female'})

#create freq table
x = da['DMDMARTLx'].value_counts()
x / x.sum()*100
```

Married	48.474281
Never married	17.506539
Divorced	10.095902
Living with partner	9.189189
Widowed	6.904969
Missing	4.551003
Separated	3.243243
Refused	0.034874

Name: DMDMARTLx, dtype: float64

```
#freq table for women only
x = da[da['RIAGENDR']==2]['DMDMARTLx'].value_counts()
x / x.sum()*100
```

Married	43.783602
Never married	17.473118
Divorced	11.760753
Widowed	9.946237
Living with partner	8.803763
Missing	4.233871
Separated	3.965054

Refused 0.033602
Name: DMDMARTLx, dtype: float64

```
#freq table for male only  
x = da[da['RIAGENDR']==1]['DMDMARTLx'].value_counts()  
x / x.sum()*100
```

Married 53.533889
Never married 17.542588
Living with partner 9.604929
Divorced 8.300109
Missing 4.893077
Widowed 3.624502
Separated 2.464661
Refused 0.036245
Name: DMDMARTLx, dtype: float64

```
#freq table for all people, age 30-40  
age30_40 = da[(da['RIDAGEYR'] >= 30) & (da['RIDAGEYR'] <= 40)]  
x = age30_40['DMDMARTLx'].value_counts()  
x / x.sum()*100
```

Married 54.580897
Never married 21.150097
Living with partner 13.937622
Divorced 6.822612
Separated 2.923977
Widowed 0.487329
Refused 0.097466
Name: DMDMARTLx, dtype: float64

```
#freq table for females, age 30-40  
x = age30_40[age30_40['RIAGENDR']==2]['DMDMARTLx'].value_counts()  
x / x.sum()*100
```

Married	53.571429
Never married	21.804511
Living with partner	12.218045
Divorced	8.646617
Separated	3.383459
Widowed	0.375940

Name: DMDMARTLx, dtype: float64

```
#freq table for males, age 30-40
x = age30_40[age30_40['RIAGENDR']==1]['DMDMARTLx'].value_counts()
x / x.sum()*100
```

Married	55.668016
Never married	20.445344
Living with partner	15.789474
Divorced	4.858300
Separated	2.429150
Widowed	0.607287
Refused	0.202429

Name: DMDMARTLx, dtype: float64

Q1a. Briefly comment on some of the differences that you observe between the distribution of marital status between women and men, for people of all ages.

There are less married women and that seems to be due to more women being divorced

Q1b. Briefly comment on the differences that you observe between the distribution of marital status states for women between the overall population, and for women between the ages of 30 and 40.

More women between 30-40 are married compared to the whole population and this group as less rates of widowed women

Q1c. Repeat part b for the men.

More man in their 30-40 live with a partner

14.2 Question 2

Restricting to the female population, stratify the subjects into age bands no wider than ten years, and construct the distribution of marital status within each age band. Within each age band, present the distribution in terms of proportions that must sum to 1.

```
#subset df
females = da[da['RIAGENDR'] == 2].copy()

#stratify
females['agegr'] = pd.cut(females['RIDAGEYR'], [18, 30, 40, 50, 60,
↪ 70, 80])

#group data
df
↪ =females.groupby("agegr")["DMDMARTLx"].value_counts().unstack().fillna(0)

#normalize
df = df.apply(lambda x : x/x.sum() * 100, axis = 1)

df
```

DMDMARTLx	Divorced	Living with partner	Married	Missing	Never married	Refused	Separated
agegr							
(18, 30]	1.806240	18.719212	25.944171	9.195402	42.528736	0.000000	1.806240
(30, 40]	9.071730	12.025316	54.430380	0.000000	20.464135	0.000000	3.586170
(40, 50]	13.745020	7.370518	57.370518	0.000000	12.549801	0.000000	6.573029
(50, 60]	17.659574	6.808511	54.680851	0.000000	8.936170	0.212766	5.744974
(60, 70]	19.274376	4.308390	48.072562	0.000000	8.616780	0.000000	4.988000
(70, 80]	14.390244	0.731707	31.707317	0.000000	5.121951	0.000000	1.951745

Q2a. Comment on the trends that you see in this series of marginal distributions.

We see an increase in: divorce over age groups We see a decrease in the proportion of females living with a partner + women never married There is a big spike in marriages (up to 50%) from age group 18-30 to 30-40 and then a slow decline The largest group of widowed women is in the oldest age group

Q2b. Repeat the construction for males.

```

#subset df
males = da[da['RIAGENDR'] == 1].copy()

#stratify
males['agegr'] = pd.cut(males['RIDAGEYR'], [18, 30, 40, 50, 60, 70,
↪ 80])

#group data
df
↪ =males.groupby("agegr")["DMDMARTLx"].value_counts().unstack().fillna(0)

#normalize
df = df.apply(lambda x : x/x.sum() * 100, axis = 1)

df

```

DMDMARTLx	Divorced	Living with partner	Married	Missing	Never married	Refused	Sepa
agegr							
(18, 30]	0.367647	17.463235	19.117647	13.235294	48.161765	0.000000	1.28
(30, 40]	5.240175	15.720524	56.331878	0.000000	19.432314	0.218341	2.62
(40, 50]	8.478803	8.229426	70.324190	0.000000	9.725686	0.000000	2.74
(50, 60]	12.555066	7.488987	65.198238	0.000000	10.352423	0.000000	2.20
(60, 70]	12.585812	5.034325	66.590389	0.000000	8.695652	0.000000	3.20
(70, 80]	14.179104	2.238806	61.194030	0.000000	2.238806	0.000000	3.48

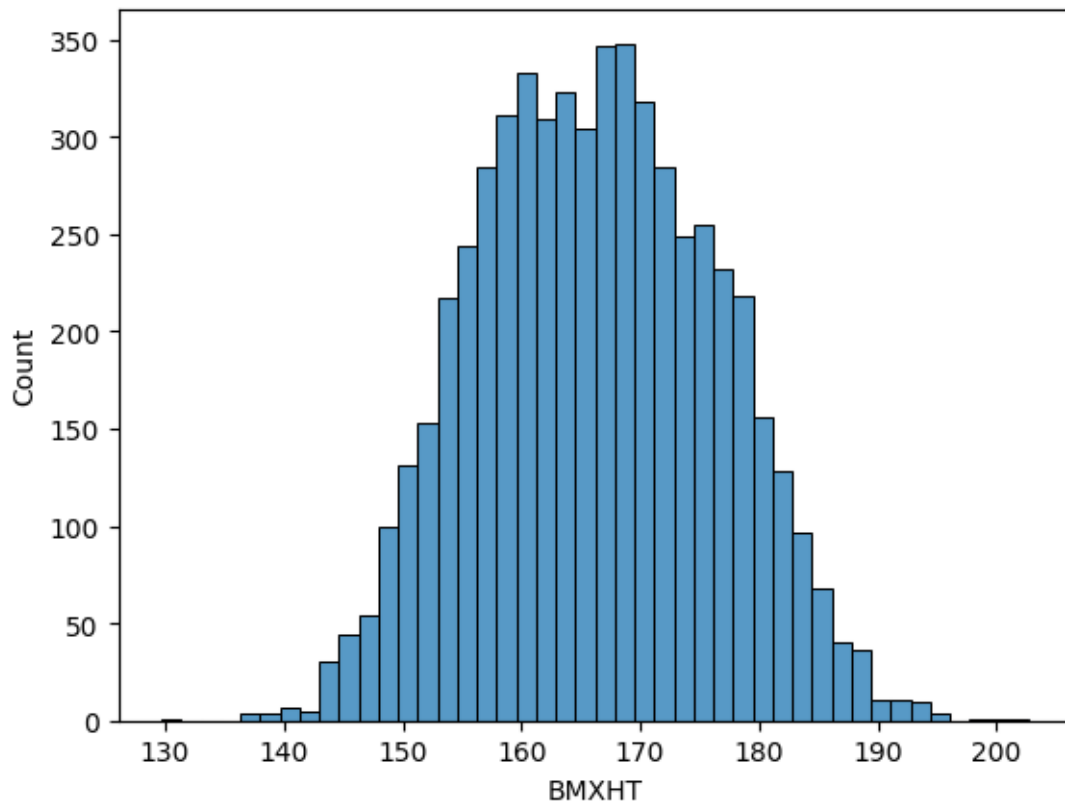
Q2c. Comment on any notable differences that you see when comparing these results for females and for males.

Increase in divorce over time Decrease of males living with a partner and males that never married Largest increase in married males in group 30-40 and then slow decrease (but not to levels as for females) Separated relatively constant Largest increase in widowed men in the last group (but small compared to females)

14.3 Question 3

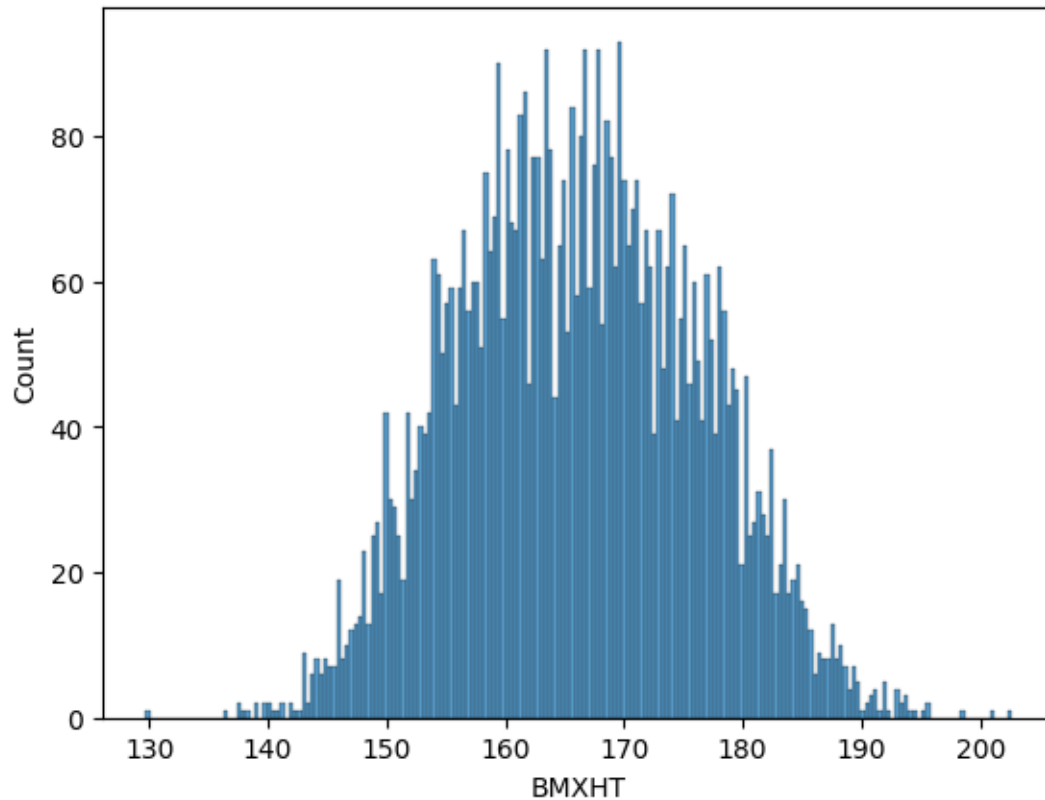
Construct a histogram of the distribution of heights using the BMXHT variable in the NHANES sample.

```
sns.histplot(da.BMXHT)
plt.show()
```



Q3a. Use the `bins` argument to `distplot` to produce histograms with different numbers of bins. Assess whether the default value for this argument gives a meaningful result, and comment on what happens as the number of bins grows excessively large or excessively small.

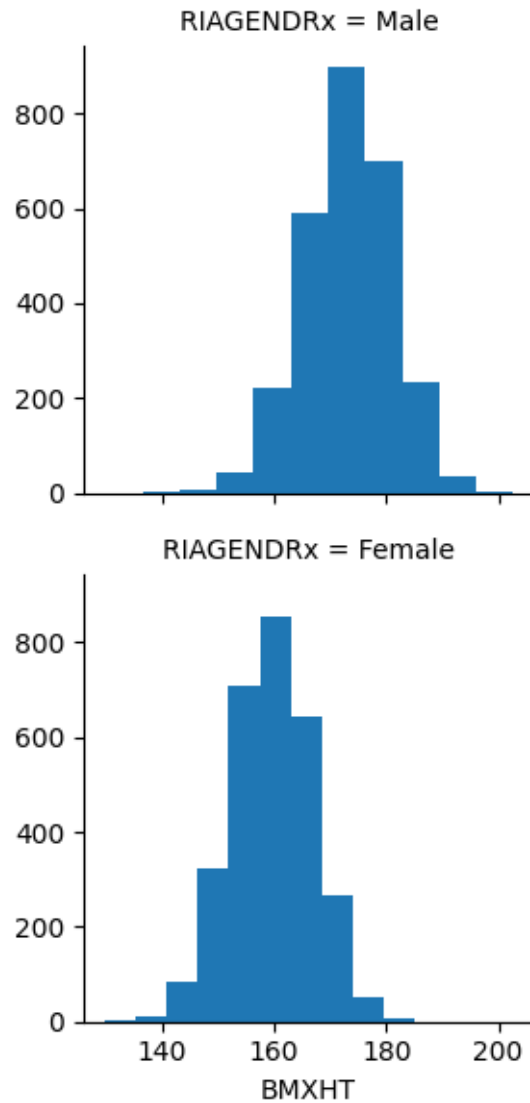
```
sns.histplot(da.BMXHT, bins = 200)
plt.show()
```



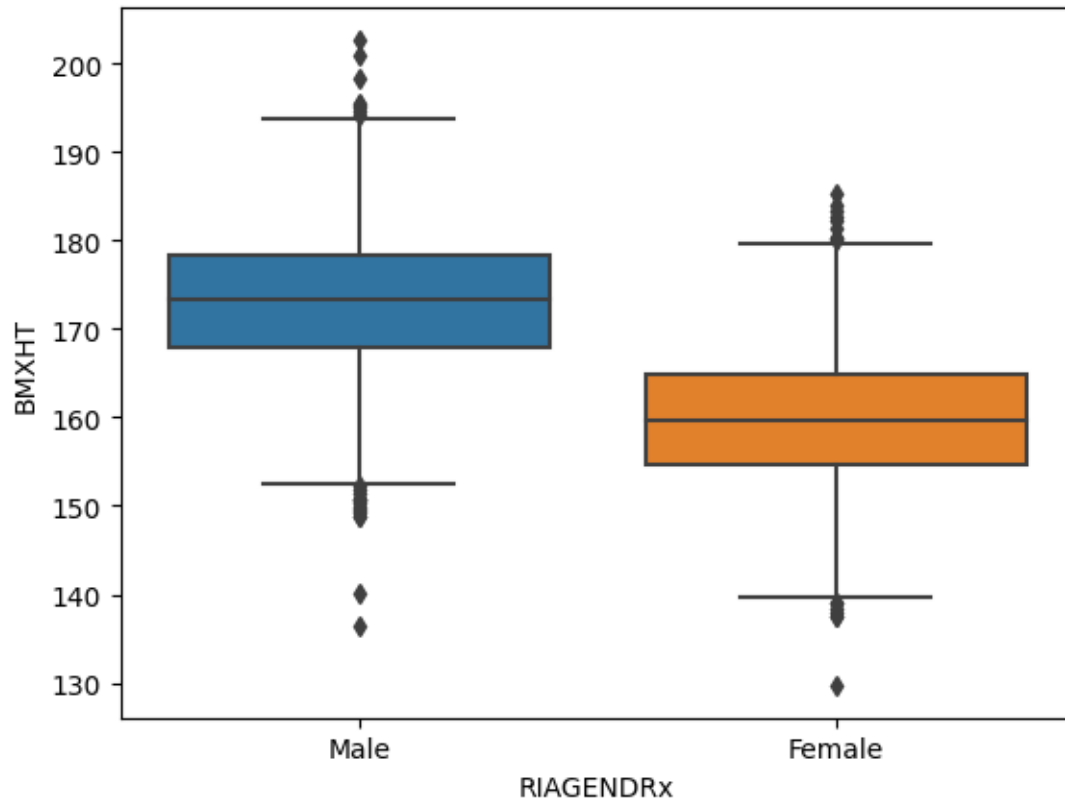
The value looks good

Q3b. Make separate histograms for the heights of women and men, then make a side-by-side boxplot showing the heights of women and men.

```
g = sns.FacetGrid(da, row = 'RIAGENDRx')
g = g.map(plt.hist, "BMXHT")
plt.show()
```



```
sns.boxplot(y = da['BMXHT'], x = da['RIAGENDRx'])  
plt.show()
```



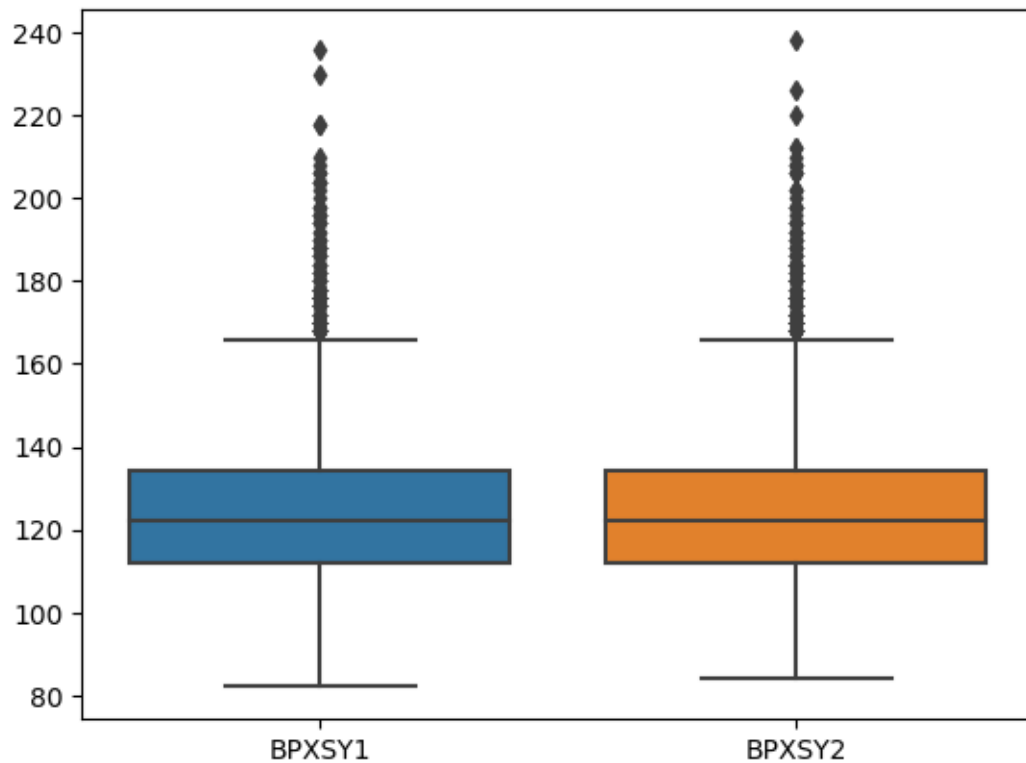
Q3c. Comment on what features, if any are not represented clearly in the boxplots, and what features, if any, are easier to see in the boxplots than in the histograms.

Males are larger than females (we can see this in both plots, however the median is easier to see in the boxplot). There are outliers on both ends (clearer in the boxplot)

14.4 Question 4

Make a boxplot showing the distribution of within-subject differences between the first and second systolic blood pressure measurements ([BPXSY1](#) and [BPXSY2](#)).

```
sns.boxplot(da[['BPXSY1', 'BPXSY2']])
plt.show()
```

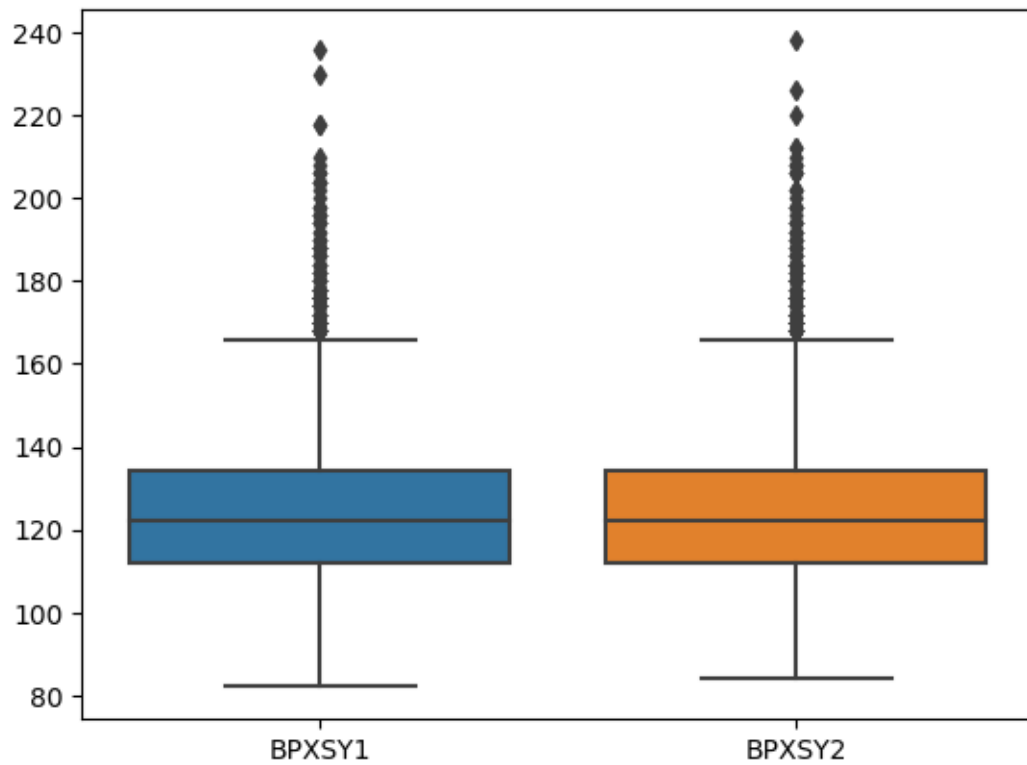


Q4a. What proportion of the subjects have a lower SBP on the second reading compared to the first?

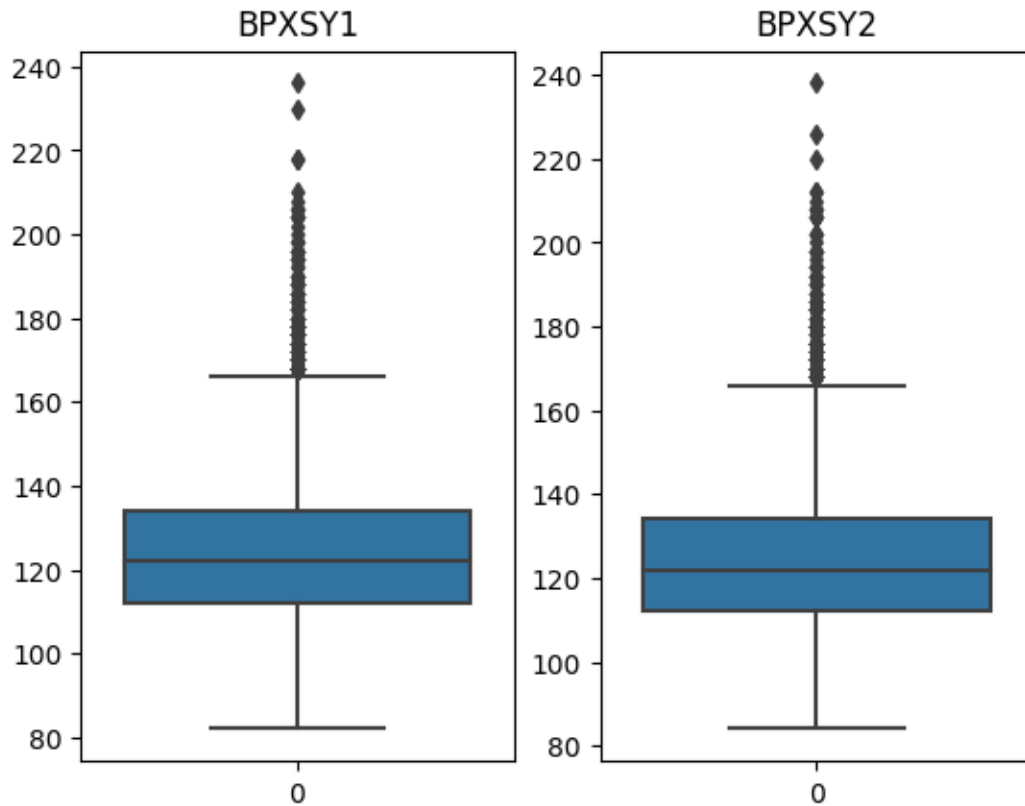
```
# insert your code here
```

Q4b. Make side-by-side boxplots of the two systolic blood pressure variables.

```
sns.boxplot(da[['BPXSY1', 'BPXSY2']])  
plt.show()
```



```
fig, ax =plt.subplots(1,2)
sns.boxplot(da['BPXSY1'], ax = ax[0]).set_title("BPXSY1")
sns.boxplot(da['BPXSY2'], ax = ax[1]).set_title("BPXSY2")
plt.show()
```

Q4c. Comment on the variation within either the first or second systolic blood pressure measurements, and the variation in the within-subject differences between the first and second systolic blood pressure measurements.

14.5 Question 5

Construct a frequency table of household sizes for people within each educational attainment category (the relevant variable is `DMDEDUC2`). Convert the frequencies to proportions.

```
dx = da.groupby(["DMDEDUC2"])["DMDHHSIZ"].value_counts().unstack()
dx = dx.apply(lambda x: x/x.sum(), axis=1)
dx
#print(dx.to_string(float_format="%.2f"))
```

DMDHHSIZ	1	2	3	4	5	6	7
DMDEDUC2							
1.0	0.109924	0.224427	0.146565	0.132824	0.148092	0.108397	0.129771
2.0	0.116641	0.222395	0.163297	0.152411	0.146190	0.113530	0.085537
3.0	0.152614	0.270658	0.171164	0.161889	0.109612	0.065767	0.068297
4.0	0.151141	0.268970	0.193091	0.169031	0.122147	0.050586	0.045034
5.0	0.142753	0.347731	0.193997	0.165447	0.095168	0.029283	0.025622
9.0	NaN	0.666667	NaN	NaN	0.333333	NaN	NaN

Q5a. Comment on any major differences among the distributions.

Q5b. Restrict the sample to people between 30 and 40 years of age. Then calculate the median household size for women and men within each level of educational attainment.

```
da[(da.RIDAGEYR >= 30) & (da.RIDAGEYR <= 40)].groupby(["DMDEDUC2",
↪ "RIAGENDR"])["DMDHHSIZ"].median()
```

```
DMDEDUC2  RIAGENDR
1.0        1        5.0
          2        5.0
2.0        1        4.5
          2        5.0
3.0        1        4.0
          2        5.0
4.0        1        4.0
          2        4.0
5.0        1        3.0
          2        3.0
```

Name: DMDHHSIZ, dtype: float64

14.6 Question 6

The participants can be clustered into “made variance units” (MVU) based on every combination of the variables [SDMVSTRA](#) and [SDMVPSU](#). Calculate the mean age ([RIDAGEYR](#)), height ([BMXHT](#)), and BMI ([BMXBMI](#)) for each gender ([RIAGENDR](#)), within each MVU, and report the ratio between the largest and smallest mean (e.g. for height) across the MVUs.

```
da.groupby(['SDMVSTRA', 'SDMVPSU', 'RIAGENDR']) \
    [['RIDAGEYR', 'BMXHT', 'BMXBMI']] \
    .mean().unstack()
```

SDMVSTRA	RIAGENDR SDMVPSU	RIDAGEYR		BMXHT		BMXBMI	
		1	2	1	2	1	2
119	1	47.861111	47.663265	172.741667	159.570408	26.958333	30.052041
	2	54.363636	52.987952	172.906818	159.244578	27.160465	27.849398
120	1	43.130000	43.636364	169.537755	155.402041	30.939175	32.419388
	2	45.219178	43.736111	173.075342	159.218056	27.727397	27.400000
121	1	46.750000	44.397959	172.177885	158.871579	29.416505	30.856842
	2	42.063158	44.376344	174.764516	160.229032	26.273118	26.470968
122	1	44.653061	42.897436	173.998969	161.315385	28.528866	29.447436
	2	44.320000	47.333333	170.332323	157.231111	25.744444	26.611111
123	1	47.829787	44.841121	174.315217	162.059615	29.231522	29.905769
	2	52.126582	46.457447	174.454430	160.476596	28.811392	30.641489
124	1	50.750000	51.664000	172.109009	158.788710	28.614414	29.533065
	2	48.245614	42.541667	174.291228	162.853521	27.714035	28.640845
125	1	55.165289	50.900901	173.631092	160.762385	29.727731	30.385321
	2	49.705882	51.660000	174.456863	160.021429	29.143564	28.564286
126	1	48.416667	46.229167	175.149398	160.387500	29.033333	31.262500
	2	48.666667	47.205882	174.713043	160.892000	29.039130	29.612121
127	1	53.137931	49.694444	171.545349	157.422430	31.062353	32.189720
	2	54.070588	51.486239	173.366667	159.022936	30.557831	30.770642
128	1	53.673267	55.638462	169.325000	156.339063	31.749000	32.303125
	2	45.822785	45.589744	172.400000	160.437179	26.835443	27.491026
129	1	43.922222	45.329787	171.094318	156.900000	26.493182	29.019149
	2	45.775510	43.500000	173.138298	161.034259	28.961702	29.429630
130	1	50.516854	47.810526	176.974157	161.977895	30.337079	30.700000
	2	50.535354	50.833333	175.061224	160.060577	29.237755	31.490385
131	1	53.140187	54.893617	175.610476	161.989362	28.259615	30.061702
	2	46.778846	45.000000	175.091346	161.673810	30.077885	32.984127
132	1	42.380435	43.210526	172.534066	161.508421	28.546154	29.848421
	2	49.038760	51.700000	172.809524	159.138281	28.966667	30.540625
133	1	44.054795	45.105882	171.509722	158.295122	27.495833	27.959259
	2	47.489796	47.063158	171.179167	158.627368	27.966667	29.000000

Q6a. Comment on the extent to which mean age, height, and BMI vary among the MVUs.

Q6b. Calculate the inter-quartile range (IQR) for age, height, and BMI for each gender and each MVU. Report the ratio between the largest and smallest IQR across the MVUs.

```
# insert your code here
```

Q6c. Comment on the extent to which the IQR for age, height, and BMI vary among the MVUs.

15 How to select dataframe subsets from multivariate data

```
import numpy as np
import pandas as pd

# Show all columns when looking at dataframe
pd.set_option('display.max_columns', 100)
```

```
# Download NHANES 2015-2016 data
df = pd.read_csv("../data/nhanes_2015_2016.csv")
```

```
df.head()
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCH
0	83732	1.0	NaN	1.0	1	1	62	3	1.0
1	83733	1.0	NaN	6.0	1	1	53	3	2.0
2	83734	1.0	NaN	NaN	1	1	78	3	1.0
3	83735	2.0	1.0	1.0	2	2	56	3	1.0
4	83736	2.0	1.0	1.0	2	2	42	4	1.0

15.0.1 Selecting columns

15.0.1.1 Selecting columns using list comprehensions

Keep only body measures columns, so only columns with “BMX” in the name

```
# get columns names
col_names = df.columns
```

```
col_names
```

```
Index(['SEQN', 'ALQ101', 'ALQ110', 'ALQ130', 'SMQ020', 'RIAGENDR', 'RIDAGEYR',  
      'RIDRETH1', 'DMDCITZN', 'DMDDEDUC2', 'DMDMARTL', 'DMDHHSIZ', 'WTINT2YR',  
      'SDMVPSU', 'SDMVSTRA', 'INDFMPIR', 'BPXSY1', 'BPXDI1', 'BPXSY2',  
      'BPXDI2', 'BMXWT', 'BMXHT', 'BMXBMI', 'BMXLEG', 'BMXARML', 'BMXARMC',  
      'BMXWAIST', 'HIQ210'],  
      dtype='object')
```

```
# One way to get the column names we want to keep is simply by  
→ copying from the above output and storing in a list  
keep = ['BMXWT', 'BMXHT', 'BMXBMI', 'BMXLEG', 'BMXARML', 'BMXARMC',  
        'BMXWAIST']
```

```
# Another way to get only column names that include 'BMX' is with  
→ list comprehension  
# [keep x for x in list if condition met]  
[column for column in col_names if 'BMX' in column]
```

```
['BMXWT', 'BMXHT', 'BMXBMI', 'BMXLEG', 'BMXARML', 'BMXARMC', 'BMXWAIST']
```

```
keep = [column for column in col_names if 'BMX' in column]
```

```
# use [] notation to keep columns  
df_BMX = df[keep]
```

```
df_BMX.head()
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
0	94.8	184.5	27.8	43.3	43.6	35.9	101.1
1	90.4	171.4	30.8	38.0	40.0	33.2	107.9
2	83.4	170.1	28.8	35.6	37.0	31.0	116.5
3	109.8	160.9	42.4	38.5	37.7	38.3	110.1

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
4	55.2	164.9	20.3	37.4	36.0	27.2	80.4

There are two methods for selecting by row and column.

15.0.1.2 Selecting columns using loc

link for pandas cheat sheets

- `df.loc[row labels or bool, col labels or bool]`
- `df.iloc[row int or bool, col int or bool]`

From pandas docs:

- ☐ column indexing
 - `.loc` is primarily label based, but may also be used with a boolean array.
- `.iloc` is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

```
df.loc[:, keep].head()
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
0	94.8	184.5	27.8	43.3	43.6	35.9	101.1
1	90.4	171.4	30.8	38.0	40.0	33.2	107.9
2	83.4	170.1	28.8	35.6	37.0	31.0	116.5
3	109.8	160.9	42.4	38.5	37.7	38.3	110.1
4	55.2	164.9	20.3	37.4	36.0	27.2	80.4

15.0.1.3 Selecting columns using numpy

```
index_bool = np.isin(df.columns, keep)
```

```
index_bool
```

```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, True, True, True, True, True, True, True, True,
       False])
```

```
df.iloc[:,index_bool].head() # Indexing with boolean list
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
0	94.8	184.5	27.8	43.3	43.6	35.9	101.1
1	90.4	171.4	30.8	38.0	40.0	33.2	107.9
2	83.4	170.1	28.8	35.6	37.0	31.0	116.5
3	109.8	160.9	42.4	38.5	37.7	38.3	110.1
4	55.2	164.9	20.3	37.4	36.0	27.2	80.4

15.0.2 Selection by conditions

```
# Lets only look at rows who 'BMXWAIST' is larger than the median
# get the median of 'BMXWAIST'
waist_median = pd.Series.median(df_BMX['BMXWAIST'])
```

```
#alternative code
df_BMX['BMXWAIST'].median()
```

98.3

```
waist_median
```

98.3

```
#subset the dataframe
df_BMX[df_BMX['BMXWAIST'] > waist_median].head()
```


	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
0	94.8	184.5	27.8	43.3	43.6	35.9	101.1
1	90.4	171.4	30.8	38.0	40.0	33.2	107.9
2	83.4	170.1	28.8	35.6	37.0	31.0	116.5
3	109.8	160.9	42.4	38.5	37.7	38.3	110.1
9	108.3	179.4	33.6	46.0	44.1	38.5	116.0

```
# Lets add another condition, that 'BMXLEG' must be less than 32
condition1 = df_BMX['BMXWAIST'] > waist_median
condition2 = df_BMX['BMXLEG'] < 32

# Subset the data using [] method
# Note: can't use 'and' instead of '&'
df_BMX[condition1 & condition2].head()
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
15	80.5	150.8	35.4	31.6	32.7	33.7	113.5
27	75.6	145.2	35.9	31.0	33.1	36.0	108.0
39	63.7	147.9	29.1	26.0	34.0	31.5	110.0
52	105.9	157.7	42.6	29.2	35.0	40.7	129.1
55	77.5	148.3	35.2	30.5	34.0	34.4	107.6

```
# Alternative using df.loc[] method
# note that the conditiona are describing the rows to keep
df_BMX.loc[condition1 & condition2, :].head()
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
15	80.5	150.8	35.4	31.6	32.7	33.7	113.5
27	75.6	145.2	35.9	31.0	33.1	36.0	108.0
39	63.7	147.9	29.1	26.0	34.0	31.5	110.0
52	105.9	157.7	42.6	29.2	35.0	40.7	129.1
55	77.5	148.3	35.2	30.5	34.0	34.4	107.6

```
# Lets make a small dataframe and give it a new index so can more
↳ clearly see the differences between .loc and .iloc
# If you use different years than 2015-2016, this my give an error.
↳ Why?
tmp = df_BMX.loc[condition1 & condition2, :].head()
tmp.index = ['a', 'b', 'c', 'd', 'e']
tmp
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
a	80.5	150.8	35.4	31.6	32.7	33.7	113.5
b	75.6	145.2	35.9	31.0	33.1	36.0	108.0
c	63.7	147.9	29.1	26.0	34.0	31.5	110.0
d	105.9	157.7	42.6	29.2	35.0	40.7	129.1
e	77.5	148.3	35.2	30.5	34.0	34.4	107.6

```
#use loc
tmp.loc[['a', 'b'], 'BMXLEG']
```

```
a    31.6
b    31.0
Name: BMXLEG, dtype: float64
```

```
#use iloc (use only the index values not the label)
tmp.iloc[[0, 1], 3]
```

```
a    31.6
b    31.0
Name: BMXLEG, dtype: float64
```

15.0.3 Common errors and how to read them

```
#Gives an an invalid key error, we need to use iloc or the regular  
↪ loc  
#tmp[:, 'BMXBMI']
```

15.0.4 Problem

The above gives: `TypeError: unhashable type: 'slice'`

The `[]` method uses hashes to identify the columns to keep, and each column has an associated hash. A 'slice' (a subset of rows and columns) does not have an associated hash, thus causing this `TypeError`.

```
tmp.loc[:, 'BMXBMI']
```

```
a    35.4  
b    35.9  
c    29.1  
d    42.6  
e    35.2
```

```
Name: BMXBMI, dtype: float64
```

```
tmp.loc[:, 'BMXBMI'].values
```

```
array([35.4, 35.9, 29.1, 42.6, 35.2])
```

```
#this will also give an error because iloc just takes indices  
#tmp.iloc[:, 'BMXBMI']
```

15.0.5 Problem

The above gives: `ValueError: Location based indexing can only have [integer, integer slice (START point is INCLUDED, END point is EXCLUDED), listlike of integers, boolean array] types`

'BMXBMI' is not an integer that is less than or equal number of columns -1, or a list of boolean values, so it is the wrong value type.

```
tmp.iloc[:, 2]
```

```
a    35.4
b    35.9
c    29.1
d    42.6
e    35.2
```

Name: BMXBMI, dtype: float64

```
#this gives a key error , because no column is named 2
#tmp.loc[:, 2]
```

15.0.6 Problem

The above code gives: `TypeError: cannot do label indexing on <class 'pandas.core.indexes.base.Index'> with these indexers [2] of <class 'int'>`

2 is not one of the labels (i.e. column names) in the dataframe

```
# Here is another example of using a boolean list for indexing
↪ columns
tmp.loc[:, [False, False, True] + [False]*4]
```

	BMXBMI
a	35.4
b	35.9
c	29.1
d	42.6

	BMXBMI
e	35.2

```
tmp.iloc[:, 2]
```

```
a    35.4
b    35.9
c    29.1
d    42.6
e    35.2
```

Name: BMXBMI, dtype: float64

15.0.7 Change values inside a df

```
# We can use the .loc and .iloc methods to change values within the
↪ dataframe
tmp.iloc[0:3,2] = [0]*3
tmp.iloc[:,2]
```

```
a    0.0
b    0.0
c    0.0
d    42.6
e    35.2
```

Name: BMXBMI, dtype: float64

```
tmp.loc['a':'c', 'BMXBMI'] = [1]*3
tmp.loc[:, 'BMXBMI']
```

```
a    1.0
b    1.0
c    1.0
d    42.6
e    35.2
```

Name: BMXBMI, dtype: float64

```
# We can use the [] method when changing all the values of a column
tmp['BMXBMI'] = range(0, 5)
tmp
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
a	80.5	150.8	0	31.6	32.7	33.7	113.5
b	75.6	145.2	1	31.0	33.1	36.0	108.0
c	63.7	147.9	2	26.0	34.0	31.5	110.0
d	105.9	157.7	3	29.2	35.0	40.7	129.1
e	77.5	148.3	4	30.5	34.0	34.4	107.6

```
# We will get a warning when using the [] method with conditions to
↳ set new values in our dataframe
tmp[tmp.BMXBMI > 2]['BMXBMI'] = [10]*2

# Setting new values to a copy of tmp, but not tmp itself
# You can see that the above code did not change our dataframe 'tmp'.
tmp
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
a	80.5	150.8	0	31.6	32.7	33.7	113.5
b	75.6	145.2	1	31.0	33.1	36.0	108.0
c	63.7	147.9	2	26.0	34.0	31.5	110.0
d	105.9	157.7	3	29.2	35.0	40.7	129.1
e	77.5	148.3	4	30.5	34.0	34.4	107.6

```
# The correct way to do the above is with .loc or .iloc
tmp.loc[tmp.BMXBMI > 2, 'BMXBMI'] = [10]*2
tmp
```

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
a	80.5	150.8	0	31.6	32.7	33.7	113.5
b	75.6	145.2	1	31.0	33.1	36.0	108.0
c	63.7	147.9	2	26.0	34.0	31.5	110.0

	BMXWT	BMXHT	BMXBMI	BMXLEG	BMXARML	BMXARMC	BMXWAIST
d	105.9	157.7	10	29.2	35.0	40.7	129.1
e	77.5	148.3	10	30.5	34.0	34.4	107.6

16 Plot Multivariate Distributions in Python

Sometimes we can get a lot of information about how two variables (or more) relate if we plot them together. This tutorial aims to show how plotting two variables together can give us information that plotting each one separately may miss.

```
# import the packages we are going to be using
import numpy as np # for getting our distribution
import matplotlib.pyplot as plt # for plotting
import seaborn as sns; sns.set() # For a different plotting theme
```

```
# Don't worry so much about what rho is doing here
# Just know if we have a rho of 1 then we will get a perfectly
# upward sloping line, and if we have a rho of -1, we will get
# a perfectly downward sloping line. A rho of 0 will
# get us a 'cloud' of points
r = 1

# Don't worry so much about the following three lines of code for now
# this is just getting the data for us to plot
mean = [15, 5]
cov = [[1, r], [r, 1]]

#create two vectors,x and y, both having a normal distribution
x, y = x, y = np.random.multivariate_normal(mean, cov, 400).T
```

```
# Adjust the figure size
plt.figure(figsize=(10,5))

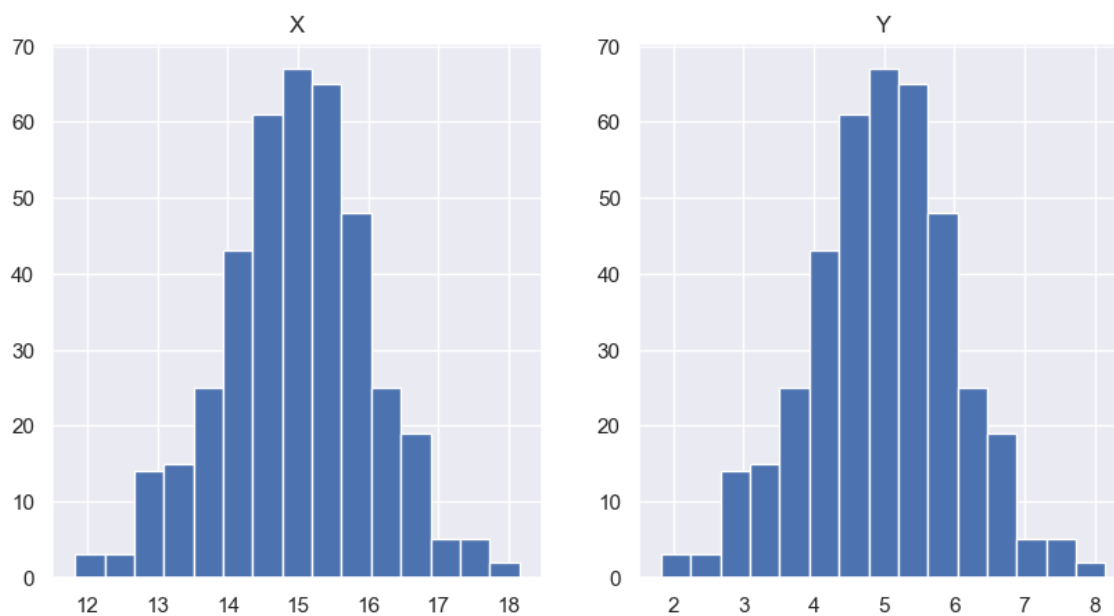
# Plot the histograms of X and Y next to each other
#create first plot
plt.subplot(1,2,1)
```



```
plt.hist(x = x, bins = 15)
plt.title("X")

#Plot second plot
plt.subplot(1,2,2)
plt.hist(x = y, bins = 15)
plt.title("Y")

plt.show()
```



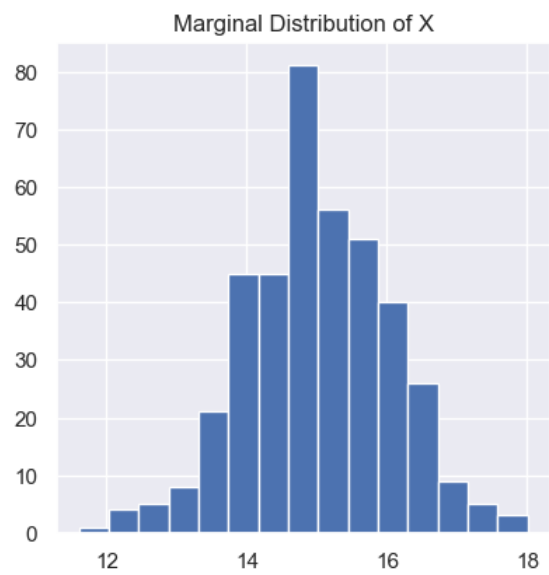
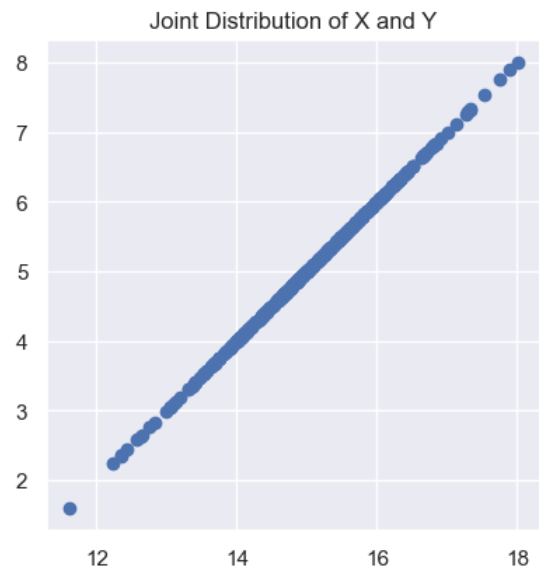
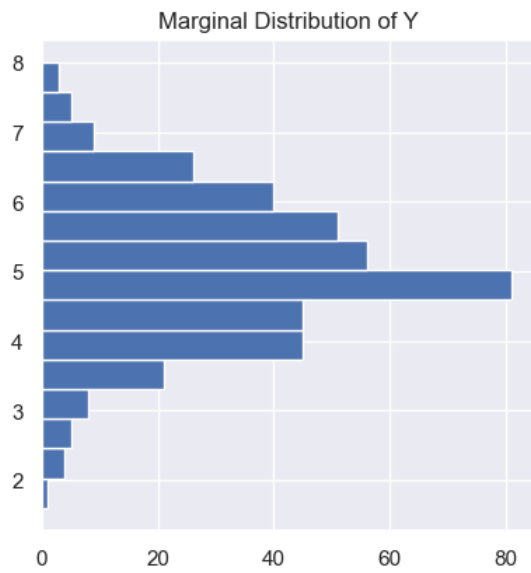
```
# Plot the data by including a scatterplot
plt.figure(figsize=(10,10))

#create the scatterplot
#we create a two by two plot and plot the scatter in the upper right
↪ corner
plt.subplot(2,2,2)
plt.scatter(x = x, y = y)
plt.title("Joint Distribution of X and Y")
```

```
# Plot the Marginal X Distribution at the bottom right (position 4)
plt.subplot(2,2,4)
plt.hist(x = x, bins = 15)
plt.title("Marginal Distribution of X")

# Plot the Marginal Y Distribution
plt.subplot(2,2,1)
plt.hist(x = y, orientation = "horizontal", bins = 15)
plt.title("Marginal Distribution of Y")

# Show the plots
plt.show()
```



17 Unit Testing

While we will not cover the [unit testing library](#) that python has, we wanted to introduce you to a simple way that you can test your code.

Unit testing is important because it the only way you can be sure that your code is do what you think it is doing.

Remember, just because ther are no errors does not mean your code is correct.

```
import numpy as np
import pandas as pd
import matplotlib as plt
pd.set_option('display.max_columns', 100) # Show all columns when
↳ looking at dataframe
```

```
# Download NHANES 2015-2016 data
df = pd.read_csv("../data/nhanes_2015_2016.csv")
df.index = range(1,df.shape[0]+1)
```

```
df.head()
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCHI
1	83732	1.0	NaN	1.0	1	1	62	3	1.0
2	83733	1.0	NaN	6.0	1	1	53	3	2.0
3	83734	1.0	NaN	NaN	1	1	78	3	1.0
4	83735	2.0	1.0	1.0	2	2	56	3	1.0
5	83736	2.0	1.0	1.0	2	2	42	4	1.0

17.0.1 Goal

We want to find the mean of first 100 rows of 'BPXSY1' when 'RIDAGEYR' > 60

```
# One possible way of doing this is:
# Current version of python will include this warning, older versions
↳ will not
#this gives an index error
#pd.Series.mean(df[df.RIDAGEYR > 60].loc[range(0,100), 'BPXSY1'])
```

```
df[df.RIDAGEYR > 60]['BPXSY1'].head(100).mean()
```

136.29166666666666

```
# test our code on only ten rows so we can easily check
test = pd.DataFrame({'col1': np.repeat([3,1],5), 'col2':
↳ range(3,13)}, index=range(1,11))
test
```

	col1	col2
1	3	3
2	3	4
3	3	5
4	3	6
5	3	7
6	1	8
7	1	9
8	1	10
9	1	11
10	1	12

```
# pd.Series.mean(df[df.RIDAGEYR > 60].loc[range(0,5), 'BPXSY1'])
# should return 5

#the code below would give the wrong number but returns an error
#pd.Series.mean(test[test.col1 > 2].loc[range(0,5), 'col2'])
test[test.col1 > 2]['col2'].head(5).mean()
```

5.0

What went wrong?

```
#test[test.col1 > 2].loc[range(0,5), 'col2']  
# 0 is not in the row index labels because the second row's value is  
↪ < 2. For now, pandas defaults to filling this  
# with NaN
```

```
# Using the .iloc method instead, we are correctly choosing the first  
↪ 5 rows, regardless of their row labels  
test[test.col1 >2].iloc[range(0,5), 1]
```

```
1    3  
2    4  
3    5  
4    6  
5    7  
Name: col2, dtype: int64
```

```
pd.Series.mean(test[test.col1 >2].iloc[range(0,5), 1])
```

5.0

```
# We can compare what our real dataframe looks like with the  
↪ incorrect and correct methods  
#Filled with NaN whenever a row label does not meet the condition  
#df[df.RIDAGEYR > 60].loc[range(0,5), :] #
```

```
#This Correctly picks the first five rows such that 'RIDAGEYR' > 60  
df[df.RIDAGEYR > 60].iloc[range(0,5), :]
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDC
1	83732	1.0	NaN	1.0	1	1	62	3	1.0
3	83734	1.0	NaN	NaN	1	1	78	3	1.0
6	83737	2.0	2.0	NaN	2	2	72	1	2.0

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCH
14	83754	2.0	1.0	1.0	2	2	67	2	1.0
15	83755	1.0	NaN	3.0	2	1	67	4	1.0

```
# Applying the correct method to the original question about BPXSY1
print(pd.Series.mean(df[df.RIDAGEYR > 60].iloc[range(0,100), 16]))
```

136.29166666666666

```
# Another way to reference the BPXSY1 variable
print(pd.Series.mean(df[df.RIDAGEYR > 60].iloc[range(0,100),
↪ df.columns.get_loc('BPXSY1')]))
```

136.29166666666666

18 Analysis of multivariate data - NHANES case study

In this notebook, we illustrate several basic techniques for exploring data using methods for understanding multivariate relationships. The statistical methods discussed here will parallel the methods discussed in the multivariate methods section of the course, and build on the univariate analysis discussed earlier. As with the univariate notebook, we use here the 2015-2016 wave of the [NHANES](#) study for illustration.

Many of the analyses presented in this notebook use the Matplotlib and Seaborn libraries for data visualization. These are very powerful tools that give you a vast number of options when constructing plots. We will not explain every option to every function in the examples below. You can use the [Matplotlib](#) and [Seaborn](#) documentation to fully understand the options, and you can experiment with these and other plots on your own to get a better sense of what can be done.

We start with the usual library import statements:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from scipy import stats
```

Next we load the NHANES data, just as we did for the univariate analyses.

```
da = pd.read_csv("../data/nhanes_2015_2016.csv")
da.head()
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCH
0	83732	1.0	NaN	1.0	1	1	62	3	1.0
1	83733	1.0	NaN	6.0	1	1	53	3	2.0

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	DMDCHI
2	83734	1.0	NaN	NaN	1	1	78	3	1.0
3	83735	2.0	1.0	1.0	2	2	56	3	1.0
4	83736	2.0	1.0	1.0	2	2	42	4	1.0

18.0.1 Quantitative bivariate data

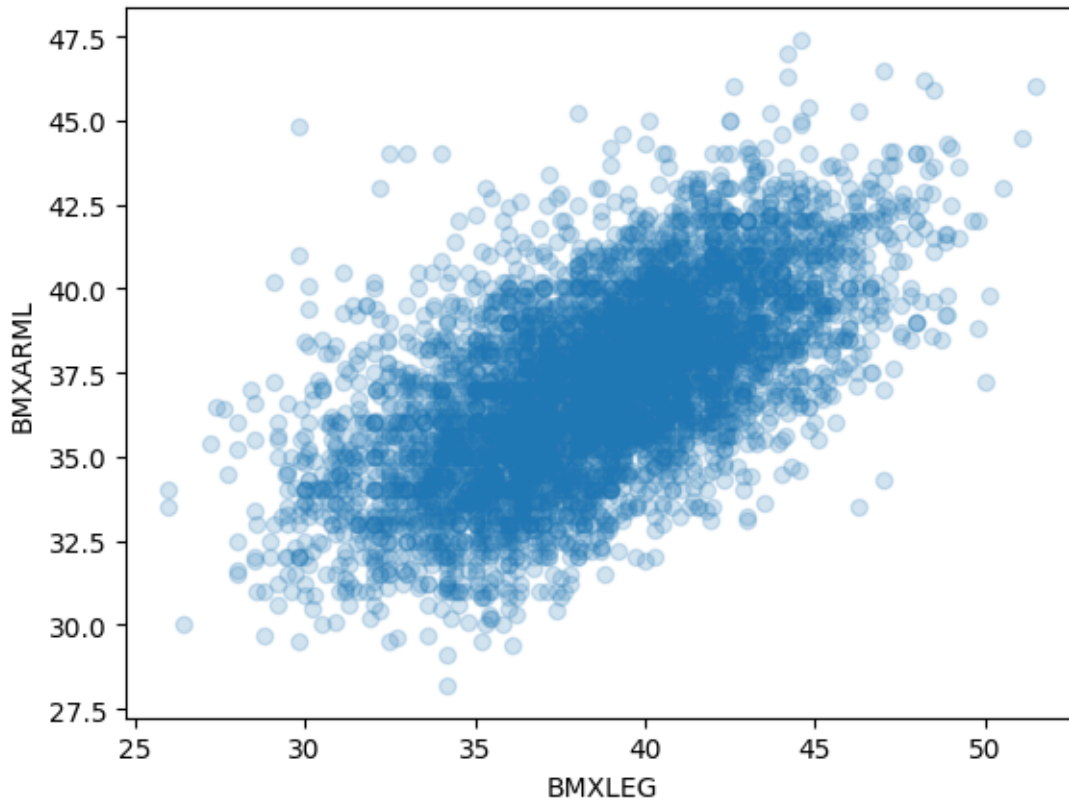
Bivariate data arise when every “unit of analysis” (e.g. a person in the NHANES dataset) is assessed with respect to two traits (the NHANES subjects were assessed for many more than two traits, but we can consider two traits at a time here).

A scatterplot is a very common and easily-understood visualization of quantitative bivariate data. Below we make a scatterplot of arm length against leg length. This means that arm length ([BMXARML](#)) is plotted on the vertical axis and leg length ([BMXLEG](#)) is plotted on the horizontal axis). We see a positive dependence between the two measures – people with longer arms tend to have longer legs, and vice-versa. However it is far from a perfect relationship.

In a scatterplot with more than around 100 points, “overplotting” becomes an issue. This means that many points fall on top of each other in the plot, which obscures relationships in the middle of the distribution and over-emphasizes the extremes. One way to mitigate overplotting is to use an “alpha” channel to make the points semi-transparent, as we have done below.

```
sns.regplot(x="BMXLEG", y="BMXARML", data=da, fit_reg=False,
↪ scatter_kws={"alpha": 0.2})
```

```
<AxesSubplot:xlabel='BMXLEG', ylabel='BMXARML'>
```



Another way to avoid overplotting is to make a plot of the “density” of points. In the plots below, darker colors indicate where a greater number of points fall. The two plot margins show the densities for the arm lengths and leg lengths separately, while the plot in the center shows their density jointly.

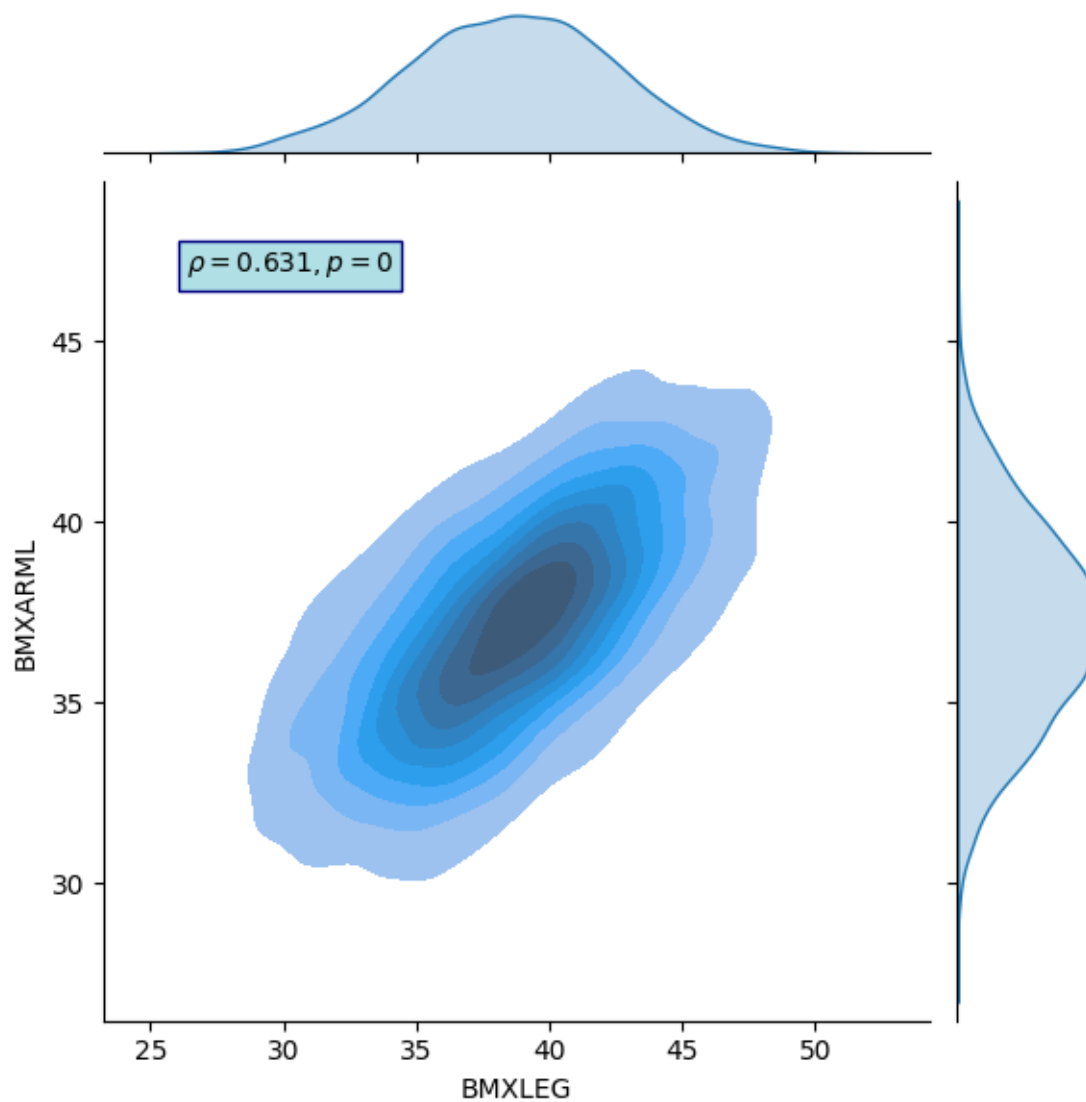
This plot also shows the Pearson correlation coefficient between the arm length and leg length, which is 0.62. As discussed in the course, the Pearson correlation coefficient ranges from -1 to 1, with values approaching 1 indicating a more perfect positive dependence. In many settings, a correlation of 0.62 would be considered a moderately strong positive dependence.

```
g = sns.jointplot(x="BMXLEG", y="BMXARML", kind='kde', fill=True,
    ↪ data=da)
r, p = stats.spearmanr(da['BMXLEG'], da['BMXARML'],
    ↪ nan_policy='omit')
g.ax_joint.annotate(f'$\\rho = {r:.3f}$', p = {p:3g}$',
```

```

xy=(0.1, 0.9), xycoords='axes fraction',
ha='left', va='center',
bbox={'boxstyle': 'square', 'fc': 'powderblue',
↪   'ec': 'navy'})
plt.show()

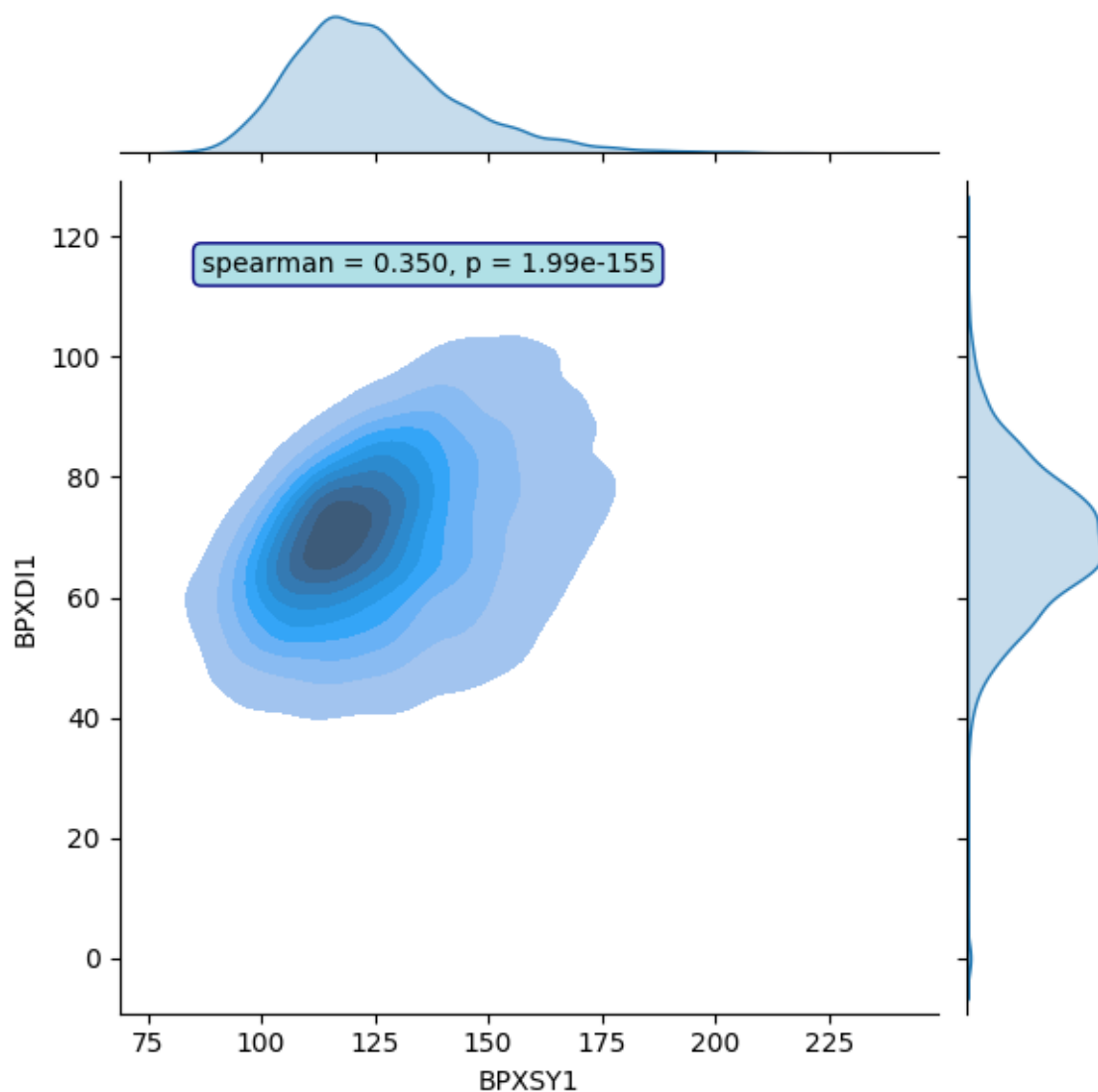
```



As another example with slightly different behavior, we see that systolic and diastolic blood

pressure (essentially the maximum and minimum blood pressure between two consecutive heart beats) are more weakly correlated than arm and leg length, with a correlation coefficient of 0.32. This weaker correlation indicates that some people have unusually high systolic blood pressure but have average diastolic blood pressure, and vice versa.

```
g = sns.jointplot(x="BPXSY1", y="BPXDI1", kind='kde', fill = True,
↳ data=da)
r, p = stats.spearmanr(da['BPXSY1'], da['BPXDI1'], nan_policy='omit')
g.ax_joint.annotate(f'spearman = {r:.3f}, p = {p:.3g}',
                    xy=(0.1, 0.9), xycoords='axes fraction',
                    ha='left', va='center',
                    bbox={'boxstyle': 'round', 'fc': 'powderblue',
↳ 'ec': 'navy'})
plt.show()
```



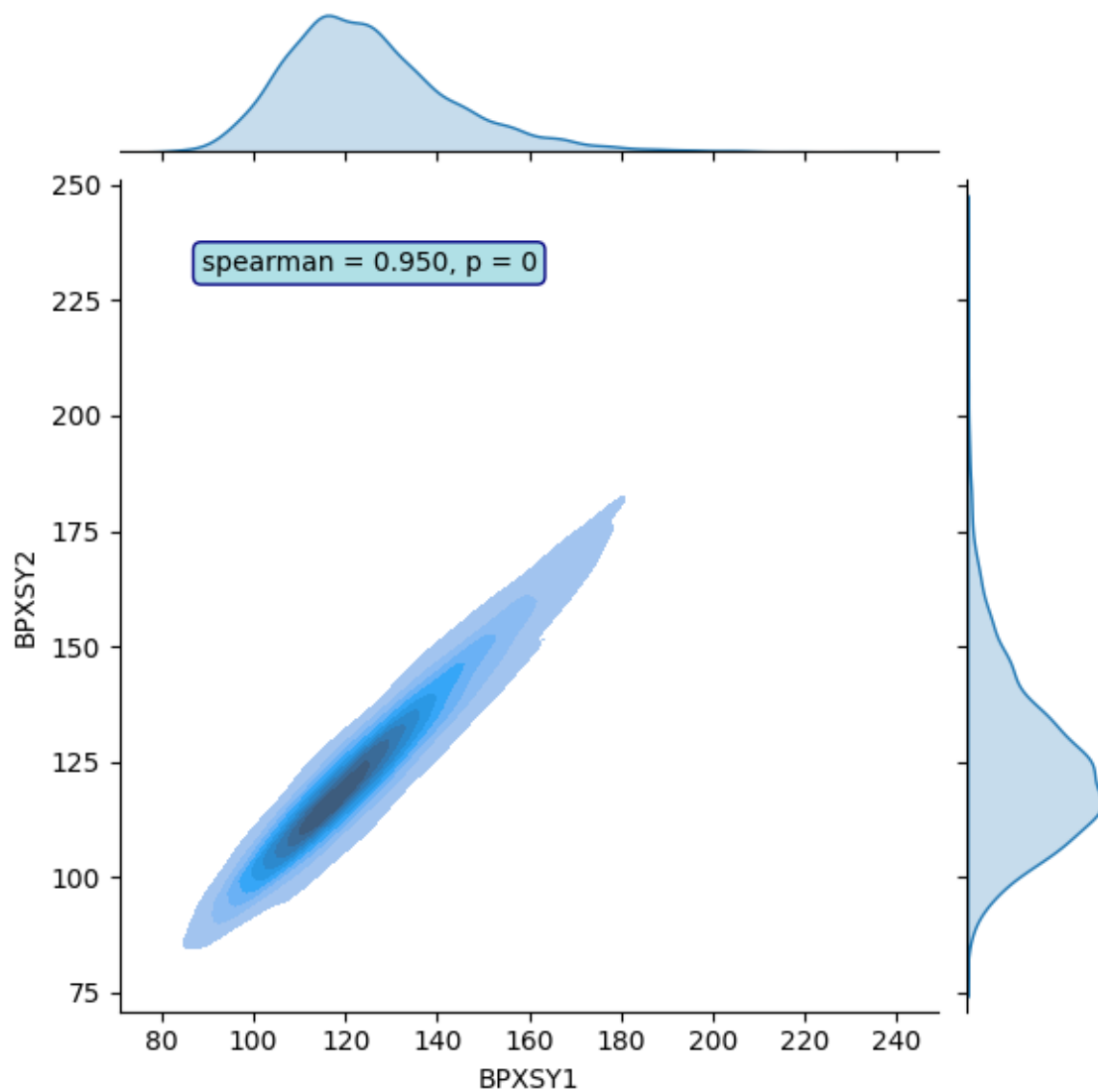
Next we look at two repeated measures of systolic blood pressure, taken a few minutes apart on the same person. These values are very highly correlated, with a correlation coefficient of around 0.96.

```
g = sns.jointplot(x="BPXSY1", y="BPXSY2", kind='kde', fill = True,
↪ data=da)
r, p = stats.spearmanr(da['BPXSY1'], da['BPXSY2'], nan_policy='omit')
```

```

g.ax_joint.annotate(f'spearman = {r:.3f}, p = {p:.3g}',
                    xy=(0.1, 0.9), xycoords='axes fraction',
                    ha='left', va='center',
                    bbox={'boxstyle': 'round', 'fc': 'powderblue',
↪      'ec': 'navy'})
plt.show()

```



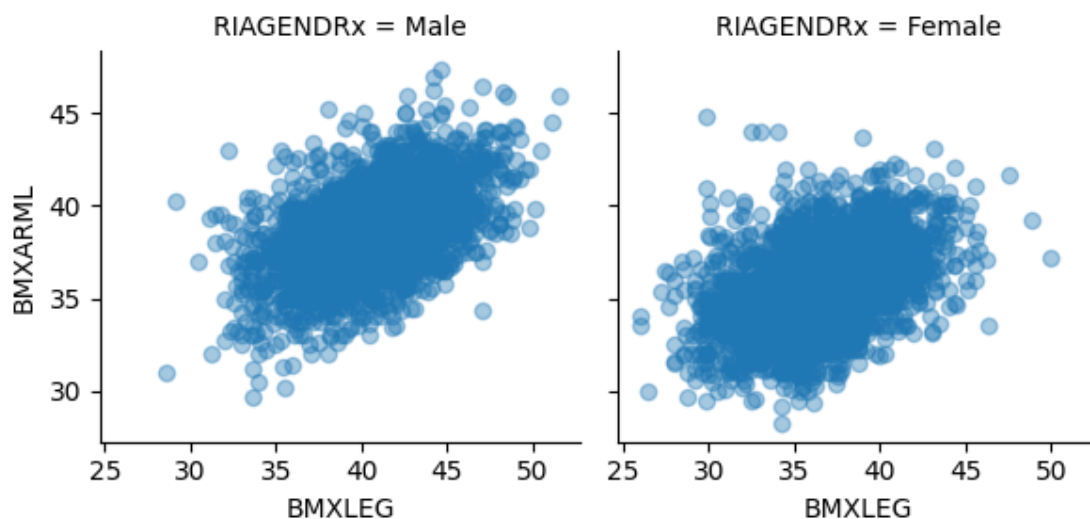
18.0.2 Heterogeneity and stratification

Most human characteristics are complex – they vary by gender, age, ethnicity, and other factors. This type of variation is often referred to as “heterogeneity”. When such heterogeneity is present, it is usually productive to explore the data more deeply by stratifying on relevant factors, as we did in the univariate analyses.

Below, we continue to probe the relationship between leg length and arm length, stratifying first by gender, then by gender and ethnicity. The gender-stratified plot indicates that men tend to have somewhat longer arms and legs than women – this is reflected in the fact that the cloud of points on the left is shifted slightly up and to the right relative to the cloud of points on the right. In addition, the correlation between arm length and leg length appears to be somewhat weaker in women than in men.

```
da["RIAGENDRx"] = da.RIAGENDR.replace({1: "Male", 2: "Female"})
sns.FacetGrid(da, col="RIAGENDRx").map(plt.scatter, "BMXLEG",
    ↪ "BMXARML", alpha=0.4).add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x7fd4f55a0710>



Consistent with the scatterplot, a slightly weaker correlation between arm length and leg length in women (compared to men) can be seen by calculating the correlation coefficient separately within each gender.

The `'corr'` method of a dataframe calculates the correlation coefficients for every pair of variables in the dataframe. This method returns a “correlation matrix”, which is a table containing the correlations between every pair of variables in the data set. Note that the diagonal of a correlation matrix always contains 1's, since a variable always has correlation 1 with itself. The correlation matrix is also symmetric around this diagonal, since the correlation between two variables ‘X’ and ‘Y’ does not depend on the order in which we consider the two variables.

In the results below, we see that the correlation between leg length and arm length in men is 0.50, while in women the correlation is 0.43.

```
print(da.loc[da.RIAGENDRx=="Female", ["BMXLEG",
↪ "BMXARML"]].dropna().corr())
```

	BMXLEG	BMXARML
BMXLEG	1.000000	0.434703
BMXARML	0.434703	1.000000

```
print(da.loc[da.RIAGENDRx=="Male", ["BMXLEG",
↪ "BMXARML"]].dropna().corr())
```

	BMXLEG	BMXARML
BMXLEG	1.000000	0.505426
BMXARML	0.505426	1.000000

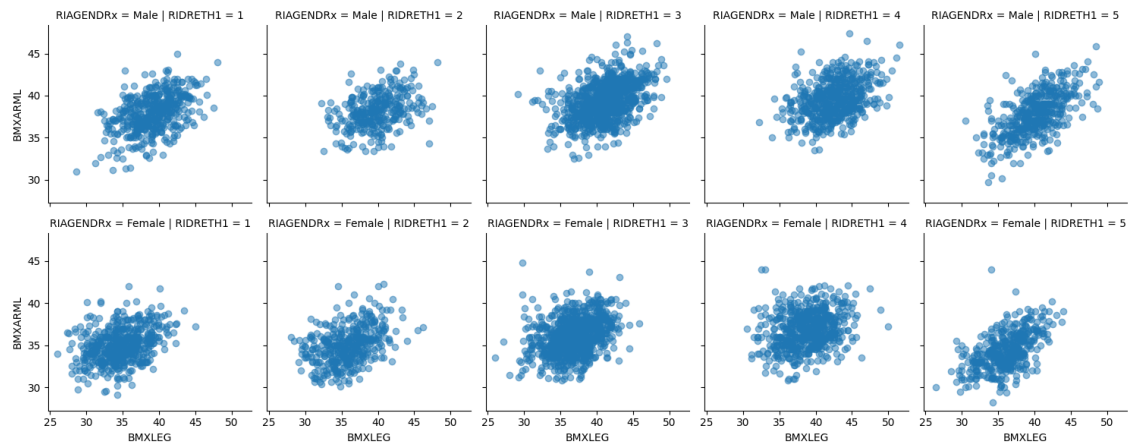
Next we look to stratifying the data by both gender and ethnicity. This results in $2 \times 5 = 10$ total strata, since there are 2 gender strata and 5 ethnicity strata. These scatterplots reveal differences in the means as well as differences in the degree of association (correlation) between different pairs of variables. We see that although some ethnic groups tend to have longer/shorter arms and legs than others, the relationship between arm length and leg length within genders is roughly similar across the ethnic groups.

One notable observation is that ethnic group 5, which consists of people who report being multi-racial or are of any race not treated as a separate group (due to small sample size), the correlation between arm length and leg length is stronger, especially for men. This is not surprising, as greater heterogeneity can allow correlations to emerge that are indiscernible in more homogeneous data.


```

_ = sns.FacetGrid(da, col="RIDRETH1",
↪ row="RIAGENDRx").map(plt.scatter, "BMXLEG", "BMXARML",
↪ alpha=0.5).add_legend()

```



18.0.3 Categorical bivariate data

In this section we discuss some methods for working with bivariate data that are categorical. We can start with a contingency table, which counts the number of people having each combination of two factors. To illustrate, we will consider the NHANES variables for marital status and education level.

First, we create new versions of these two variables using text labels instead of numbers to represent the categories. We also create a new data set that omits people who responded “Don’t know” or who refused to answer these questions.

```

#CLEAN up df
da["DMDEDUC2x"] = da.DMDEDUC2.replace({1: "<9", 2: "9-11", 3:
↪ "HS/GED", 4: "Some college/AA", 5: "College",
↪ 7: "Refused", 9: "Don't
↪ know"})
da["DMDMARTLx"] = da.DMDMARTL.replace({1: "Married", 2: "Widowed", 3:
↪ "Divorced", 4: "Separated", 5: "Never married",
↪ 6: "Living w/partner", 77:
↪ "Refused"})

```

```
db = da.loc[(da.DMDEDUC2x != "Don't know") & (da.DMDMARTLx !=  
↪ "Refused"), :]
```

18.0.4 Create count table with pd.crosstab

Now we can create a contingency table, counting the number of people in each cell defined by a combination of education and marital status.

```
x = pd.crosstab(db.DMDEDUC2x, da.DMDMARTLx)  
x
```

DMDMARTLx DMDEDUC2x	Divorced	Living w/partner	Married	Never married	Separated	Widowed
9-11	62	80	305	117	39	40
<9	52	66	341	65	43	88
College	120	85	827	253	22	59
HS/GED	127	133	550	237	40	99
Some college/AA	217	163	757	332	42	108

The results will be easier to interpret if we normalize the data. A contingency table can be normalized in three ways – we can make the rows sum to 1, the columns sum to 1, or the whole table sum to 1. Below we normalize within rows. This gives us the proportion of people in each educational attainment category who fall into each group of the marital status variable.

The modal (most common) marital status for people within each educational attainment group is “married”. However quantitatively, the proportion of people who are married varies substantially, and is notably higher for college graduates (around 61%) compared to groups with lower educational attainment.

```
x.apply(lambda z: z/z.sum() * 100, axis=1)
```

DMDMARTLx DMDEDUC2x	Divorced	Living w/partner	Married	Never married	Separated	Widowed
9-11	9.642302	12.441680	47.433904	18.195956	6.065319	6.220840
<9	7.938931	10.076336	52.061069	9.923664	6.564885	13.435115

DMDMARTLx DMDEDUC2x	Divorced	Living w/partner	Married	Never married	Separated	Widowed
College	8.784773	6.222548	60.541728	18.521230	1.610542	4.319180
HS/GED	10.708263	11.214165	46.374368	19.983137	3.372681	8.347386
Some college/AA	13.403335	10.067943	46.757258	20.506485	2.594194	6.670784

We can also normalize within the columns instead of normalizing within the rows. This gives us the proportion of people with each marital status group who have each level of educational attainment.

```
x.apply(lambda z: z/z.sum() * 100, axis=0)
```

DMDMARTLx DMDEDUC2x	Divorced	Living w/partner	Married	Never married	Separated	Widowed
9-11	10.726644	15.180266	10.971223	11.653386	20.967742	10.152284
<9	8.996540	12.523719	12.266187	6.474104	23.118280	22.335025
College	20.761246	16.129032	29.748201	25.199203	11.827957	14.974619
HS/GED	21.972318	25.237192	19.784173	23.605578	21.505376	25.126904
Some college/AA	37.543253	30.929791	27.230216	33.067729	22.580645	27.411168

We see here that the plurality of divorced people have some college but have not graduated from college, while the plurality of married people are college graduates.

It is quite plausible that there are gender differences in the relationship between educational attainment and marital status. Therefore we can look at the proportion of people in each marital status category, for each combination of the gender and education variables. This analyses yields some interesting trends, notably that women are much more likely to be widowed or divorced than men (e.g. women in the HS/GED group are around 3 times more likely to be widowed than men in the HS/GED group).

```
# The following line does these steps, reading the code from left to
↪ right:
# 1 Group the data by every combination of gender, education, and
↪ marital status
# 2 Count the number of people in each cell using the 'size' method
# 3 Pivot the marital status results into the columns (using unstack)
# 4 Fill any empty cells with 0
```

```
# 5 Normalize the data by row
db.groupby(["RIAGENDRx", "DMDDEDUC2x",
↳ "DMDMARTLx"]).size().unstack().fillna(0).apply(lambda x:
↳ x/x.sum() * 100, axis=1)
```

	DMDMARTLx	Divorced	Living w/partner	Married	Never married	Separated
RIAGENDRx	DMDDEDUC2x					
Female	9-11	11.340206	12.371134	41.237113	17.182131	7.560137
	<9	9.169054	9.169054	42.406877	10.888252	8.882521
	College	11.018131	5.578801	57.740586	18.270572	1.673640
	HS/GED	12.178388	10.977702	41.337907	18.867925	4.116638
	Some college/AA	14.867841	9.911894	41.850220	21.035242	3.193833
Male	9-11	8.238636	12.500000	52.556818	19.034091	4.829545
	<9	6.535948	11.111111	63.071895	8.823529	3.921569
	College	6.317411	6.933744	63.636364	18.798151	1.540832
	HS/GED	9.286899	11.442786	51.243781	21.061360	2.653400
	Some college/AA	11.533052	10.267229	53.023910	19.831224	1.828411

One factor behind the greater number of women who are divorced and widowed could be that women live longer than men. To minimize the impact of this factor, we can recalculate the above table using a few narrow bands of ages. To simplify here, we collapse the marital status data to characterize people as being either “married” or “unmarried” This allows us to focus on the marriage rate, which is a widely-studied variable in social science research.

There are a number of intriguing results here. For example, the marriage rate seems to drop as college-educated people get older (e.g. 71% of college educated women between 49 and 50 are married, but only 65% of college educated women between 50 and 59 are married, an even larger drop occurs for men). However in people with a HS/GED level of education, the marriage rate is higher for older people (although it is lower compared to the college educated sample). There are a number of possible explanations for this, for example, that remarriage after divorce is less common among college graduates.

```
#look at narrower age group
dx = db.loc[(db.RIDAGEYR >= 40) & (db.RIDAGEYR < 50)]
a = dx.groupby(["RIAGENDRx", "DMDDEDUC2x",
↳ "DMDMARTLx"]).size().unstack().fillna(0).apply(lambda x:
↳ x/x.sum(), axis=1)
a
```

RIAGENDRx	DMDMARTLx DMDEDUC2x	Divorced	Living w/partner	Married	Never married	Separated	W
Female	9-11	0.090909	0.072727	0.581818	0.163636	0.090909	0
	<9	0.089286	0.142857	0.464286	0.142857	0.125000	0
	College	0.121019	0.019108	0.713376	0.108280	0.012739	0
	HS/GED	0.151163	0.127907	0.476744	0.116279	0.104651	0
	Some college/AA	0.184713	0.082803	0.509554	0.152866	0.057325	0
Male	9-11	0.092593	0.129630	0.574074	0.166667	0.037037	0
	<9	0.023810	0.142857	0.714286	0.095238	0.000000	0
	College	0.051724	0.034483	0.879310	0.034483	0.000000	0
	HS/GED	0.093023	0.081395	0.616279	0.162791	0.046512	0
	Some college/AA	0.125000	0.125000	0.625000	0.096154	0.019231	0

```
#look at another age group
dx = db.loc[(db.RIDAGEYR >= 50) & (db.RIDAGEYR < 60)]
b = dx.groupby(["RIAGENDRx", "DMDEDUC2x",
↳ "DMDMARTLx"]).size().unstack().fillna(0).apply(lambda x:
↳ x/x.sum(), axis=1)
```

```
#view the two dfs for the married group
print(a.loc[:, ["Married"]].unstack())
```

DMDMARTLx	Married				
DMDEDUC2x	9-11	<9	College	HS/GED	Some college/AA
RIAGENDRx					
Female	0.581818	0.464286	0.713376	0.476744	0.509554
Male	0.574074	0.714286	0.879310	0.616279	0.625000

```
print(b.loc[:, ["Married"]].unstack())
```

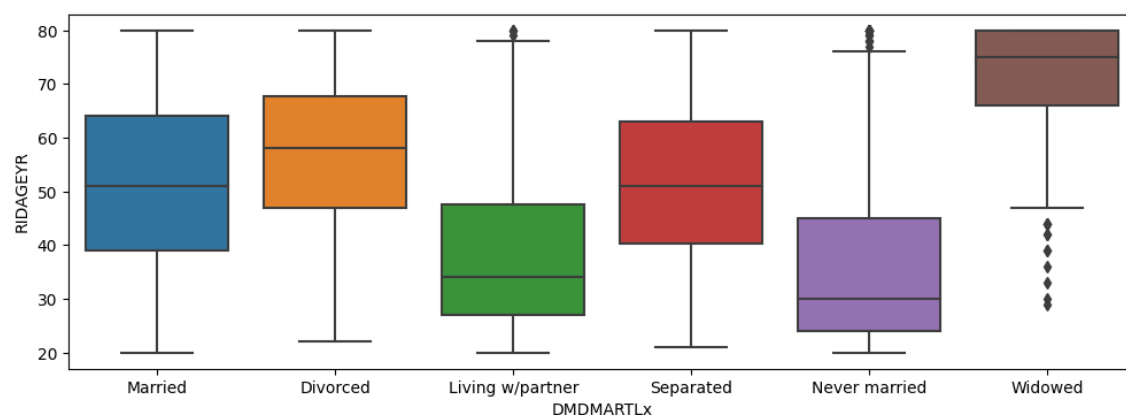
DMDMARTLx	Married				
DMDEDUC2x	9-11	<9	College	HS/GED	Some college/AA
RIAGENDRx					
Female	0.490566	0.511111	0.648649	0.563107	0.496403
Male	0.666667	0.622642	0.737374	0.637255	0.555556

We conclude this section by noting that marital status is associated with many factors, including gender and educational status, but also varies strongly by age and birth cohort. For example, it is unlikely for young people to be widowed, and it is less likely for older people to be “never married”, since a person can transition from “never married” into one of the other categories, but can never move back. Below we will consider the role of age in more detail, and later in the course we will revisit these questions using more sophisticated analytic methods that can account for all of these factors simultaneously. However, since NHANES is a cross-sectional study, there are certain important questions that it cannot be used to answer. For example, while we know each person’s current marital status, we do not know their full marital history (e.g. how many times and at what ages they were married or divorced).

18.0.5 Mixed categorical and quantitative data

Another situation that commonly arises in data analysis is when we wish to analyze bivariate data consisting of one quantitative and one categorical variable. To illustrate methods that can be used in this setting, we consider the relationship between marital status and age in the NHANES data. Specifically, we consider the distribution of ages for people who are currently in each marital status category. A natural tool in this setting is side-by-side boxplots. Here we see some unsurprising things – widowed people tend to be older, and never-married people tend to be younger.

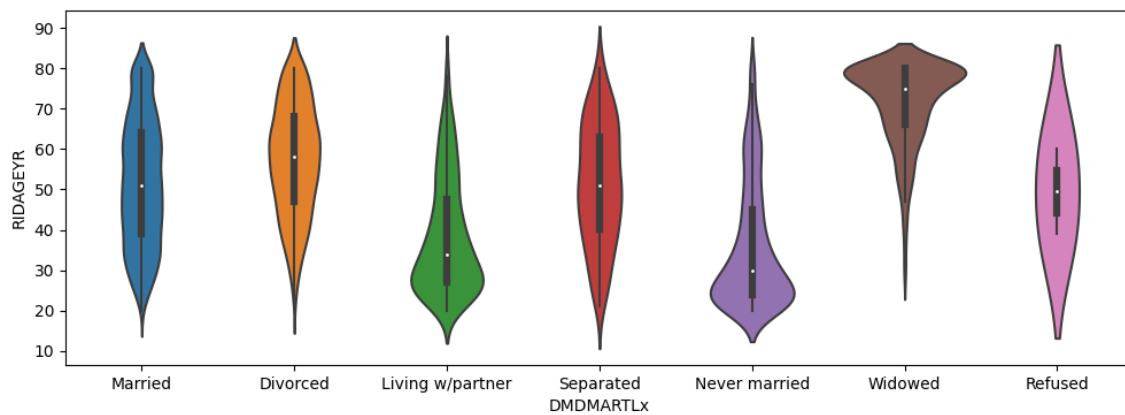
```
plt.figure(figsize=(12, 4))
a = sns.boxplot(x = db.DMDMARTLx, y = db.RIDAGEYR)
```



When we have enough data, a “violinplot” gives a bit more insight into the shapes of the

distributions compared to a traditional boxplot. The violinplot below is based on the same data as the boxplot above. We can see quite clearly that the distributions with low mean (living with partner, never married) are strongly right-skewed, while the distribution with high mean (widowed) is strongly left-skewed. The other distributions have intermediate mean values, and are approximately symmetrically distributed. Note also that the never-married distribution has a long shoulder, suggesting that this distributions includes many people who are never-married because they are young, and have not yet reached the ages when people typically marry, but also a substantial number of people will marry for the first time anywhere from their late 30's to their mid-60's.

```
plt.figure(figsize=(12, 4))
a = sns.violinplot(x = da.DMDMARTLx, y = da.RIDAGEYR)
```



19 Practice notebook for multivariate analysis using NHANES data

This notebook will give you the opportunity to perform some multivariate analyses on your own using the NHANES study data. These analyses are similar to what was done in the week 3 NHANES case study notebook.

You can enter your code into the cells that say “enter your code here”, and you can type responses to the questions into the cells that say “Type Markdown and LaTeX”.

Note that most of the code that you will need to write below is very similar to code that appears in the case study notebook. You will need to edit code from that notebook in small ways to adapt it to the prompts below.

To get started, we will use the same module imports and read the data in the same way as we did in the case study:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import statsmodels.api as sm
import numpy as np

da = pd.read_csv("../data/nhanes_2015_2016.csv")

#have a more descriptive name for some columns
da["RIAGENDRx"] = da.RIAGENDR.replace({1: "Male", 2: "Female"})
da["DMDEDUC2x"] = da.DMDEDUC2.replace({1: "<9", 2: "9-11", 3:
↪ "HS/GED", 4: "Some college/AA", 5: "College",
↪ "Refused", 9: "Don't know"})
da["DMDMARTLx"] = da.DMDMARTL.replace({1: "Married", 2: "Widowed", 3:
↪ "Divorced", 4: "Separated", 5: "Never married",
```



```
da.columns
```

```
6: "Living w/partner", 77:  
  ↪ "Refused"})
```

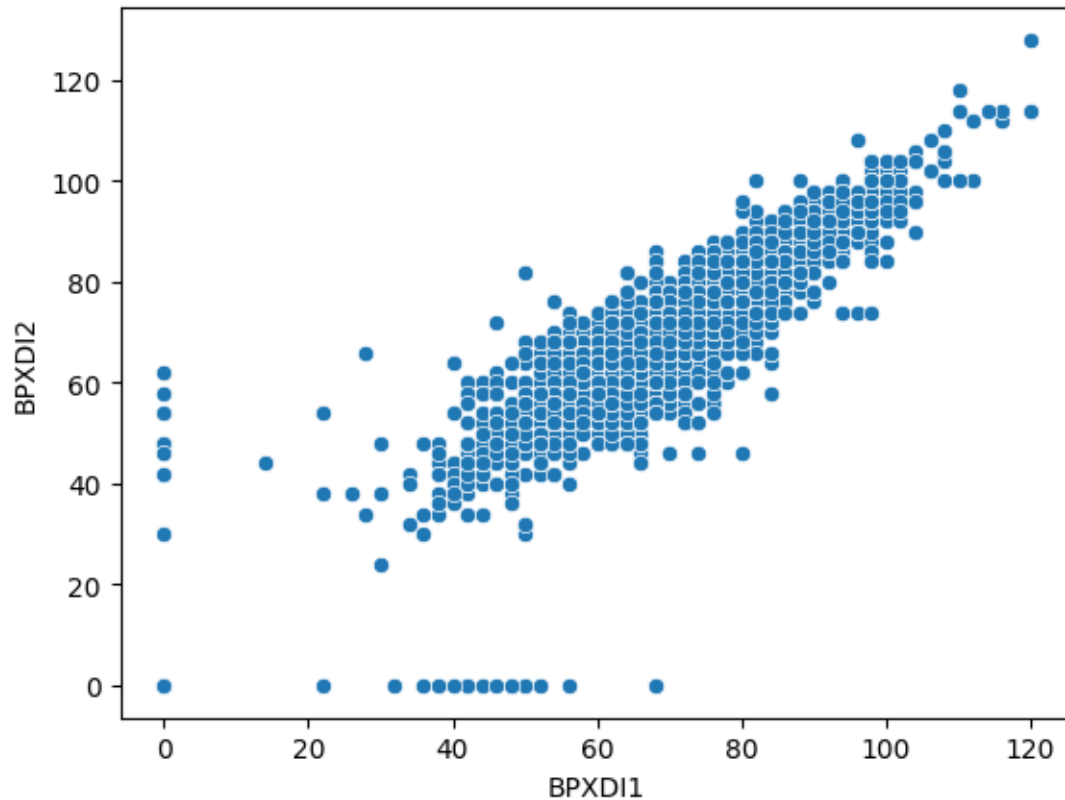
```
Index(['SEQN', 'ALQ101', 'ALQ110', 'ALQ130', 'SMQ020', 'RIAGENDR', 'RIDAGEYR',  
      'RIDRETH1', 'DMDCITZN', 'DMDEDUC2', 'DMDMARTL', 'DMDHHSIZ', 'WTINT2YR',  
      'SDMVPSU', 'SDMVSTRA', 'INDFMPIR', 'BPXSY1', 'BPXDI1', 'BPXSY2',  
      'BPXDI2', 'BMXWT', 'BMXHT', 'BMXBMI', 'BMXLEG', 'BMXARML', 'BMXARMC',  
      'BMXWAIST', 'HIQ210', 'RIAGENDRx', 'DMDEDUC2x', 'DMDMARTLx'],  
      dtype='object')
```

19.1 Question 1

Make a scatterplot showing the relationship between the first and second measurements of diastolic blood pressure ([BPXDI1](#) and [BPXDI2](#)). Also obtain the 4x4 matrix of correlation coefficients among the first two systolic and the first two diastolic blood pressure measures.

```
# show scatterplot  
sns.scatterplot(x='BPXDI1', y='BPXDI2', data = da)
```

```
<AxesSubplot:xlabel='BPXDI1', ylabel='BPXDI2'>
```



```
#generate correlation matrix
print(da.loc[:,["BPXDI1", "BPXDI2","BPXSY1",
↪ "BPXSY2"]].dropna().corr())
```

	BPXDI1	BPXDI2	BPXSY1	BPXSY2
BPXDI1	1.000000	0.884722	0.317497	0.329843
BPXDI2	0.884722	1.000000	0.277681	0.298392
BPXSY1	0.317497	0.277681	1.000000	0.962287
BPXSY2	0.329843	0.298392	0.962287	1.000000

Q1a. How does the correlation between repeated measurements of diastolic blood pressure relate to the correlation between repeated measurements of systolic blood pressure?

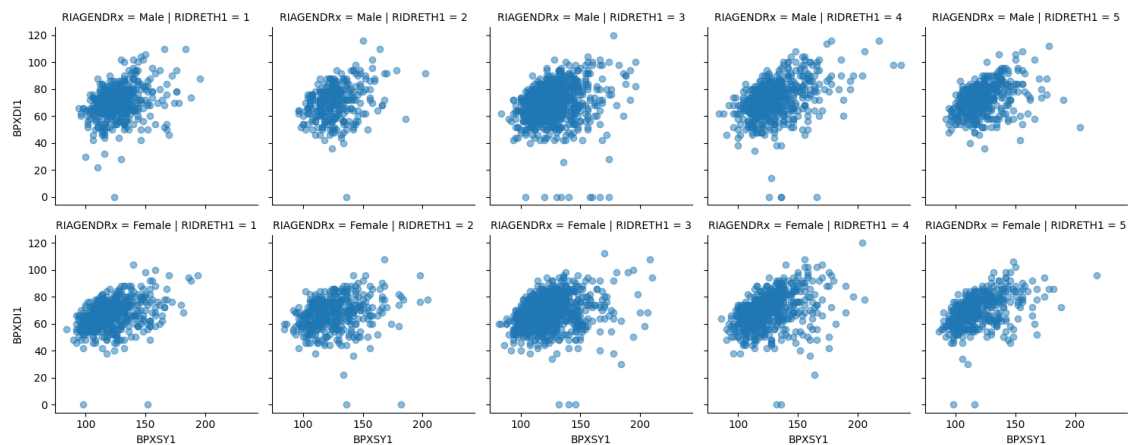
There is a better correlation between repeated systolic compared to diastolic measurements.

Q2a. Are the second systolic and second diastolic blood pressure measure more correlated or less correlated than the first systolic and first diastolic blood pressure measure?

19.2 Question 2

Construct a grid of scatterplots between the first systolic and the first diastolic blood pressure measurement. Stratify the plots by gender (rows) and by race/ethnicity groups (columns).

```
_ = sns.FacetGrid(da, col="RIDRETH1",  
  ↪ row="RIAGENDRx").map(plt.scatter, "BPXSY1", "BPXDI1",  
  ↪ alpha=0.5).add_legend()
```



Q3a. Comment on the extent to which these two blood pressure variables are correlated to different degrees in different demographic subgroups.

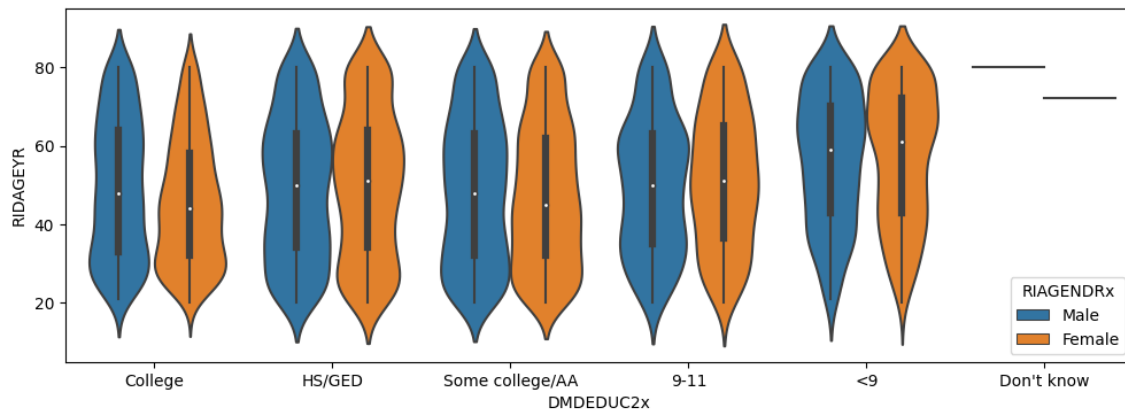
For all groups we see a positive correlation, but the slope esp. in ETH group 1 and 2 seems steeper for males compared to females.

19.3 Question 3

Use “violin plots” to compare the distributions of ages within groups defined by gender and educational attainment.

```
plt.figure(figsize = (12,4))
sns.violinplot(data=da, x="DMDEDUC2x", y="RIDAGEYR", hue="RIAGENDRx")
```

```
<AxesSubplot:xlabel='DMDEDUC2x', ylabel='RIDAGEYR'>
```



Q4a. Comment on any evident differences among the age distributions in the different demographic groups.

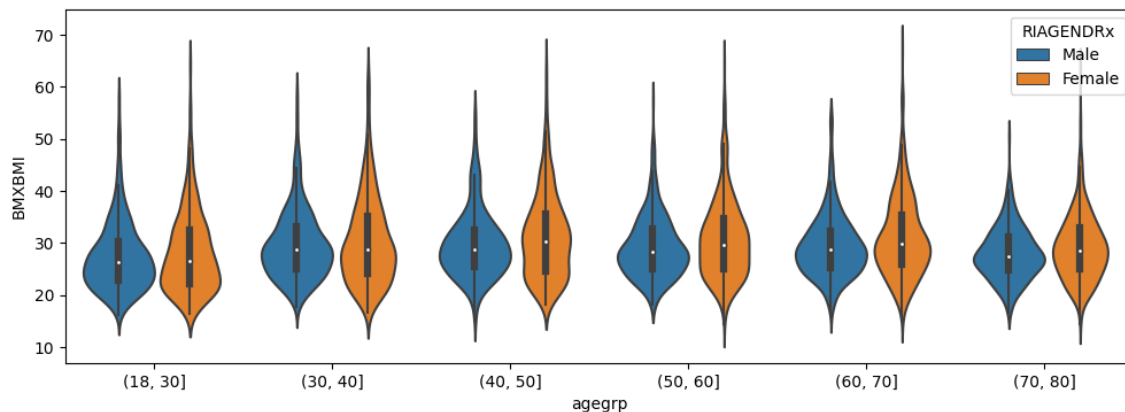
19.4 Question 4

Use violin plots to compare the distributions of BMI within a series of 10-year age bands. Also stratify these plots by gender.

```
#create age bands
da["agegrp"] = pd.cut(da.RIDAGEYR, [18, 30, 40, 50, 60, 70, 80])

#plot
plt.figure(figsize = (12,4))
sns.violinplot(data=da, y="BMXBMI", x="agegrp", hue="RIAGENDRx")
```

```
<AxesSubplot:xlabel='agegrp', ylabel='BMXBMI'>
```



Q5a. Comment on the trends in BMI across the demographic groups.

19.5 Question 5

Construct a frequency table for the joint distribution of ethnicity groups ([RIDRETH1](#)) and health-insurance status ([HIQ210](#)). Normalize the results so that the values within each ethnic group are proportions that sum to 1.

```
da.groupby(["RIDRETH1",
↳ "HIQ210"]).size().unstack().fillna(0).apply(lambda x: x/x.sum() *
↳ 100, axis=1)
```

HIQ210	1.0	2.0	9.0
RIDRETH1			
1	13.798220	85.756677	0.445104
2	12.869565	87.130435	0.000000
3	6.735437	93.143204	0.121359
4	10.865385	89.038462	0.096154
5	8.176101	91.572327	0.251572

Q6a. Which ethnic group has the highest rate of being uninsured in the past year?

Group 1 has the highest rate of not being ensured (category 1)