project2

November 14, 2024

[48]:
```
# Initialize Otter
import otter
grader = otter.Notebook("project2.ipynb")
```

1 Project 2: Climate Change—Temperatures and Precipitation

In this project, you will investigate data on climate change, or the long-term shifts in temperatures and weather patterns!

1.0.1 Logistics

Deadline. This project is due at **5:00pm PT on Friday, 11/15**. You can receive 5 bonus points for submitting the project by **5:00pm PT on Thursday, 11/14.** Projects submitted fewer than 24 hours after the deadline will receive 80% credit. Any submissions later than 24 hours after the deadline will not be accepted. It's **much** better to be early than late, so start working now.

Checkpoint. For full credit on the checkpoint, you must complete the questions up to the checkpoint, **pass all *public* autograder tests** for those sections, and submit to the Gradescope Project 2 Checkpoint assignment by **5:00pm PT on Friday, 11/08**. **The checkpoint is worth 5% of your entire project grade**. After you've submitted the checkpoint, you may still change your project answers before the final project deadline - **only your final submission, to the "Project 2" assignment, will be graded for correctness** (including questions from before the checkpoint). You will have some lab time to work on these questions, but we recommend that you start the project before lab and leave time to finish the checkpoint afterward.

Partners. You may work with one other partner; your partner must be from your assigned lab section. **Only one partner should submit the project notebook to Gradescope. If both partners submit, you will be docked 10% of your project grade. On Gradescope, the person who submits should also designate their partner so that both of you receive credit.** Once you submit, click into your submission, and there will be an option to Add Group Member in the top right corner. You may also reference this walkthrough video on how to add partners on Gradescope.

Rules. Don't share your code with anybody but your partner. You are welcome to discuss questions with other students, but don't share the answers. The experience of solving the problems in this project will prepare you for exams (and life). If someone asks you for the answer, resist! Instead, you can demonstrate how you would solve a similar problem.

Support. You are not alone! Come to office hours, post on Ed, and talk to your classmates. If you want to ask about the details of your solution to a problem, make a private Ed post and the staff will respond. If you're ever feeling overwhelmed or don't know how to make progress, email your TA or tutor for help. You can find contact information for the staff on the course website.

Tests. The tests that are given are **not comprehensive** and passing the tests for a question **does not** mean that you answered the question correctly. Tests usually only check that your table has the correct column labels. However, more tests will be applied to verify the correctness of your submission in order to assign your final score, so be careful and check your work! You might want to create your own checks along the way to see if your answers make sense. Additionally, before you submit, make sure that none of your cells take a very long time to run (several minutes).

Free Response Questions: Make sure that you put the answers to the written questions in the indicated cell we provide. **Every free response question should include an explanation** that adequately answers the question.

Advice. Develop your answers incrementally. To perform a complicated task, break it up into steps, perform each step on a different line, give a new name to each result, and check that each intermediate result is what you expect. You can add any additional names or functions you want to the provided cells. Make sure that you are using distinct and meaningful variable names throughout the notebook. Along that line, **DO NOT** reuse the variable names that we use when we grade your answers.

You **never** have to use just one line in this project or any others. Use intermediate variables and multiple lines as much as you would like!

All of the concepts necessary for this project are found in the textbook. If you are stuck on a particular problem, reading through the relevant textbook section will often help to clarify concepts.

To get started, load `datascience`, `numpy`, and `matplotlib`. Make sure to also run the first cell of this notebook to load `otter`.

[49]:
```
# Run this cell to set up the notebook, but please don't change it.
from datascience import *
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
np.set_printoptions(legacy='1.13')

import warnings
warnings.simplefilter('ignore')

import plotly.graph_objects as go
```

1.1 Part 1: Temperatures

In the following analysis, we will investigate one of the 21st century's most prominent issues: climate change. While the details of climate science are beyond the scope of this course, we can start to learn about climate change just by analyzing public records of different cities' temperature and precipitation over time.

We will analyze a collection of historical daily temperature and precipitation measurements from weather stations in 210 U.S. cities. The dataset was compiled by Yuchuan Lai and David Dzombak [1]; a description of the data from the original authors and the data itself is available here.

[1] Lai, Yuchuan; Dzombak, David (2019): Compiled historical daily temperature and precipitation data for selected 210 U.S. cities. Carnegie Mellon University. Dataset.

1.1.1 Part 1, Section 1: Cities

Run the following cell to load information about the `cities` and preview the first few rows.

```
[50]: cities = Table.read_table('city_info.csv', index_col=0)
      cities.show(3)
```

```
<IPython.core.display.HTML object>
```

The `cities` table has one row per weather station and the following columns:

1. `"Name"`: The name of the US city
2. `"ID"`: The unique identifier for the US city
3. `"Lat"`: The latitude of the US city (measured in degrees of latitude)
4. `"Lon"`: The longitude of the US city (measured in degrees of longitude)
5. `"Stn.Name"`: The name of the weather station in which the data was collected
6. `"Stn.stDate"`: A string representing the date of the first recording at that particular station
7. `"Stn.edDate"`: A string representing the date of the last recording at that particular station

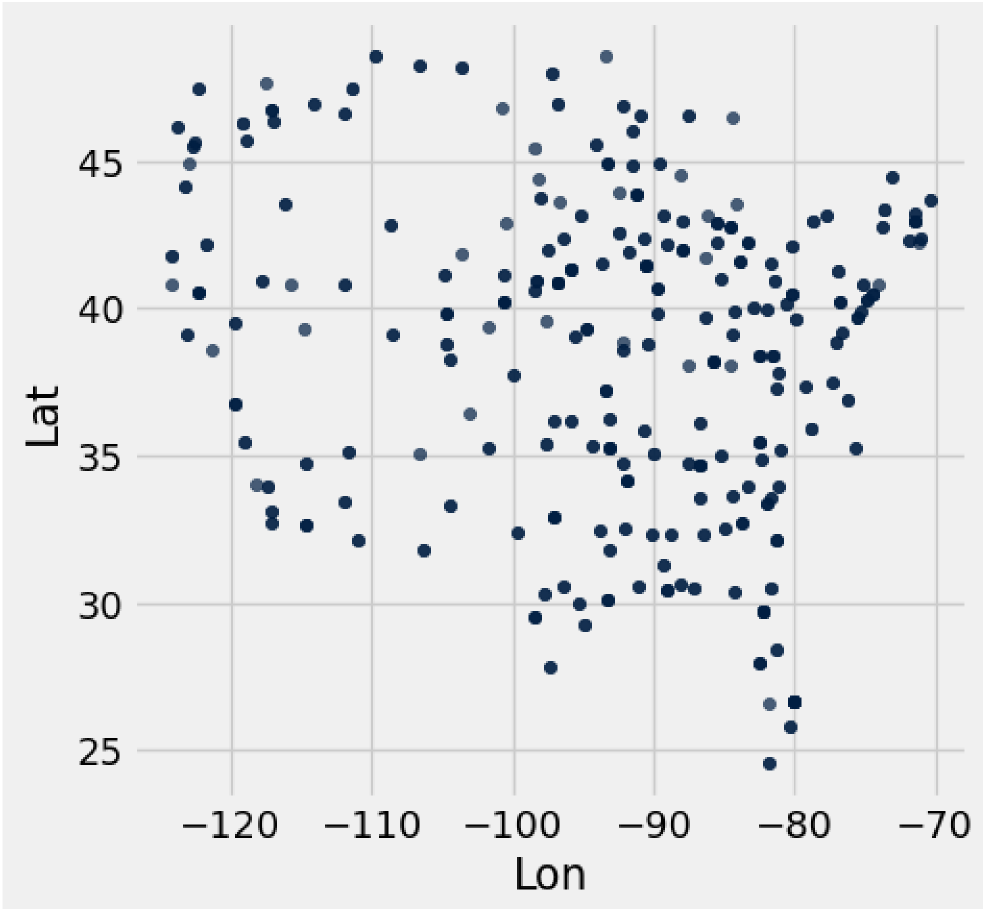
The data lists the weather stations at which temperature and precipitation data were collected. Note that although some cities have multiple weather stations, only one is collecting data for that city at any given point in time. Thus, we are able to just focus on the cities themselves.

Question 1.1.1: In the cell below, produce a scatter plot that plots the latitude and longitude of every city in the `cities` table so that the result places northern cities at the top and western cities at the left.

Note: It's okay to plot the same point multiple times!

Hint: A latitude is the set of horizontal lines that measures distances *north or south* of the equator. A longitude is the set of vertical lines that measures distances *east or west* of the prime meridian.

```
[51]: sorted_cities_table = cities.sort("Lat").sort("Lon")
      sorted_cities_table.scatter("Lon", "Lat")
```

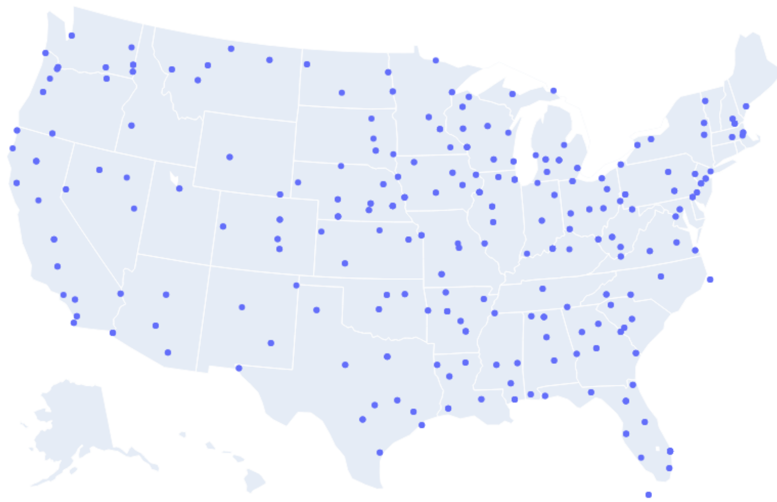


These cities are all within the continental U.S., and so the general shape of the U.S. should be visible in your plot. The shape will appear distorted compared to most maps for two reasons: the scatter plot is square even though the U.S. is wider than it is tall, and this scatter plot is an equirectangular projection of the spherical Earth. A geographical map of the same data uses the common Pseudo-Mercator projection.

Note: If this visualization doesn't load for you, please view a version of it online here.

[52]:
```
# Just run this cell
fig = go.Figure(data=go.Scattergeo(
        lon = cities.column('Lon'),
        lat = cities.column('Lat'),
        text = cities.column('Name'),
        mode = 'markers'))

fig.update_layout(geo_scope='usa', width=800, height=500, margin={'r': 0, 't':␣
 ↪0, 'l': 0, 'b': 0})
fig.show()
```



Question 1.1.2 Does it appear that these city locations are sampled uniformly at random from all land in the continental U.S.? Why or why not?

No, because the city locations are clustered towards the eastern half of the continental US. However this may be because the western half of the continental US has less population density than the east.

Question 1.1.3: Assign `num_unique_cities` to the number of unique cities that appear in the `cities` table.

[53]:
```
num_unique_cities = cities.group("Name").num_rows #210

# Do not change this line
print(f"There are {num_unique_cities} unique cities that appear within our␣
↪dataset.")
```

```
There are 210 unique cities that appear within our dataset.
```

[54]:
```
grader.check("q1_1_3")
```

[54]:
```
q1_1_3 results: All test cases passed!
```

In order to investigate further, it will be helpful to determine what region of the United States each city was located in: Northeast, Northwest, Southeast, or Southwest. For our purposes, we will be using the following geographical boundaries:

1. A station is located in the `"Northeast"` region if its latitude is above or equal to 40 degrees and its longtitude is greater than or equal to -100 degrees.
2. A station is located in the `"Northwest"` region if its latitude is above or equal to 40 degrees and its longtitude is less than -100 degrees.

3. A station is located in the "Southeast" region if its latitude is below 40 degrees and its longtitude is greater than or equal to -100 degrees.
4. A station is located in the "Southwest" region if its latitude is below 40 degrees and its longtitude is less than -100 degrees.

Question 1.1.4: Define the `coordinates_to_region` function below. It should take in two arguments, a city's latitude (`lat`) and longitude (`lon`) coordinates, and output a string representing the region it is located in.

[55]:
```
def coordinates_to_region(lat, lon):
    if lat >= 40: # northern city
        if lon >= -100: # eastern city
            return "Northeast"
        elif lon < -100:
            return "Northwest"
    elif lat < 40: # southern city
        if lon >= -100: # eastern city
            return "Southeast"
        elif lon < -100:
            return "Southwest"
```

[56]:
```
grader.check("q1_1_4")
```

[56]:
```
q1_1_4 results: All test cases passed!
```

Question 1.1.5: Add a new column in `cities` labeled `Region` that contains the region in which the city is located. For full credit, you must use the `coordinates_to_region` function you defined rather than reimplementing its logic.

[57]:
```
regions_array = cities.apply(coordinates_to_region, "Lat", "Lon")
cities = cities.with_column("Region", regions_array)
cities.show(5)
```

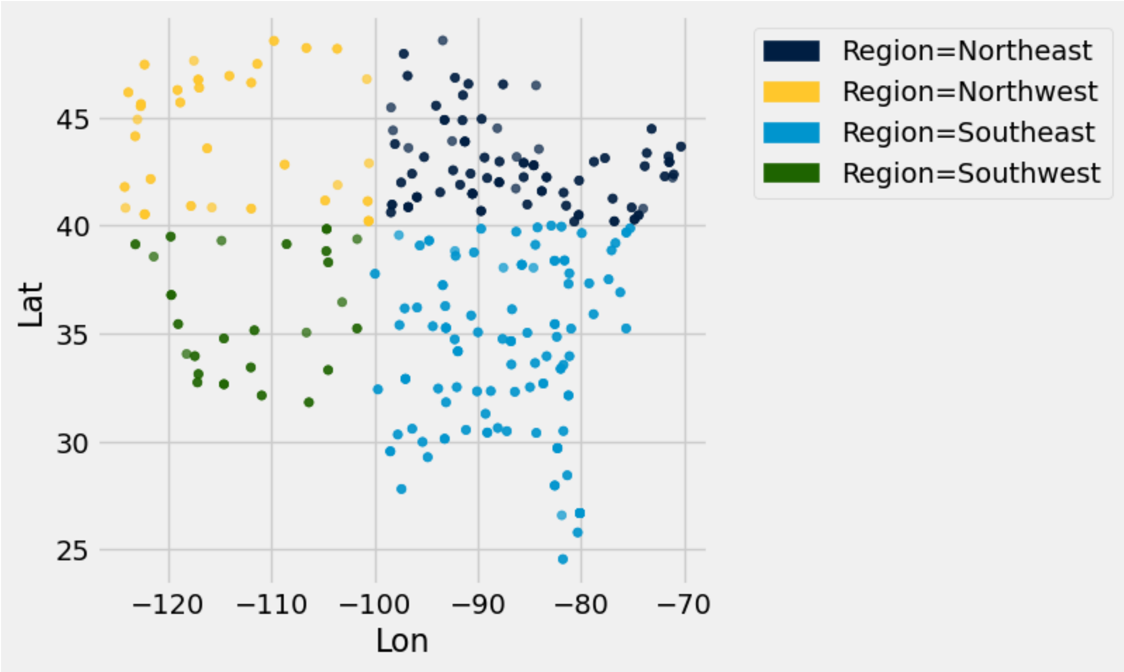
```
<IPython.core.display.HTML object>
```

[58]:
```
grader.check("q1_1_5")
```

[58]:
```
q1_1_5 results: All test cases passed!
```

To confirm that you've defined your `coordinates_to_region` function correctly and successfully added the `Region` column to the `cities` table, run the following cell. Each region should have a different color in the result.

[59]:
```
# Just run this cell
cities.scatter("Lon", "Lat", group="Region")
```



Challenge Question 1.1.6 (OPTIONAL, ungraded): Create a new table called `cities_nearest`. It should contain the same columns as the `cities` table and an additional column called `"Nearest"` that contains the **name of the nearest city** that is in a different region from the city described by the row.

To approximate the distance between two cities, take the square root of the sum of the squared difference between their latitudes and the square difference between their longitudes. **Don't use a `for` statement; instead, use the `apply` method and array arithmetic.**

Hint: We have defined a `distance` function for you, which can be called on numbers `lat0` and `lon0` and arrays `lat1` and `lon1`.

[60]:
```
def distance(lat0, lon0, lat1, lon1):
    "Approximate the distance between point (lat0, lon0) and (lat1, lon1) pairs␣
 ↪in the arrays."
    return np.sqrt((lat0 - lat1) * (lat0 - lat1) + (lon0 - lon1) * (lon0 -␣
 ↪lon1))


...
# each city has a "nearest city"
# a "nearest city" is in a different region from the sister city




nearest_array = ...
```

```
cities_nearest = cities.with_column("Nearest", nearest_array)
# Note: remove the comment(#) on the next line if you choose to do this question
#cities_nearest.show(5)
```

1.1.2 Part 1, Section 2: Welcome to Phoenix, Arizona

Each city has a different CSV file full of daily temperature and precipitation measurements. The file for Phoenix, Arizona is included with this project as `phoenix.csv`. The files for other cities can be downloaded here by matching them to the ID of the city in the `cities` table.

Since Phoenix is located on the upper edge of the Sonoran Desert, it has some impressive temperatures.

Run the following cell to load in the `phoenix` table. It has one row per day and the following columns:

1. `"Date"`: The date (a string) representing the date of the recording in **YYYY-MM-DD** format
2. `"tmax"`: The maximum temperature for the day (°F)
3. `"tmin"`: The minimum temperature for the day (°F)
4. `"prcp"`: The recorded precipitation for the day (inches)

[61]:
```
phoenix = Table.read_table("phoenix.csv", index_col=0)
phoenix.show(3)
```

```
<IPython.core.display.HTML object>
```

Question 1.2.1: Assign the variable `largest_2010_range_date` to the date of the **largest temperature range** in Phoenix, Arizona for any day between January 1st, 2010 and December 31st, 2010. To get started, use the variable `phoenix_with_ranges_2010` to filter the phoenix table to days in 2010 with an additional column corresponding to the temperature range for that day.

Your answer should be a string in the "YYYY-MM-DD" format. Feel free to use as many lines as you need. A temperature range is calculated as the difference between the max and min temperatures for the day.

Hint: To limit the values in a column to only those that *contain* a certain string, pick the right `are.` predicate from the Python Reference Sheet.

Note: Do **not** re-assign the `phoenix` variable; please use the `phoenix_with_ranges_2010` variable instead.

[62]:
```
def calc_temp_range(max, min):
    return abs(max-min)


phoenix_with_ranges_2010 = phoenix.where("Date", are.containing("2010"))
```

```python
temp_ranges_for_2010_array = phoenix_with_ranges_2010.apply(calc_temp_range,␣
↪"tmax", "tmin")

pwr = phoenix_with_ranges_2010.with_column("Ranges", temp_ranges_for_2010_array)

guh = pwr.where("Ranges", are.equal_to( int(np.max(pwr.column("Ranges")))))

largest_2010_range_date = guh.column("Date").item(0)
largest_2010_range_date
```

[62]: '2010-06-24'

[63]:
```python
grader.check("q1_2_1")
```

[63]: q1_2_1 results: All test cases passed!

We can look back to our `phoenix` table to check the temperature readings for our `largest_2010_range_date` to see if anything special is going on. Run the cell below to find the row of the `phoenix` table that corresponds to the date we found above.

[64]:
```python
# Just run this cell
phoenix.where("Date", largest_2010_range_date)
```

[64]:
```
Date       | tmax | tmin | prcp
2010-06-24 | 113  | 79   | 0
```

ZOO WEE MAMA! Look at the maximum temperature for that day. That's hot.

The function `extract_year_from_date` takes a date string in the **YYYY-MM-DD** format and returns an integer representing the **year**. The function `extract_month_from_date` takes a date string and returns a string describing the month. Run this cell, but you do not need to understand how this code works or edit it.

[65]:
```python
# Just run this cell
import calendar

def extract_year_from_date(date):
    """Returns an integer corresponding to the year of the input string's date.
↪"""
    return int(date[:4])

def extract_month_from_date(date):
    "Return an abbreviation of the name of the month for a string's date."
    month = date[5:7]
    return f'{month} ({calendar.month_abbr[int(date[5:7])]})'


# Example
```

```
print('2022-04-01 has year', extract_year_from_date('2022-04-01'),
      'and month', extract_month_from_date('2022-04-01'))
```

```
2022-04-01 has year 2022 and month 04 (Apr)
```

Question 1.2.2: Add two new columns called `Year` and `Month` to the `phoenix` table that contain the year as an **integer** and the month as a **string** (such as `"04 (Apr)"`) for each day, respectively.

Note: The functions above may be helpful!

[66]:
```
years_array = phoenix.apply(extract_year_from_date, "Date")
months_array = phoenix.apply(extract_month_from_date, "Date")

phoenix = phoenix.with_columns("Year", years_array, "Month", months_array)

phoenix.show(5)
```

```
<IPython.core.display.HTML object>
```

[67]:
```
grader.check("q1_2_2")
```

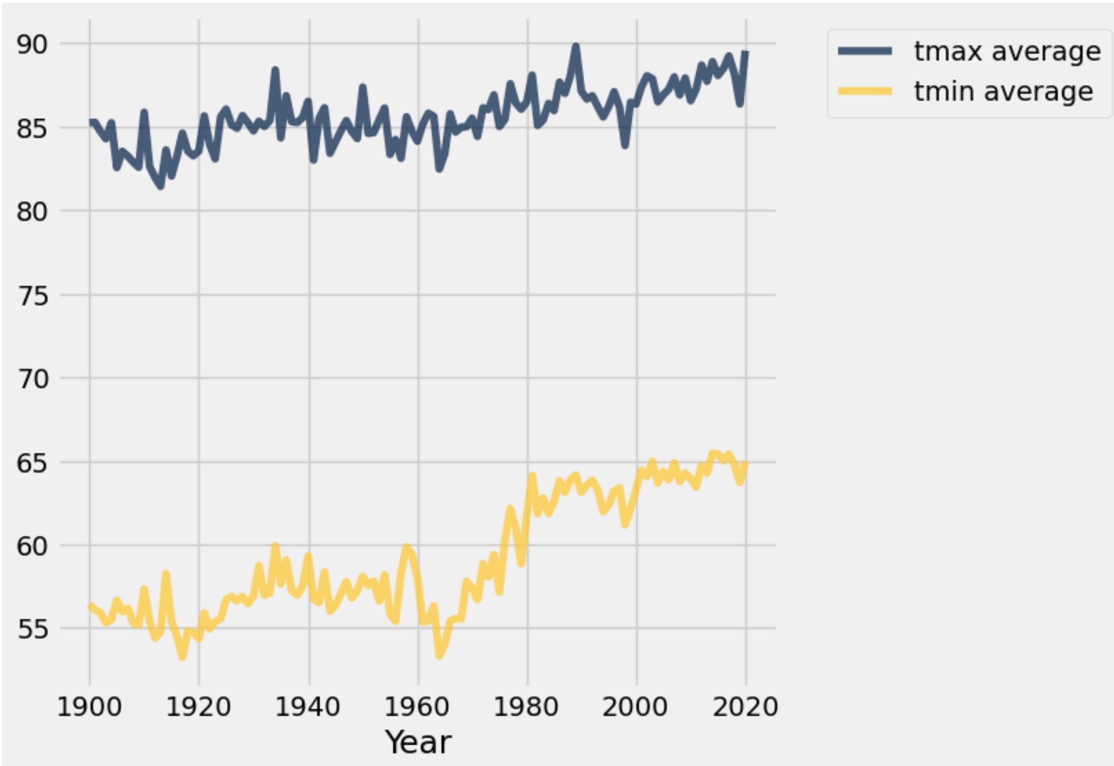
[67]: q1_2_2 results: All test cases passed!

Question 1.2.3: Using the `phoenix` table, create an overlaid line plot of the **average maximum temperature** and **average minimum temperature** for each year between 1900 and 2020 (both ends inclusive).

Hint: To draw a line plot with more than one line, call `plot` on the column label of the x-axis values and all other columns will be treated as y-axis values.

[68]:
```
# inclusive range of years 1900 through 2020
# 2 line plots
# a) avg max temp
# b) avg min temp
phoenix_1900_through_2020 = (phoenix.
                             where("Year", are.between_or_equal_to(1900, 2020)).
                             select("tmax", "tmin", "Year").
                             group("Year", np.average))

line_plot_of_yearly_avg_temp_1900_through_2020 = phoenix_1900_through_2020.
  ↪plot("Year")
line_plot_of_yearly_avg_temp_1900_through_2020
```



Question 1.2.4: Although still hotly debated (pun intended), many climate scientists agree that the effects of climate change began to surface in the early 1960s as a result of elevated levels of greenhouse gas emissions. How does the graph you produced in Question 1.2.3 support the claim that modern-day global warming began in the early 1960s?

The graph supports the claim that modern-day global warming began in the early 1960s, because it shows that beginning in the early 60s a trend of average maximum and minimum temperatures rising above past decades and staying there.

Averaging temperatures across an entire year can obscure some effects of climate change. For example, if summers get hotter but winters get colder, the annual average may not change much. Let's investigate how average **monthly** maximum temperatures have changed over time in Phoenix.

Question 1.2.5: Create a `monthly_increases` table with one row per month and the following four columns in order: 1. `"Month"`: The month (such as `"02 (Feb)"`) 2. `"Past"`: The average max temperature in that month from 1900-1960 (both ends inclusive) 3. `"Present"`: The average max temperature in that month from 2019-2021 (both ends inclusive) 4. `"Increase"`: The difference between the present and past average max temperatures in that month

First, make a copy of the `phoenix` table and add a new column containing the corresponding **period** for each row. The period refers to whether the year is in the `"Past"`, `"Present"`, or `"Other"` category. You may find the period function helpful to see which years correspond to each period. Then, use this new table to construct `monthly_increases`. Feel free to use as many lines as you need.

Hint: What table method can we use to get each **unique value** as its own column?

Note: Please do **not** re-assign the `phoenix` variable!

[69]:
```
def period(year):
    "Output if a year is in the Past, Present, or Other."
    if 1900 <= year <= 1960:
        return "Past"
    elif 2019 <= year <= 2021:
        return "Present"
    else:
        return "Other"


def calc_increase(past, present):
    return abs(present - past)


# Step A
phoenix_copy_table = phoenix.sort("Year")
period_array = phoenix_copy_table.apply(period, "Year")
phoenix_copy_table = phoenix_copy_table.with_column("Period", period_array)
# test
#phoenix_copy_table.show(3)


# Step B: construct monthly_increases table
"""
Month | Past | Present | Increase
01 Jan
02 Feb
03 Mar
"""
# Past: The average max temperature in that month from 1900-1960 (both ends␣
↪inclusive)
past_table = (phoenix_copy_table.
        where("Year", are.between_or_equal_to(1900, 1960)).
        select("Period", "Month", "tmax").
        pivot("Period", "Month", "tmax", np.mean))
# test
#past_table.show(5)

# Present: The average max temperature in that month from 2019-2021 (both ends␣
↪inclusive)
present_table = (phoenix_copy_table.
```

```
        where("Year", are.between_or_equal_to(2019, 2021)).
        select("Period", "Month", "tmax").
        pivot("Period", "Month", "tmax", np.mean))
# test
present_table.show(5)

# Increase
temp_table = past_table.join("Month", present_table)
temp_increase_array = temp_table.apply(calc_increase, "Past", "Present")
temp_table = temp_table.with_column("Increase", temp_increase_array)
# test
#temp_table.show(12)


monthly_increases = temp_table.with_column("Increase", temp_increase_array)
monthly_increases.show()
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[70]:
```
grader.check("q1_2_5")
```

[70]:
```
q1_2_5 results: All test cases passed!
```

1.1.3 February in Phoenix

The "Past" column values are averaged over many decades, and so they are reliable estimates of the average high temperatures in those months before the effects of modern climate change. However, the "Present" column is based on only three years of observations. February, the shortest month, has the fewest total observations: only 85 days. Run the following cell to see this.

[71]:
```
# Just run this cell
feb_present = phoenix.where('Year', are.between_or_equal_to(2019, 2021)).
 ↪where('Month', '02 (Feb)')
feb_present.num_rows
```

[71]:
```
85
```

Look back to your monthly_increases table. Compared to the other months, the increase for the month of February is quite small; the February difference is very close to zero. Run the following cell to print out our observed difference.

[72]:
```
# Just run this cell
print(f"February Difference: {monthly_increases.row(1).item('Increase')}")
```

```
February Difference: 0.3373623297258632
```

Perhaps that small difference is somehow due to chance! To investigate this idea requires a thought experiment.

We can observe all of the February maximum temperatures from 2019 to 2021 (the present period), so we have access to the census; there's no random sampling involved. But, we can imagine that if more years pass with the same present-day climate, there would be different but similar maximum temperatures in future February days. From the data we observe, we can try to estimate the **average maximum February temperature** in this imaginary collection of all future February days that would occur in our modern climate, assuming the climate doesn't change any further and many years pass.

We can also imagine that the maximum temperature each day is like a **random draw from a distribution of max daily temperatures for that month**. Treating actual observations of natural events as if they were each *randomly* sampled from some unknown distribution is a simplifying assumption. These temperatures were not actually sampled at random—instead they occurred due to the complex interactions of the Earth's climate—but treating them as if they were random abstracts away the details of this naturally occuring process and allows us to carry out statistical inference. Conclusions are only as valid as the assumptions upon which they rest, but in this case thinking of daily temperatures as random samples from some unknown climate distribution seems at least plausible.

If we assume that the **actual temperatures were drawn at random from some large population of possible February days** in our modern climate, then we can not only estimate the population average of this distribution, but also quantify our uncertainty about that estimate using a confidence interval.

We will now compute the confidence interval of the present February average max daily temperature. To unpack this statement, we are saying that this confidence interval is looking at present-day February conditions (think about which table is relevant to this), and the confidence interval is for present-day February average max daily temperatures. We will compare this confidence interval to the historical average (ie. the `"Past"` value in our `monthly_increases` table). How will we do the comparison? Well, since we are essentially interested in seeing if the average February max daily temperatures have ***changed*** since the past, we care about whether the historical average lies within the confidence interval we create.

Based on the information above, think what the null hypothesis and alternative hypothesis are.

Question 1.2.6. Complete the implementation of the function `make_ci`, which takes a one-column table `tbl` containing sample observations and a confidence `level` percentage such as 95 or 99. **It returns an array containing the lower and upper bound in that order, of a confidence interval** for the population mean constructed using 5,000 bootstrap resamples.

After defining `make_ci`, we have provided a line of code that calls `make_ci` on the present-day February max temperatures to output the 99% confidence interval for the average of daily max temperatures in February. The result should be around 67 degrees for the lower bound and around 71 degrees for the upper bound of the interval.

[73]:
```python
# SCRATCH WORK
feb_present.show(5)
feb_present.select('tmax').show(5)
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[74]:
```python
def make_ci(tbl, level): # 1 column table of ints, confidence level percentage
    """Compute a level% confidence interval of the average of the population␣
 ↪for
    which column 0 of Table tbl contains a sample. This function should return␣
 ↪an
    array of the lower and upper bounds of the confidence interval.
    """
    stats = make_array()
    for k in np.arange(5000):
        stat = tbl.sample().column(0)
        stats = np.append(stats, np.mean(stat))

    # ((100 - level) / 2) => 0.5
    lower_bound = percentile( (100 - level) / 2, stats)

    # level + ((100 - level) / 2) => 99.5
    upper_bound = percentile(level + ((100 - level) / 2), stats)

    # returns an array of lower and upper bound of a CI
    return make_array(lower_bound, upper_bound) # 67.3, 72

# Call make_ci on the max temperatures in present-day February to find the␣
 ↪lower and upper bound of a 99% confidence interval.
feb_present_ci = make_ci(feb_present.select('tmax'), 99)
feb_present_ci
```

[74]:
```
array([ 67.07882353,  71.36588235])
```

[75]:
```python
grader.check("q1_2_6")
```

[75]:
```
q1_2_6 results: All test cases passed!
```

Question 1.2.7 The `feb_present_ci` 99% confidence interval contains the observed past February average maximum temperature of 68.8485 (from the `monthly_increases` table). What conclusion can you draw about the effect of climate change on February maximum temperatures in Phoenix from this information? Use a 1% p-value cutoff.

Note: If you're stuck on this question, re-reading the paragraphs under the *February* heading (particularly the first few) may be helpful.

Given that the confidence interval contains the past average maximum temperature for the month of February, we can conclude the average maximum temperature in February has not been affected by climate change yet.

1.1.4 All Months

Question 1.2.8. Repeat the process of seeing whether the **past average for each month** is **contained within the 99% confidence interval** of the **present average**. Just as a note for the context, remember that these "averages" are averages of the max daily temperatures within those time periods. Code has already been written to print out the results in the format of a month (e.g., `02 (Feb)`), the observed past average, the confidence interval for the present average, and whether the past average was contained in the interval or not.

Use the provided call to `print` in order to format the result as one line per month.

Hint: Your code should follow the same format as our code from above (i.e. the *February* section).

[76]:
```python
# SCRATCH
monthly_increases.show(4)
phoenix.show(4)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[77]:
```python
comparisons = make_array()

months = monthly_increases.column("Month")

for month in months:

    past_average = monthly_increases.where("Month", month).column("Past").
 ↪item(0)

    phoenix_present_table = (phoenix_copy_table.
        where("Year", are.between_or_equal_to(2019, 2021)).
        select("Period", "Month", "tmax"))

    present_observations = phoenix_present_table.where("Month", month).
 ↪select("tmax")

    present_ci = make_ci(present_observations, 99) # make_ci(tbl, level), where␣
 ↪level = 99


    # Do not change the code below this line
    GREEN_BOLD = '\033[1;32m'
    RED_BOLD = '\033[1;31m'
```

```
    RESET_STYLE = '\033[0m'
    within = (past_average >= present_ci.item(0)) and (past_average <=␣
↪present_ci.item(1))
    if within:
        comparison = f'{GREEN_BOLD}contained{RESET_STYLE}'
    else:
        comparison = f'{RED_BOLD}NOT contained{RESET_STYLE}'
    comparisons = np.append(comparisons, comparison)

    print('For', month, 'the past avg', round(past_average, 1),
    'is', comparison,
    'in the interval', np.round(present_ci, 1),
    ', the 99% CI of the present avg. \n')
```

```
For 01 (Jan) the past avg 65.0 is NOT contained in the interval [
66.3  69.2] , the 99% CI of the present avg.

For 02 (Feb) the past avg 68.8 is contained in the interval [ 66.9
71.3] , the 99% CI of the present avg.

For 03 (Mar) the past avg 74.6 is contained in the interval [ 74.
77.9] , the 99% CI of the present avg.

For 04 (Apr) the past avg 82.6 is NOT contained in the interval [
86.4  90.4] , the 99% CI of the present avg.

For 05 (May) the past avg 91.4 is NOT contained in the interval [
92.5  96.6] , the 99% CI of the present avg.

For 06 (Jun) the past avg 101.2 is NOT contained in the interval [
104.3  107.2] , the 99% CI of the present avg.

For 07 (Jul) the past avg 103.6 is NOT contained in the interval [
105.5  108.8] , the 99% CI of the present avg.

For 08 (Aug) the past avg 101.4 is NOT contained in the interval [
105.8  108.9] , the 99% CI of the present avg.

For 09 (Sep) the past avg 97.7 is NOT contained in the interval [
99.3  103.1] , the 99% CI of the present avg.

For 10 (Oct) the past avg 86.8 is NOT contained in the interval [
87.8  92.3] , the 99% CI of the present avg.

For 11 (Nov) the past avg 74.6 is NOT contained in the interval [ 78.
82.8] , the 99% CI of the present avg.
```

```
For 12 (Dec) the past avg 65.9 is contained in the interval [ 65.7
69.2] , the 99% CI of the present avg.
```

[78]:
```python
grader.check("q1_2_8")
```

[78]:
```
q1_2_8 results: All test cases passed!
```

Question 1.2.9. Summarize your findings. After checking whether the past average (of max temperatures, as defined above 1.2.6) is contained in the 99% confidence interval for each month, what conclusions can we make about the monthly average maximum temperature in historical (1900-1960) vs. modern (2019-2021) times in the twelve months? In other words, what null hypothesis should you consider, and for which months would you reject or fail to reject the null hypothesis? Use a 1% p-value cutoff.

Hint: Do you notice any seasonal patterns?

Considering a null hypothesis like the monthly average maximum temperatures will remain the same between past and present historical periods, with any variation due to chance. We can reject this null hypothesis for every month of the year, except for the months December, February, and March; because the past average is contained in the confidence interval of the present period. There seems to be a trend in which the temperatures of the winter months of the present period are within error of the winter months of the past period. Although January bucks this trend by not falling within its present confidence interval. However with a 1% p-value this may be an outlier case causes by compounding error due to our 5000 samples.

> It is important to be wary of making conclusions based on the results of multiple hypothesis tests. Specifically, recall that a 1% p-value cutoff means that if the null is true, we would expect to incorrectly reject the null 1% of the time. Now, if we run multiple of these tests, the chance we observe an error will be greater than if we had just run one test – the error chance compounds.

1.1.5 Checkpoint (due Friday, 11/08 by 5:00pm PT)

Congrats on reaching the checkpoint! **Teddy** is proud of you! Bark Bark Bark!

Run the following cells and submit to the Project 2 Checkpoint Gradescope assignment.

To earn full credit for this checkpoint, you must pass all the public autograder tests above this cell. The cell below will re-run all of the autograder tests for Part 1 to double check your work.

[79]:
```python
checkpoint_tests = ["q1_1_3", "q1_1_4", "q1_1_5",
                    "q1_2_1", "q1_2_2", "q1_2_5", "q1_2_6", "q1_2_8"]

for test in checkpoint_tests:
    display(grader.check(test))
```

```
q1_1_3 results: All test cases passed!
```

```
q1_1_4 results: All test cases passed!

q1_1_5 results: All test cases passed!

q1_2_1 results: All test cases passed!

q1_2_2 results: All test cases passed!

q1_2_5 results: All test cases passed!

q1_2_6 results: All test cases passed!

q1_2_8 results: All test cases passed!
```

1.2 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

Reminders:

- If you worked on Project 2 with a partner, please remember to add your partner to your Gradescope submission. If you resubmit, make sure to re-add your partner, as Gradescope does not save any partner information.
- Make sure to wait until the autograder finishes running to ensure that your submission was processed properly and that you submitted to the correct assignment.

[80]:
```python
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False)
```

```
<IPython.core.display.HTML object>
```

[81]:
```python
# Run this cell to set up the notebook, but please don't change it.
from datascience import *
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
np.set_printoptions(legacy='1.13')

import warnings
warnings.simplefilter('ignore')
```

2 Part 2: Drought

According to the United States Environmental Protection Agency, “Large portions of the Southwest have experienced drought conditions since weekly Drought Monitor records began in 2000. For extended periods from 2002 to 2005 and from 2012 to 2020, nearly the entire region was abnormally dry or even drier.”

Assessing the impact of drought is challenging with just city-level data because so much of the water that people use is transported from elsewhere, but we'll explore the data we have and see what we can learn.

Let's first take a look at the precipitation data in the Southwest region. The `southwest.csv` file contains total annual precipitation for 13 cities in the southwestern United States for each year from 1960 to 2021. This dataset is aggregated from the daily data and includes only the Southwest cities from the original dataset that have consistent precipitation records back to 1960.

[82]:
```
southwest = Table.read_table('southwest.csv')
southwest.show(5)
```

```
<IPython.core.display.HTML object>
```

Question 2.1. Create a table `totals` that has one row for each year in chronological order. It should contain the following columns: 1. `"Year"`: The year (a number) 2. `"Precipitation"`: The total precipitation in all 13 southwestern cities that year

[83]:
```
totals = southwest.select("Year", "Total Precipitation").group("Year", sum).
↪relabeled("Total Precipitation sum", "Precipitation")
totals
```

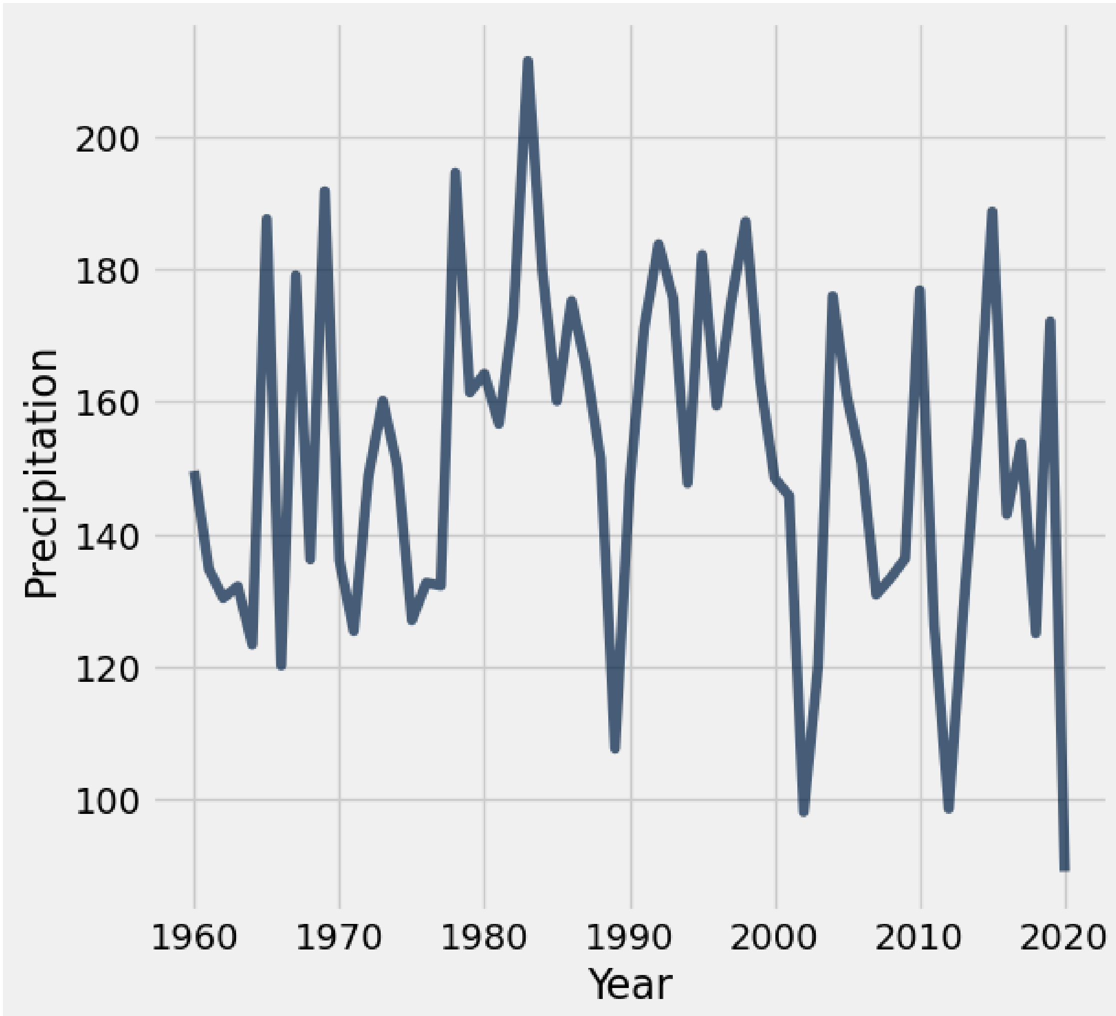
[83]:
```
Year | Precipitation
1960 | 149.58
1961 | 134.82
1962 | 130.41
1963 | 132.18
1964 | 123.41
1965 | 187.53
1966 | 120.27
1967 | 179.02
1968 | 136.25
1969 | 191.72
… (51 rows omitted)
```

[84]:
```
grader.check("q2_1")
```

[84]:
```
q2_1 results: All test cases passed!
```

Run the cell below to plot the total precipitation in these cities over time, so that we can try to spot the drought visually. As a reminder, the drought years given by the EPA were (2002-2005) and (2012-2020).

[85]:
```
# Just run this cell
totals.plot("Year", "Precipitation")
```



This plot isn't very revealing. Each year has a different amount of precipitation, and there is quite a bit of variability across years, as if each year's precipitation is a random draw from a distribution of possible outcomes.

Could it be that these so-called "drought conditions" from 2002-2005 and 2012-2020 can be explained by chance? In other words, could it be that the annual precipitation amounts in the Southwest for these drought years are like **random draws from the same underlying distribution** as for other years? Perhaps nothing about the Earth's precipitation patterns has really changed, and the Southwest U.S. just happened to experience a few dry years close together.

To assess this idea, let's conduct an A/B test in which **each year's total precipitation** is an outcome, and the condition is **whether or not the year is in the EPA's drought period**.

This `drought_label` function distinguishes between drought years as described in the U.S. EPA statement above (2002-2005 and 2012-2020) and other years. Note that the label "other" is perhaps misleading, since there were other droughts before 2000, such as the massive 1988 drought that affected much of the U.S. However, if we're interested in whether these modern drought periods (2002-2005 and 2012-2020) are *normal* or *abnormal*, it makes sense to distinguish the years in this way.

[86]:
```python
def drought_label(n):
    """Return the label for an input year n."""
    if 2002 <= n <= 2005 or 2012 <= n <= 2020:
        return 'drought'
    else:
        return 'other'
```

Question 2.2. Define null and alternative hypotheses for an A/B test that investigates whether drought years are **drier** (have less precipitation) than other years.

Note: Please format your answer using the following structure.

- *Null hypothesis:* …
- *Alternative hypothesis:* …

- *Null hypothesis:* In the population, the distribution of each year's total precipitation is the same whether or not the year is in the EPA's drought period, like random draws from the same underlying distribution. The difference in the sample is due to random chance.
- *Alternative hypothesis:* In the population, the mean total precipitation of drought years is lower, on average, than non-drought years, and this is not due to random chance.

Question 2.3. First, define the table `drought`. It should contain one row per year and the following two columns: - `"Label"`: Denotes if a year is part of a `"drought"` year or an `"other"` year - `"Precipitation"`: The sum of the total precipitation in 13 Southwest cities that year

Then, construct an overlaid histogram of two observed distributions: the total precipitation in drought years and the total precipitation in other years.

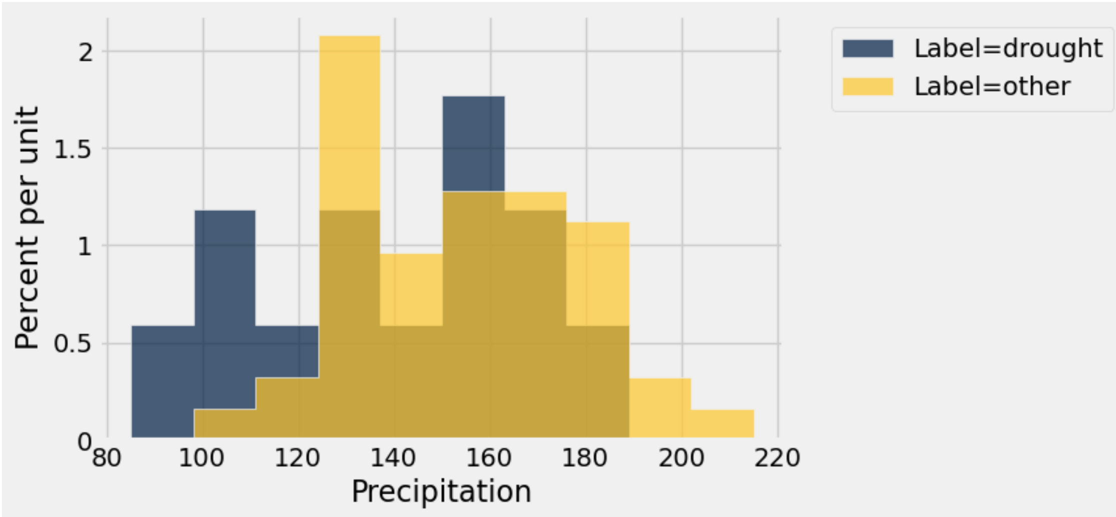
Note: Use the provided `bins` when creating your histogram, and do not re-assign the `southwest` table. Feel free to use as many lines as you need!

Hint: The optional `group` argument in a certain function might be helpful!

[88]:
```python
bins = np.arange(85, 215+1, 13)

drought_years_labeled_array = totals.apply(drought_label, "Year")

drought = totals.with_column("Label", drought_years_labeled_array) # table
drought.hist("Precipitation", bins=bins, group="Label")
```



Before you continue, inspect the histogram you just created and try to guess the conclusion of the A/B test. Building intuition about the result of hypothesis testing from visualizations is quite useful for data science applications.

Question 2.4. Our next step is to choose a test statistic based on our alternative hypothesis in Question 2.2. Which of the following options are valid choices for the test statistic? Assign `ab_test_stat` to **an array of integers** corresponding to valid choices. Assume averages and totals are taken over the total precipitation sums for each year.

1. The difference between the **total** precipitation in **drought** years and the **total** precipitation in **other** years.
2. The difference between the **total** precipitation in **others** years and the **total** precipitation in **drought** years.
3. The **absolute** difference between the **total** precipitation in others years and the **total** precipitation in drought years.
4. The difference between the **average** precipitation in **drought** years and the **average** precipitation in **other** years.
5. The difference between the **average** precipitation in **others** years and the **average** precipitation in **drought** years.
6. The **absolute** difference between the **average** precipitation in others years and the **average** precipitation in drought years.

[89]:
```
ab_test_stat = make_array(4, 5, 6)
```

[90]:
```
grader.check("q2_4")
```

[90]:
```
q2_4 results: All test cases passed!
```

Question 2.5. Fellow climate scientists Noah and Sarah point out that there are more **other** years than **drought** years, and so measuring the difference between total precipitation will always favor the **other** years. They conclude that all of the options above involving **total** precipitation are invalid test statistic choices. Do you agree with them? Why or why not?

> *Hint:* Think about how permutation tests work with imbalanced classes!

We agree, because total precipitation is skewed by imbalances in sample size. A test statistic based on total precipitation would be testing for the total amount of precipitation accumulated across all years in each class, however we are interested in differences in precipitation on a per year basis. Using mean precipitation per class as a test statistic allows us to analyze what we are interested in.

Before going on, check your `drought` table. It should have two columns `Label` and `Precipitation` with 61 rows, 13 of which are for `"drought"` years.

[91]:
```
drought.show(3)
```

```
<IPython.core.display.HTML object>
```

[92]:
```
drought.group('Label')
```

[92]:
```
Label   | count
drought | 13
other   | 48
```

Question 2.6. For our A/B test, we'll use the difference between the average precipitation in drought years and the average precipitation in other years as our test statistic:

$$\text{average precipitation in "drought" years} - \text{average precipitation in "other" years}$$

First, complete the function `test_statistic`. It should take in a two-column table `t` with one row per year and two columns: - `Label`: the label for that year (either `'drought'` or `'other'`) - `Precipitation`: the total precipitation in the 13 Southwest cities that year.

Then, use the function you define to assign `observed_statistic` to the observed test statistic.

[122]:
```
def test_statistic(t): # Label, Precipitation
    drought_avg_precip = np.mean(t.where("Label", "drought").
 ↪column("Precipitation"))
    other_avg_precip = np.mean(t.where("Label", "other").
 ↪column("Precipitation"))

    return drought_avg_precip - other_avg_precip


observed_statistic = test_statistic(drought)
observed_statistic
```

[122]: -15.856714743589748

[94]:
```
grader.check("q2_6")
```

[94]: q2_6 results: All test cases passed!

Now that we have defined our hypotheses and test statistic, we are ready to conduct our hypothesis test. We'll start by defining a function to simulate the test statistic under the null hypothesis, and then call that function 5,000 times to construct an empirical distribution under the null hypothesis.

Question 2.7. Write a function to simulate the test statistic under the null hypothesis. The `simulate_precipitation_null` function should simulate the null hypothesis once (not 5,000 times) and return the value of the test statistic for that simulated sample.

Hint: For example, using `t.with_column(...)` with a column name that already exists in an arbitrary table `t` will replace that column with the newly specified values.

[159]:
```
def simulate_precipitation_null():
    shuffled_labels = drought.sample(with_replacement = False).column("Label")
    shuffled_table = drought.select("Precipitation").with_column("Shuffled␣
↪Label", shuffled_labels)

    means_table = shuffled_table.group("Shuffled Label", np.mean)

    precip_means = means_table.column(1)

    drought_mean = precip_means.item(0)
    other_mean = precip_means.item(1)

    return drought_mean - other_mean

# Run your function a couple times to make sure that it works
simulate_precipitation_null()
```

[159]: -4.442676282051252

[160]:
```
grader.check("q2_7")
```

[160]: q2_7 results: All test cases passed!

Question 2.8. Fill in the blanks below to complete the simulation for the hypothesis test. Your simulation should compute 5,000 values of the test statistic under the null hypothesis and store the result in the array `sampled_stats`.

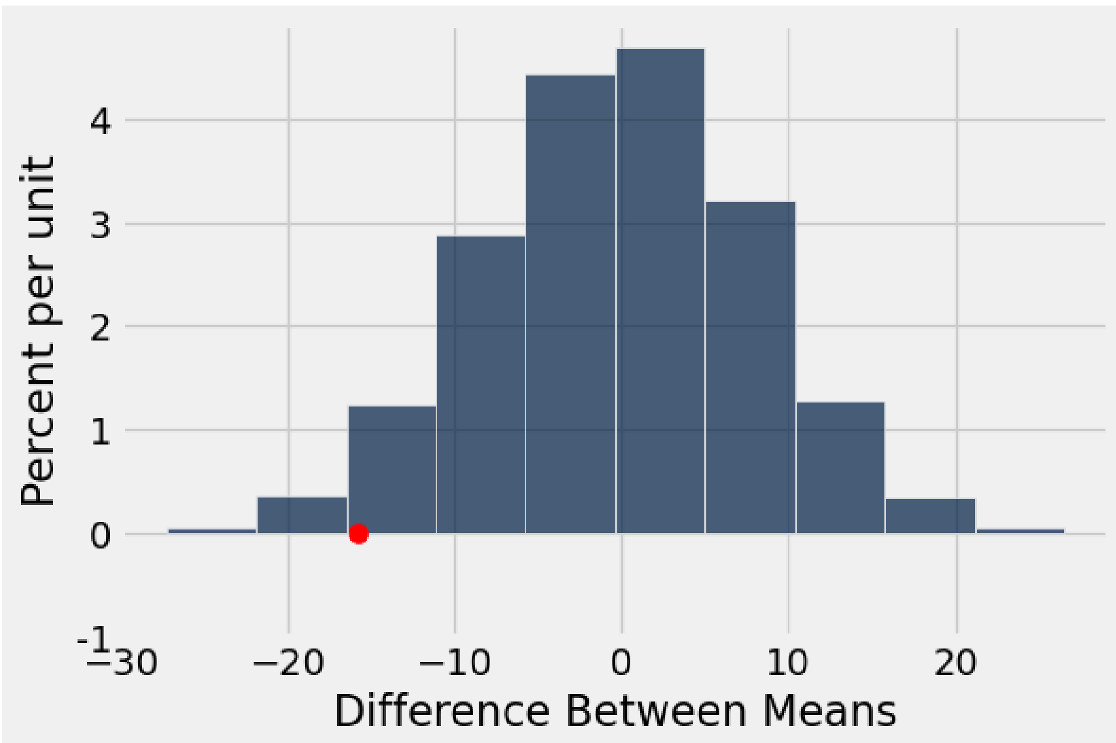
Hint: You should use the `simulate_precipitation_null` function you wrote in the previous question!

Note: Running this cell may take a few seconds. If it takes more than a minute, try to find a different (faster) way to implement your `simulate_precipitation_null` function.

[161]:
```
sampled_stats = make_array()

repetitions = 5_000
for i in np.arange(repetitions):
    new_stat = simulate_precipitation_null()
    sampled_stats = np.append(sampled_stats, new_stat)

# Do not change these lines
Table().with_column('Difference Between Means', sampled_stats).hist()
plt.scatter(observed_statistic, 0, c="r", s=50);
plt.ylim(-0.01);
```



[162]:
```
grader.check("q2_8")
```

[162]: q2_8 results: All test cases passed!

Question 2.9. Compute the p-value for this hypothesis test, and assign it to the variable `precipitation_p_val`.

[166]:
```
precipitation_p_val = np.count_nonzero(sampled_stats <= observed_statistic) /
    ↪repetitions

precipitation_p_val
```

[166]: 0.0278

[167]:
```
grader.check("q2_9")
```

[167]: q2_9 results: All test cases passed!

Question 2.10. State a conclusion from this test using a p-value cutoff of 5%. What have you learned about the EPA's statement on drought?

Using a p-value cutoff of five percent, we can reject the null hypothesis, since our empirical p-value is less than five percent. With this, we have learned that the drought conditions are not due to random chance, these conditions are abnormal, and we support the EPA's statement on drought conditions for the 13 cities we have analyzed.

Question 2.11. Does your conclusion from Question 2.10 apply to the entire Southwest region of the U.S.? Why or why not?

> *Note:* Feel free to do some research into geographical features of this region of the U.S.!

No. Our conclusion in Question 2.10 is based on an A/B test of a sample of 13 southwestern US cities (not evenly distributed across the southwest region). The conclusion of our A/B test is appropriate for comparing the distribution of samples, however our conclusion cannot be stretched to evaluate other cities/regions not tested here. Doing so is out of scope for an A/B test, and bootstrap testing would be more applicable to evaluate such questions.

3 Conclusion

Data science plays a central role in climate change research because massive simulations of the Earth's climate are necessary to assess the implications of climate data recorded from weather stations, satellites, and other sensors. Berkeley Earth is a common source of data for these kinds of projects.

In this project, we found ways to apply our statistical inference technqiues that rely on random sampling even in situations where the data were not generated randomly, but instead by some complicated natural process that appeared random. We made assumptions about randomness and then came to conclusions based on those assumptions. Great care must be taken to choose assumptions that are realistic, so that the resulting conclusions are not misleading. However, making assumptions about data can be productive when doing so allows inference techniques to apply to novel situations.

Congratulations – Chipper is proud of you for finishing Project 2! Maybe you can take a nice nap later! ***Important Reminders:***

- If you worked on Project 2 with a partner, please remember to add your partner to your Gradescope submission. If you resubmit, make sure to re-add your partner, as Gradescope does not save any partner information.
- Make sure to wait until the autograder finishes running to ensure that your submission was processed properly and that you submitted to the correct assignment.

3.1 Written Work Submission

Below, you will see two cells. Running the first cell will automatically generate a PDF of all questions that need to be manually graded, and running the second cell will automatically generate a zip with your autograded answers. You are responsible for submitting both the coding portion (the zip) and the written portion (the PDF) to their respective Gradescope portals. **Please save before exporting!**

> **Important: You must correctly assign the pages of your PDF after you submit to the correct gradescope assignment. If your pages are not correctly assigned and/or not in the correct PDF format by the deadline, we reserve the right to award no points for your written work.**

If there are issues with automatically generating the PDF in the first cell, you can try downloading the notebook as a PDF by clicking on `File -> Save and Export Notebook As... -> Webpdf`. If that doesn't work either, you can manually take screenshots of your answers to the manually graded questions and submit one single PDF of your screenshots. Either way, **you are responsible for ensuring your submision follows our requirements, we will NOT be granting regrade requests for submissions that don't follow instructions.**

You must submit the PDF generated via one of these methods, we will not accept screenshots or Word documents.

[174]:
```
from otter.export import export_notebook
from os import path
from IPython.display import display, HTML
name = 'project2'
export_notebook(f"{name}.ipynb", filtering=True, pagebreaks=True)
if(path.exists(f'{name}.pdf')):
    display(HTML(f"Download your PDF <a href='{name}.pdf' download>here</a>."))
else:
    print("\n Pdf generation failed, please try the other methods described␣
 ↪above")
```

```
<IPython.core.display.HTML object>
```

3.2 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[175]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False, run_tests=True)
```

```
Running your submission against local test cases…



Your submission received the following results when run against available test
cases:

    q1_1_3 results: All test cases passed!

    q1_1_4 results: All test cases passed!

    q1_1_5 results: All test cases passed!

    q1_2_1 results: All test cases passed!

    q1_2_2 results: All test cases passed!

    q1_2_5 results: All test cases passed!

    q1_2_6 results: All test cases passed!

    q1_2_8 results: All test cases passed!

    q2_1 results: All test cases passed!

    q2_4 results: All test cases passed!

    q2_6 results: All test cases passed!

    q2_7 results: All test cases passed!

    q2_8 results: All test cases passed!

    q2_9 results: All test cases passed!
<IPython.core.display.HTML object>
```