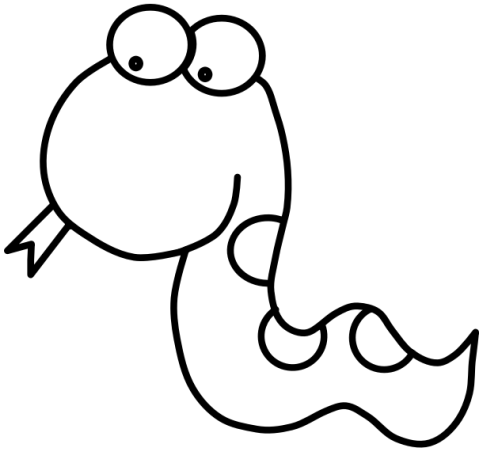


Pre-Work Overview

Angelica (Jelly) Spratley, MSc Analytics

Instructor, DS LIVE

Diane Tunnicliffe
Flatiron Data Science Grad
Coach, DS LIVE

Agenda

- Day 1: Python Essentials & Loops/Functions
- Day 2: Statistical Measures
- Day 3: Data Visualization

- Independent DataCamp Assignment (Due 1/21/22): Link
- Friday ThinkStats Book Club: Link

Day 1: Python Essentials Loops & Functions

Variable Types

- Numbers (Integers, Floats, etc.)
- Strings (e.g. “Hello World)
- Lists
- Tuples
- Dictionaries
- Booleans

} Collections

Variable Types

- **Numbers (Integers, Floats, etc.)**
- Strings (e.g. “Hello World)
- Lists
- Tuples
- Dictionaries
- Booleans

} Collections

Numbers

- Integers **(e.g. -10, 20, 100)**
- Floats - decimals **(-5.5, 10.8, 120.22)**
- Complex - imaginary numbers **(e.g. x + yj)**

Order of Operations for **NUMBERS**

- **P**arenthesis
- **E**xponentiation
- **M**ultiplication
- **D**ivision
- **A**ddition
- **S**ubtraction

```
In [4]: x = 3
        y = 2
        z = 7

        example = (x + z) ** y
        example

Out[4]: 100
```

```
In [5]: example2 = (3 + 7)**2
        example2

Out[5]: 100
```

ARITHMETIC OPERATIONS

- Modulus aka **Remainder (%)**

```
In [8]: x = 10
        y = 1
        z = 3

        example = (x % z) ** y
        example

Out[8]: 1
```

- Floor Division **(//)**

```
In [9]: x = 10
        y = 1
        z = 3

        example = (x // z) ** y
        example

Out[9]: 3
```

Variable Types

- Numbers (Integers, Floats, etc.)
- **Strings (e.g. “Hello World)**
- Lists
- Tuples
- Dictionaries
- Booleans
- Print formatting

} Collections

```
>>> print("Hello World!")
Hello World!
>>>
```

Strings

- Way to represent text characters
- Strings are **ARRAYS (**more on this later)
  - Index them
  - Loop through them
- Usually surrounded by **single or double quotes**

```
In [12]: x = 'Python for Beginners'

         print(x)

Python for Beginners
```

Strings - Type Casting

- You can **change** a number to a string or a string to a number by using:
  - **str()**
  - **int()**

- This is called type casting

```
In [14]: x = '1234'
         x = int(x)
         x

Out[14]: 1234
```

```
In [15]: y = 1234
         y = str(1234)
         y

Out[15]: '1234'
```

The Beauty of **Double Quotes**

- Allows you to type text that has built in apostrophes (e.g. I'll, You're)

```
In [13]: y = 'You're Ready'
         print(y)

           File "<ipython-input-13-5c5edb70947a>", line 1
             y = 'You're Ready'
                       ^
         SyntaxError: invalid syntax
```

VS

```
In [14]: y = "You're Ready"
         print(y)

         You're Ready
```

CONCATENATION (Placing strings side-by-side)

- You **CANNOT** concatenate numbers and strings together **(+)**

- Force **(aka type cast)** the number to a string if you want to concatenate

```
In [15]: Name = 'James'
         Dept_Num = 100
         Dept_Name = 'Finance'

         Work_Location = Name + Dept_Num + Dept_Name
         print(Work_Location)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-8c2824e2c62c> in <module>
      3 Dept_Name = 'Finance'
      4
----> 5 Work_Location = Name + Dept_Num + Dept_Name
      6 print(Work_Location)

TypeError: can only concatenate str (not "int") to str
```

```
In [16]: Name = 'James'
         Dept_Num = '100'
         Dept_Name = 'Finance'

         Work_Location = Name + Dept_Num + Dept_Name
         print(Work_Location)
```

```
James100Finance
```

CONCATENATION (Adding spaces aka empty strings)

- You can add empty strings **“ “** to concatenate spaces to strings

Indexing Strings

- First character position is **zero (0)**
- Can get the length of a string by using the **len()** built-in Python function
- Adding a colon **(:)** at the **end** of an index goes to the END of the string

```
In [24]: Email = 'samplemail@gmail.com'
         Email[11]

Out[24]: 'g'
```

```
In [25]: Email = 'samplemail@gmail.com'
         Email[11:]

Out[25]: 'gmail.com'
```

```
In [26]: Email = 'samplemail@gmail.com'
         len(Email)

Out[26]: 20
```

Common String Methods

- **.upper()** - change to uppercase
- **.lower()** - change to lowercase
- **.replace()** - replace letters/words
- **.split()** - split string into substrings
- More methods **HERE**.
- Type **help(str)** in Jupyter to see all the methods :)

Common String Methods

In [28]:
```
First_Name = 'James'
Last_Name = 'Bond'
Full_Name = 'James Bond'

print(First_Name.upper())
print(Last_Name.lower())
print(Full_Name.split(" "))
```

```
JAMES
bond
['James', 'Bond']
```

Formatted String Literals

- If you want to **add a variable assignment** to your printed text output, you use a formatted string literal

```
In [30]: Name = "Su Roberts"

         print(f"Welcome {Name}, Glad to Have You")

         Welcome Su Roberts, Glad to Have You
```

User Input aka Input Function

- You can prompt the user to respond using the input function

```
In [31]: Username = input("What is your username?")
         print(f"Welcome {Username}, thanks for visiting our site")

         What is your username?Jelly
         Welcome Jelly, thanks for visiting our site
```

Youtube Video

- Python Numbers and Strings

Variable Types

- Numbers (Integers, Floats, etc.)
- Strings (e.g. “Hello World)
- **Lists**
- Tuples
- Dictionaries
- Booleans

} Collections

Syntax of Lists

- Denoted by square brackets **[ ]**
- Items in the list are separated by commas
- Items can be any variable type
  - Numbers (Integers & Floats)
  - Strings
  - Booleans
  - Even Lists...

Indexing Lists

- Lists can be **indexed** like strings

```
In [9]: #Lists

        List = ["James Bond", 55, '123 Main Street']
        List[2]
```

```
Out[9]: '123 Main Street'
```

```
In [10]: List2 = [["James Bond", 55], ["Jill Bond", 52]]
         List2[1][1]
```

```
Out[10]: 52
```

Common Methods Lists

- **.insert**(position, item to insert)
- **.append**(x)
- **.remove**(item name)
- **.pop**(index number of item)

```
In [12]: List1 = ["Alan", "Sue"]
         List2 = ["001", "002"]

         print(List1 + List2)
```

```
['Alan', 'Sue', '001', '002']
```

```
In [21]: List1 = ["Alan", "Sue"]
         List1.append("Bob")
         List1
```

```
Out[21]: ['Alan', 'Sue', 'Bob']
```

List Comprehensions

- Generate a new list from existing iterables
- Syntax **[function for x in iterable <if condition>]**
  - **Still** surrounded by **brackets**
  - **X** can be **ANY** letter or word (p, q, number, etc.)
  - **Condition** is **OPTIONAL**
  - The **iterable** is a **collection (aka a created list)**
  - **Create these piece by piece**
  - Practice, practice, practice

```
In [22]: #List Comprehensions

         odds = [1, 3, 5, 7, 9]

         even_nums = [nums+1 for nums in odds]
         even_nums

Out[22]: [2, 4, 6, 8, 10]
```

List Comprehension Example 2

In [24]:
```
Salutation = ["Dr", "Mrs", "Ms"]

upper_salutation = [letters.upper() for letters in Salutation]
upper_salutation
```

Out[24]:
```
['DR', 'MRS', 'MS']
```

List Comprehension Example 3

In [26]:
```
minutes = [480, 540, 720, 120]

mins_to_hours = [minute/60 for minute in minutes]
mins_to_hours
```

Out[26]:
```
[8.0, 9.0, 12.0, 2.0]
```

List Comprehension w/ Condition

In [29]:
```
integers = [-10, 90, 15, -8, 86, -2]

change_negs = [abs(num) for num in integers if num < 0]
change_negs
```

Out[29]:
```
[10, 8, 2]
```

Yes...FOR LOOPS do the same thing (Example 1)

- List comprehensions are more **efficient**
- However, do the method you are comfortable with

```
In [1]: #List Comprehensions

        odds = [1, 3, 5, 7, 9]

        even_nums = [nums+1 for nums in odds]
        even_nums
```

```
Out[1]: [2, 4, 6, 8, 10]
```

VS

```
In [2]: even_list_from_odds = []
        for num in odds:
            even_list_from_odds.append(num+1)
        even_list_from_odds
```

```
Out[2]: [2, 4, 6, 8, 10]
```

FOR LOOP Steps

1. Initialize an **EMPTY** list
   a. Be sure it is a different name than the iterable
2. Append the function to that list
3. Return the new **FILLED** list

```
In [1]: #List Comprehensions

        odds = [1, 3, 5, 7, 9]

        even_nums = [nums+1 for nums in odds]
        even_nums

Out[1]: [2, 4, 6, 8, 10]
```

```
In [2]: even_list_from_odds = []  1
        for num in odds:
            even_list_from_odds.append(num+1)  2
        even_list_from_odds  3

Out[2]: [2, 4, 6, 8, 10]
```

Yes...FOR LOOPS do the same thing (Example 2)

In [7]:
```
Salutation = ["Dr", "Mrs", "Ms"]

upper_salutation = [letters.upper() for letters in Salutation]
upper_salutation
```

Out[7]:
```
['DR', 'MRS', 'MS']
```

In [8]:
```
upper_prefixes = []
for letters in Salutation:
    upper_prefixes.append(letters.upper())
upper_prefixes
```

Out[8]:
```
['DR', 'MRS', 'MS']
```

Yes...FOR LOOPS do the same thing (Example 3)

```
In [9]: minutes = [480, 540, 720, 120]

        mins_to_hours = [minute/60 for minute in minutes]
        mins_to_hours

Out[9]: [8.0, 9.0, 12.0, 2.0]
```

VS

```
In [11]: hours = []
         for minute in minutes:
             hours.append(minute/60)
         hours

Out[11]: [8.0, 9.0, 12.0, 2.0]
```

Yes...FOR LOOPS do the same thing (Example 4)

In [12]:
```
integers = [-10, 90, 15, -8, 86, -2]

change_negs = [abs(num) for num in integers if num < 0]
change_negs
```

Out[12]:
```
[10, 8, 2]
```

VS

In [14]:
```
negs = []
for num in integers:
    if num < 0:
        negs.append(abs(num))
negs
```

Out[14]:
```
[10, 8, 2]
```

Yes...FOR LOOPS do the same thing (Example 4 EXTENDED)

- What is we want to return all the numbers not just the original negative numbers...

```
In [17]: negs = []
         for num in integers:
             if num < 0:
                 negs.append(abs(num))
             else:
                 negs.append(num)
         negs

Out[17]: [10, 90, 15, 8, 86, 2]
```

Youtube Video

- Lists & Lists Comprehensions

Variable Types

- Numbers (Integers, Floats, etc.)
- Strings (e.g. “Hello World)
- Lists
- **Tuples**
- Dictionaries
- Booleans

} Collections

Syntax of Tuples

- Unlike Lists, Tuples **CANNOT** be changed **(immutable)**
- Surrounded by parenthesis **( )**
- Can be **INDEXED** like strings and lists

```
In [1]: #Tuples

        cant_change = ('Olive', 'Potato', 'Ranch')
        cant_change[0]

Out[1]: 'Olive'
```

Methods for Tuples

- **count()** - returns the # of times an item is specified in a tuple
- **index()** - searches the tuple for a specific item based on its' position

```
In [2]: cant_change = ('Olive', 'Potato', 'Ranch', 'Olive')
        cant_change.count('Olive')

Out[2]: 2
```

Variable Types

- Numbers (Integers, Floats, etc.)
- Strings (e.g. “Hello World)
- Lists
- Tuples
- **Dictionaries**
- Booleans

} Collections

Syntax for Dictionaries

- Denoted by curly braces **{ }**
- **Key - value pairs**
- Can be changed
- Can access values by calling on the **KEY**

```
In [5]: pokemon_dict = {'Name': 'Pikachu', 'Type': 'Electric'}
        pokemon_dict['Name']

Out[5]: 'Pikachu'
```

Dictionary Methods - VERY IMPORTANT

- **.keys()** - returns an array of keys
- **.values()** - returns an array of values
- **.items()** - returns an array of key-value tuples

```
In [8]: pokemon_dict = {'Name': 'Pikachu', 'Type':  'Electric'}

        print(pokemon_dict.keys())
        print(pokemon_dict.values())
        print(pokemon_dict.items())

        dict_keys(['Name', 'Type'])
        dict_values(['Pikachu', 'Electric'])
        dict_items([('Name', 'Pikachu'), ('Type', 'Electric')])
```

Zip, Dict, Range Functions

- Use the **zip function** to assign **KEYS to a LIST OF VALUES**

- **Dict function** creates a dictionary

- **Range function** is a range of numbers up to but not including that number

```
In [10]: dict(zip(range(3), ['Track', 'Soccer', 'Baseball']))

Out[10]: {0: 'Track', 1: 'Soccer', 2: 'Baseball'}
```

Dictionary Comprehension (Example 1)

- Similar to **LIST** Comprehensions
- {function for k, v in iterable <if condition>}
- **Collection = Iterable**
- k,v can be **ANY** letters or words
  - Must be the **same** letters/words **used in the function**

```
dict_names = {'First': 'James', 'Last': 'Bond'}

{k:v.upper() for k, v in dict_names.items()}
```

```
{'First': 'JAMES', 'Last': 'BOND'}
```

Dictionary Comprehension (Example 2)

- Similar to **LIST** Comprehensions
- {function for k, v in iterable <if condition>}
- **Collection = Iterable**
- k,v can be **ANY** letters or words
  - Must be the **same** letters **used in the function**

```
In [20]: dict_names = {'First': 'James', 'Last': 'Bond'}

         {k.upper():v.upper() for k, v in dict_names.items()}

Out[20]: {'FIRST': 'JAMES', 'LAST': 'BOND'}
```

Dictionary Comprehension (Example 3)

In [23]:
```
{ID: TicketNum  for ID, TicketNum in zip(range(3), range(0, 15, 5))}
```

Out[23]:
```
{0: 0, 1: 5, 2: 10}
```

Dictionary Comprehension (Example 4)

```
In [29]: princesses = ['Ariel', 'Belle', 'Mulan', 'Jasmine']

         {princess: 'Princess'  for princess in princesses}
```

```
Out[29]: {'Ariel': 'Princess',
          'Belle': 'Princess',
          'Mulan': 'Princess',
          'Jasmine': 'Princess'}
```

Dictionary of Dictionaries??

- You can have dictionary of dictionaries
- Inside dictionaries you can have lists
- **AKA you can have a collection of collections!!**
- Practice indexing these nested collections

In [33]:
```
#dictionary of dictionaries

children_pop = {
    'Disney': {'princesses': ['Ariel', 'Belle', 'Mulan', 'Jasmine'],
               'parks':  ['Magic Kingdom', 'Epcot'],
                'ID': 4},
    'Dreamworks': { 'characters':  ['Shrek', 'Donkey', 'Fiona'],
                    'parks': 'NA',
                    'ID': 5}
}

print(children_pop['Disney']['parks'][0])
```

```
Magic Kingdom
```

Dictionaries in For Loops (Example 1- KEYS)

- Very similar to Lists in For Loops

- Choose if you want to iterate through the **keys**, values, or BOTH

```
In [9]: salaries = {'1': 55000, '2': 62000, '3':  41000 }

        ID_list = []
        for k in salaries.keys():
            ID_list.append('ID'+ ' ' +k)
        ID_list

Out[9]: ['ID 1', 'ID 2', 'ID 3']
```

Dictionaries in For Loops (Example 2-VALUES)

- Very similar to Lists in For Loops

- Choose if you want to iterate through the keys, **values**, or BOTH

```
In [6]: salaries = {'1': 55000, '2': 62000, '3':  41000 }

        bonus_list = []
        for v in salaries.values():
            bonus_list.append(v*0.03)
        bonus_list

Out[6]: [1650.0, 1860.0, 1230.0]
```

Dictionaries in For Loops (Example 3-**BOTH**)

- Very similar to Lists in For Loops

- Choose if you want to iterate through the keys, values, or **BOTH**

```
In [12]: salaries = {'1': 55000, '2': 62000, '3':  41000 }

         bonus_list = []
         ID_list = []
         for k, v in salaries.items():
             ID_list.append('ID'+ ' ' +k)
             bonus_list.append(v*0.03)
             final = dict(zip(ID_list,bonus_list))
         final

Out[12]: {'ID 1': 1650.0, 'ID 2': 1860.0, 'ID 3': 1230.0}
```

Youtube Video

- Dictionaries & Dictionary Comprehensions

Variable Types

- Numbers (Integers, Floats, etc.)
- Strings (e.g. “Hello World)
- Lists
- Tuples
- Dictionaries
- **Booleans**

} Collections

Booleans

- Conditional

- Two Values
  - **True (1)**
  - **False (0)**

```
In [13]: x = True

         type(x)

Out[13]: bool
```

Day 2: Statistical Measures

Agenda

- Numpy Library (Arrays vs. Lists)
- Pandas Library
- Median vs. Mean
- Outliers
- Dispersion (Standard Deviation)
- Covariance vs. Correlation

Numpy Library

- Provides us with the data type of **ARRAYS**
- Items should be of **SAME** type
- Better than lists in performing arithmetic ops (addition, multiplying, etc.)

Arrays vs. List

- Can perform arithmetic operations on arrays

- Arrays take up less memory

```
#import the numpy and pandas libraries

import numpy as np
import pandas as pd
```

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

```
x = x + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-38b88bc0b147> in <module>
----> 1 x = x + 3
      2 y = y + 3

TypeError: can only concatenate list (not "int") to list
```

```
y = y + 3
print(y)
```

```
[4 5 6]
```

Jupyter Notebook Question #9

Multi-Dimensional Arrays & .shape

```
multi = np.array([[1, 2, 3], [6, 7, 8]])
print(multi)
print(multi.shape)
print(type(multi))
```

```
[[1 2 3]
 [6 7 8]]
(2, 3)
<class 'numpy.ndarray'>
```

Jupyter Notebook Question #10

Pandas Library

- Popular library for data analysis and data manipulation
- Can convert arrays, lists, etc to DATAFRAMES :)
- Open csv/txt files
- Explore those DATAFRAMES (.head, .info, etc.)

Example of Pandas in Use

```
#read in the cars.csv dataset
import pandas as pd
cars = pd.read_csv('data/cars.csv')
```

Jupyter Notebook Question #11

Measures of Central Tendency

- Sample Mean (X bar)
- Population Mean (mu - **μ**)
  - Both calculated the same (sum/total number of observations (**N**))
  - However **MOST** of the time you have a sample not the true population

Sample or Population Examples

- Surveyed **10** retirement homes to make an inference about the nations retirement home conditions

- Called **2,000** households to see how they would vote in the upcoming elections to determine the nation’s next president

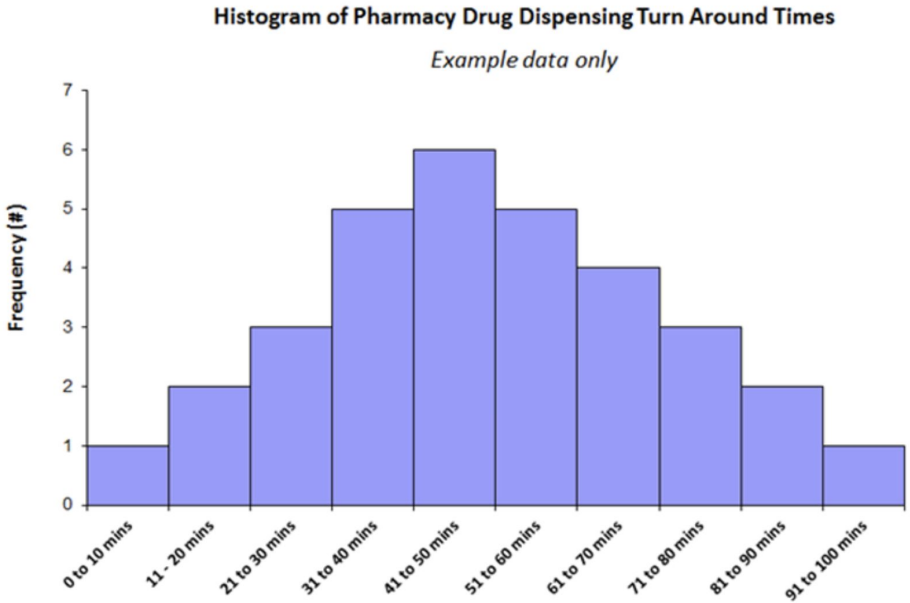
- You get **100% response rate** from all of your employees to make conclusions about your company’s working conditions

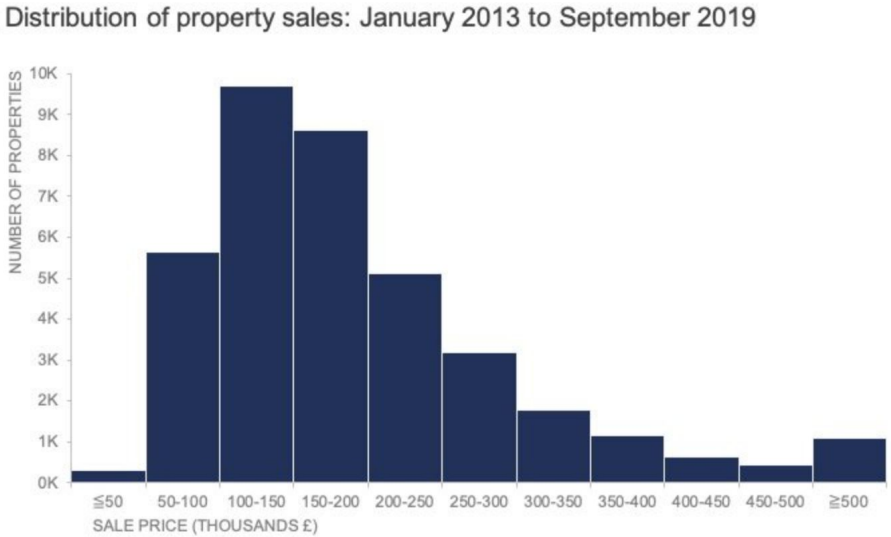
Median or Mean?

- Median - exact middle location of a distribution

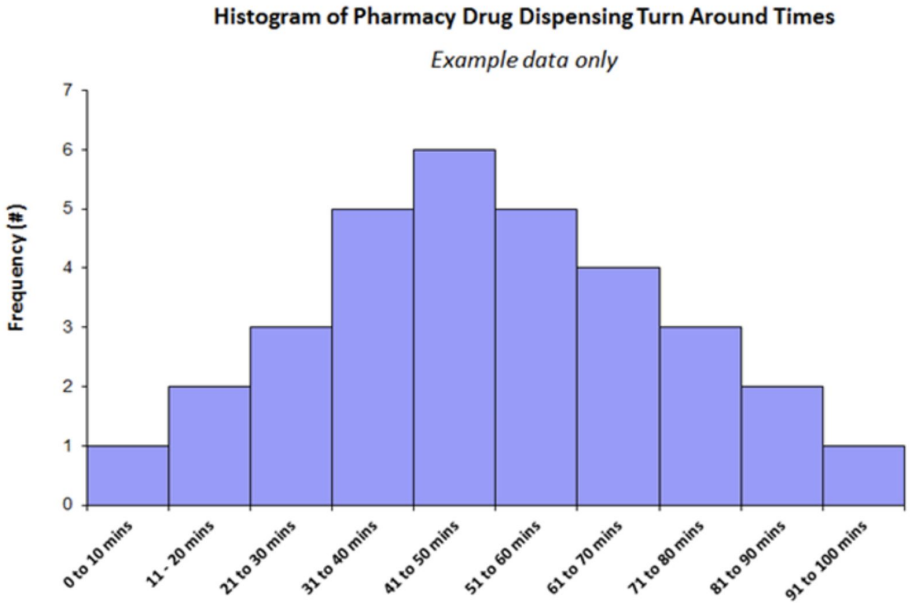
- **ROBUST (non-sensitive) to outliers**
  - Preferred measure of central tendency when your data has tons of outliers

Median or Mean? Visualize IT!

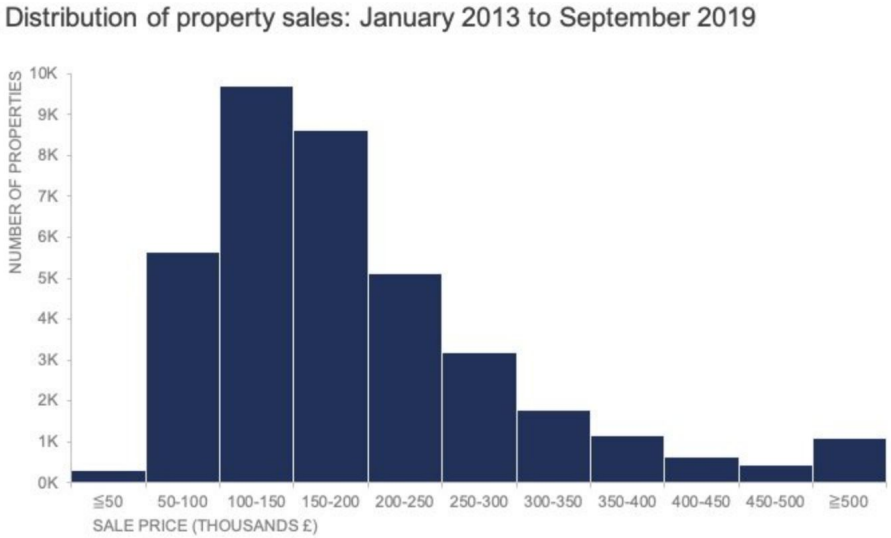




Median or Mean? Visualize IT!



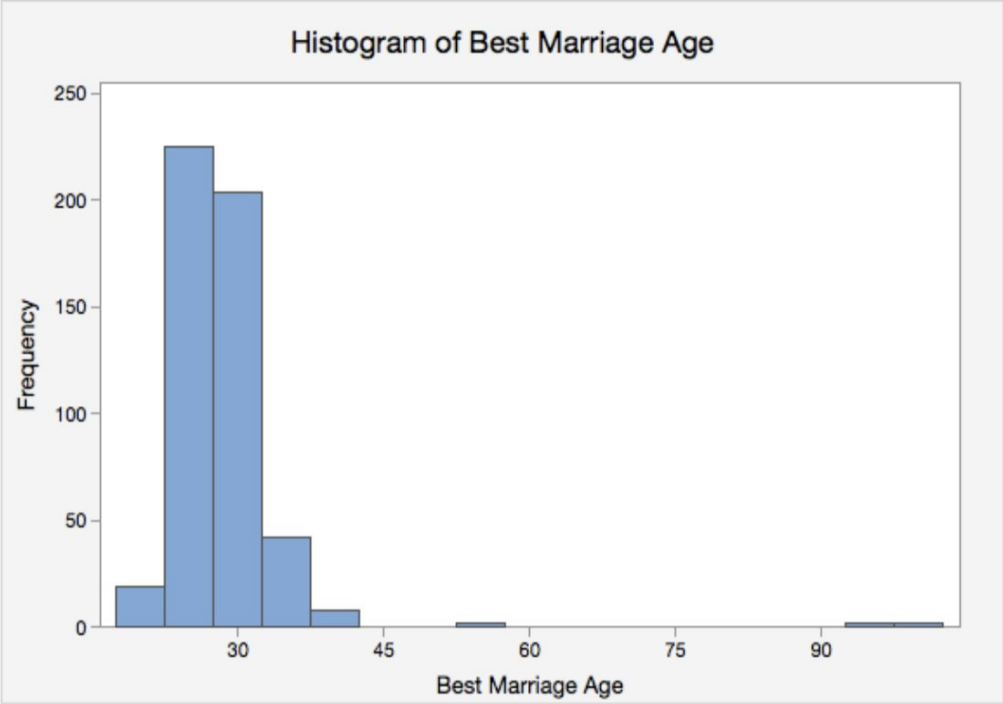
Symmetric : Mean, Median, Mode are EQUAL



Skewed: Median or Mode may be your go to here

Outliers - What to do?

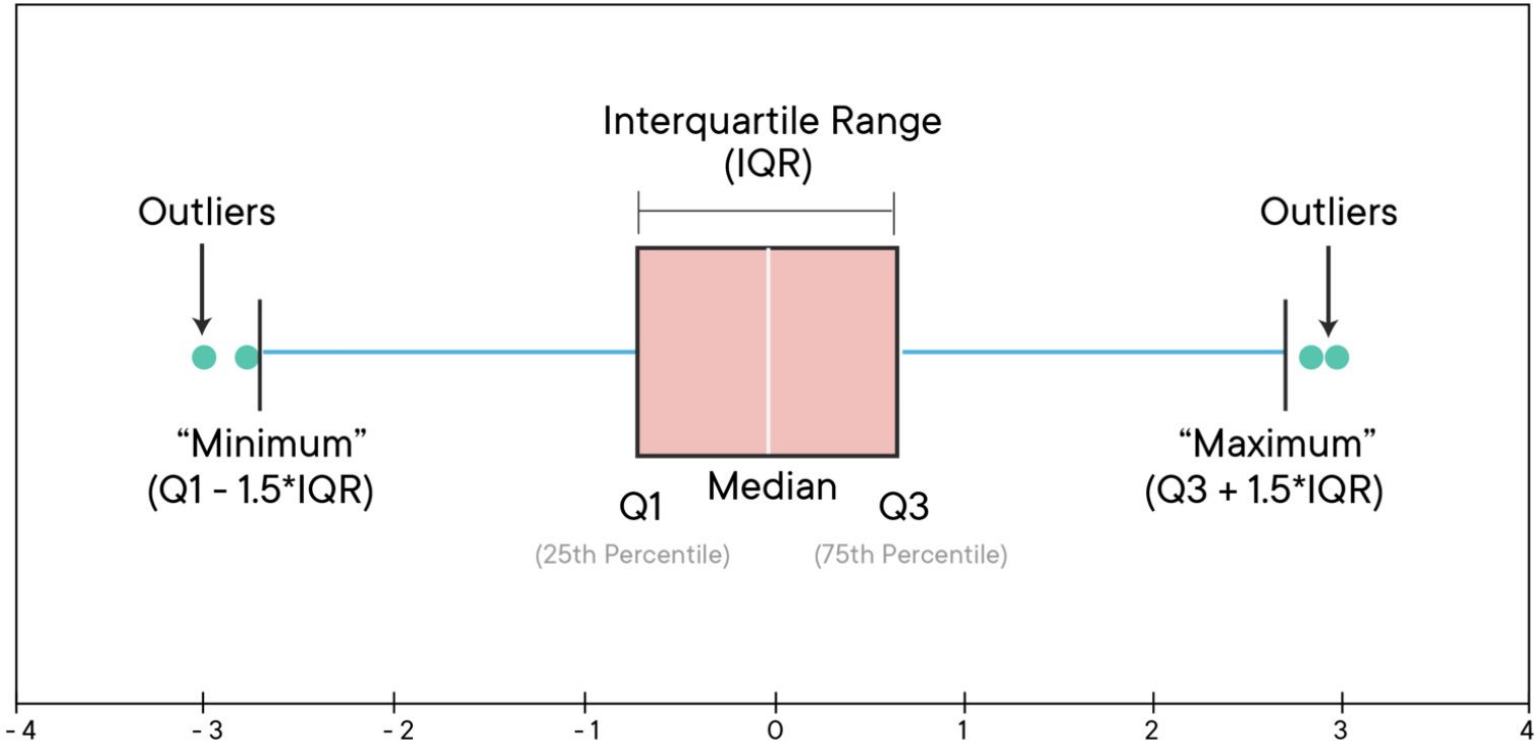
- Median Imputation
- Quantile based flooring or capping
  - (e.g. Any value above 90th percentile gets the 90th percentile value)
- Transforming the Data
- Drop them
  - If you **KNOW** it is a data error drop them
  - Do not remove data just because



Dispersion

- **Absolute deviation** - Absolute value of the actual value minus the mean
- **Variance** - Sum of Squared deviations (**lose your units** – units are squared)
  - High - values are spread out from the Mean
  - Low - values are clustered around the Mean
- **Standard Deviation** - spread of values within a dataset (**square root of Variance)**
  - Keep your units!

Visualize Dispersion!

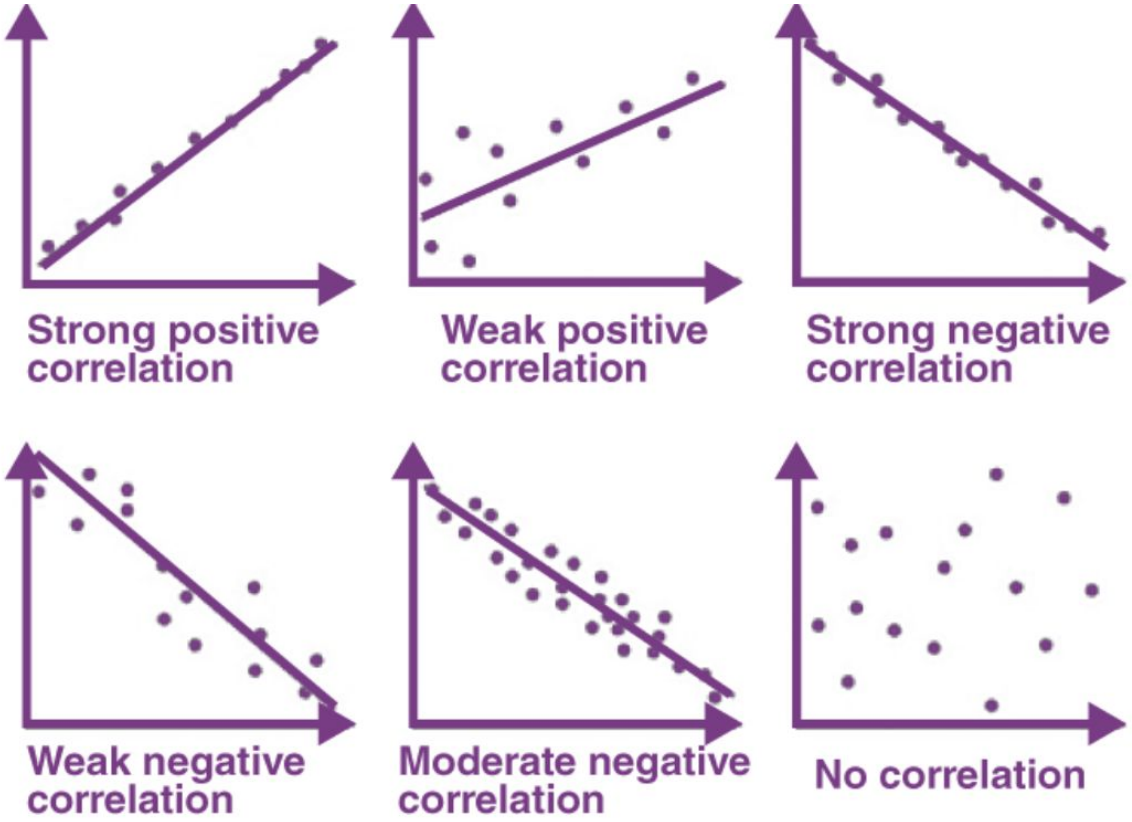


Covariance & Correlation

- **Covariance** - How two variables ‘vary’ together
  - Ranges from positive to negative infinity
  - Used for dimensionality reduction

- **Correlation** - Degree in which they vary (normalized form of covariance)
  - Pearson Correlation Coefficient **-1 to 1**
  - > 0.7 these two variables are considered highly correlated
    - Drop one of these vars?

Correlation Image



YouTube Video

- https://youtu.be/y75B05CU7S8

Day 3: Data Visualization

Agenda

- Matplotlib
- Making a Scatterplot
- Making a Histogram
- DOs and Don’ts of Visuals
- Seaborn

Matplotlib Library & Pyplot Module

- Matplotlib.pyplot > great for visualizing data

- Set up a figure and an axes object using
  - Fig, ax = plt.subplots()
    - Produces a blank box to put your visual in :)

Jupyter Notebook: Scatterplot Example (.scatter)

- Go to Jupyter Notebook and create a scatterplot of City MPG vs. Highway MPG using the cars dataset

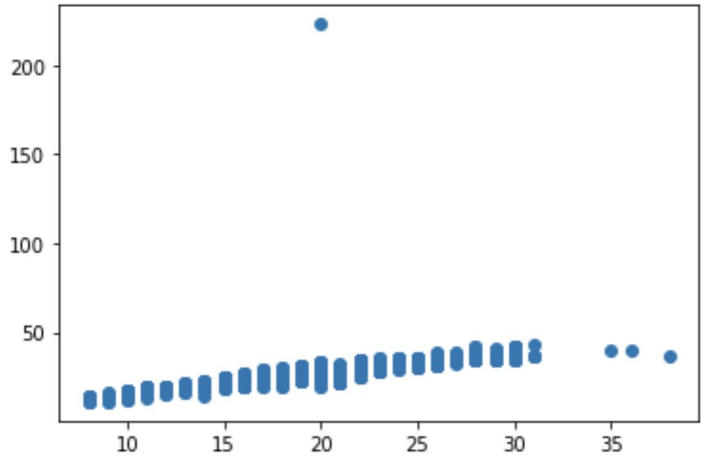
Scatterplot Answer

In [13]:
```
#Create the plot

fig, ax = plt.subplots()
x = cars['Fuel Information.City mpg']
y = cars['Fuel Information.Highway mpg']

ax.scatter(x,y)
```

Out[13]:
```
<matplotlib.collections.PathCollection at 0x7f8553277280
```

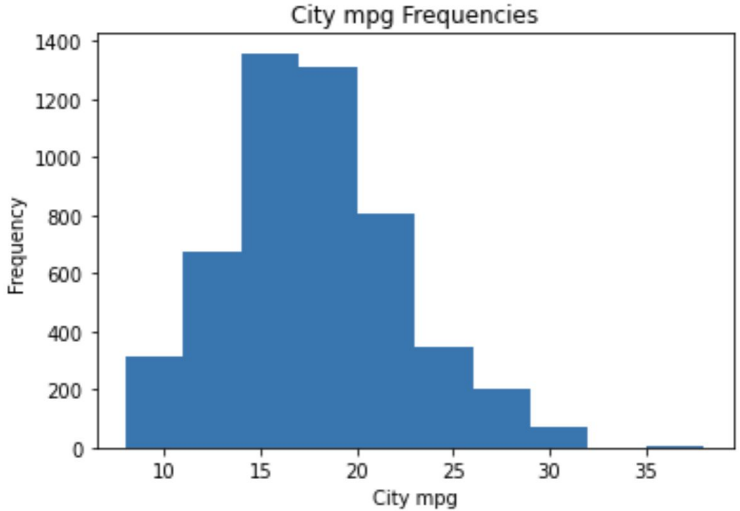


Jupyter Notebook: Histogram Example (.hist)

- Create a Histogram of the Fuel Information.City mpg column in the cars.csv dataset

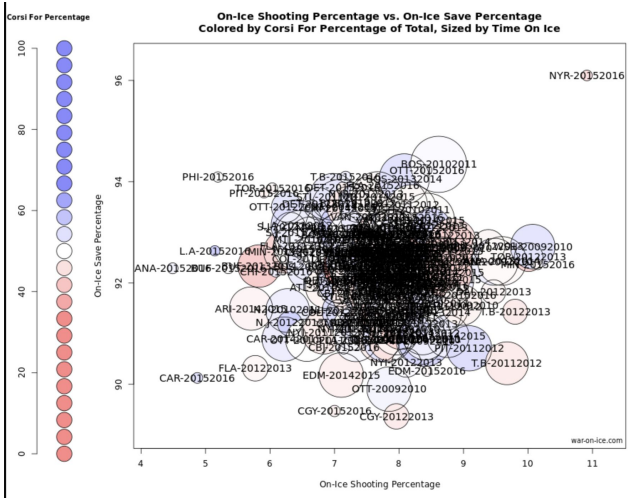
Histogram Answer

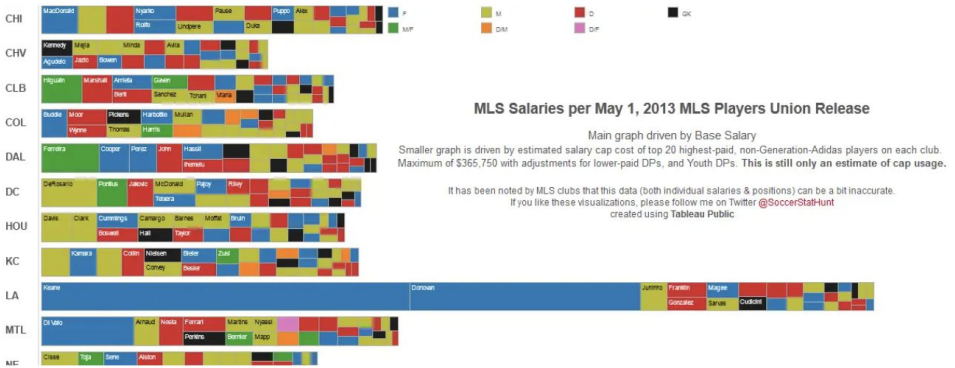
```
fig, ax = plt.subplots()
x = cars['Fuel Information.City mpg']
ax.hist(x)
ax.set_title('City mpg Frequencies')
ax.set_xlabel('City mpg')
ax.set_ylabel('Frequency')
plt.show()
```



Dont's of Visualizations

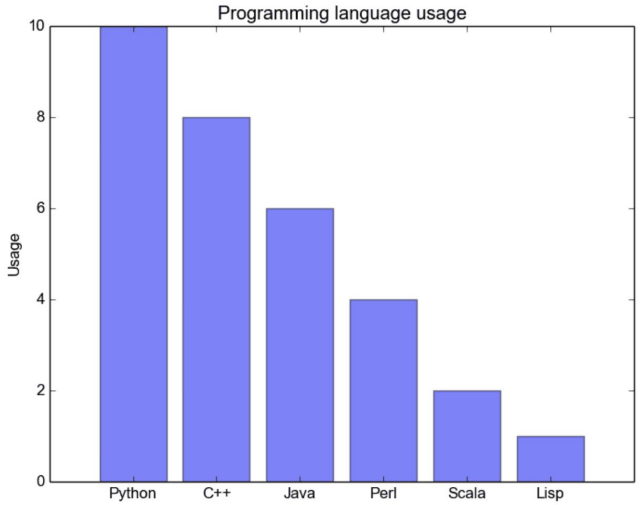
- Make visuals 'busy' and hard to read
- Not label your axes (unless it is obvious like 'Date')
- Use too many colors or 'ombre' effects
- Use red and green or blue and green in the same visual
- Make your visuals too tiny

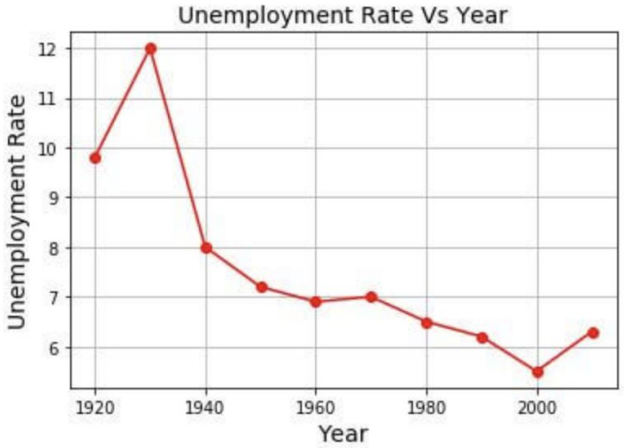




DO's of Visualizations

- Use the appropriate visual for the data
  - I.e. Two numeric columns – scatterplot
- Label your axes
- Format your axes so that it is easy to read
  - 540000 vs. $54,000
- Use few colors
  - Bars don't need to be 3 different colors
- Make it span the entire slide
  - Tiny visuals get you nowhere
- Use call-out boxes





Seaborn

- Library built on top of matplotlib to make visuals more eye-catching and professional
- import seaborn as sns
- Documentation Here:
  - https://seaborn.pydata.org/