**2 Data wrangling**

**2.1 Contents**

* [2 Data wrangling](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2_Data_wrangling)
  + [2.1 Contents](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.1_Contents)
  + [2.2 Introduction](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.2_Introduction)
    - [2.2.1 Recap Of Data Science Problem](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.2.1_Recap_Of_Data_Science_Problem)
    - [2.2.2 Introduction To Notebook](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.2.2_Introduction_To_Notebook)
  + [2.3 Imports](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.3_Imports)
  + [2.4 Objectives](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.4_Objectives)
  + [2.5 Load The Ski Resort Data](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.5_Load_The_Ski_Resort_Data)
  + [2.6 Explore The Data](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6_Explore_The_Data)
    - [2.6.1 Find Your Resort Of Interest](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.1_Find_Your_Resort_Of_Interest)
    - [2.6.2 Number Of Missing Values By Column](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.2_Number_Of_Missing_Values_By_Column)
    - [2.6.3 Categorical Features](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3_Categorical_Features)
      * [2.6.3.1 Unique Resort Names](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.1_Unique_Resort_Names)
      * [2.6.3.2 Region And State](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.2_Region_And_State)
      * [2.6.3.3 Number of distinct regions and states](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.3_Number_of_distinct_regions_and_states)
      * [2.6.3.4 Distribution Of Resorts By Region And State](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.4_Distribution_Of_Resorts_By_Region_And_State)
      * [2.6.3.5 Distribution Of Ticket Price By State](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.5_Distribution_Of_Ticket_Price_By_State)
        + [2.6.3.5.1 Average weekend and weekday price by state](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.5.1_Average_weekend_and_weekday_price_by_state)
        + [2.6.3.5.2 Distribution of weekday and weekend price by state](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.3.5.2_Distribution_of_weekday_and_weekend_price_by_state)
    - [2.6.4 Numeric Features](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4_Numeric_Features)
      * [2.6.4.1 Numeric data summary](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.1_Numeric_data_summary)
      * [2.6.4.2 Distributions Of Feature Values](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.2_Distributions_Of_Feature_Values)
        + [2.6.4.2.1 SkiableTerrain\_ac](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.2.1_SkiableTerrain_ac)
        + [2.6.4.2.2 Snow Making\_ac](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.2.2_Snow_Making_ac)
        + [2.6.4.2.3 fastEight](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.2.3_fastEight)
        + [2.6.4.2.4 fastSixes and Trams](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.6.4.2.4_fastSixes_and_Trams)
  + [2.7 Derive State-wide Summary Statistics For Our Market Segment](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.7_Derive_State-wide_Summary_Statistics_For_Our_Market_Segment)
  + [2.8 Drop Rows With No Price Data](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.8_Drop_Rows_With_No_Price_Data)
  + [2.9 Review distributions](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.9_Review_distributions)
  + [2.10 Population data](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.10_Population_data)
  + [2.11 Target Feature](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.11_Target_Feature)
    - [2.11.1 Number Of Missing Values By Row - Resort](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.11.1_Number_Of_Missing_Values_By_Row_-_Resort)
  + [2.12 Save data](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.12_Save_data)
  + [2.13 Summary](https://github.com/charriemascio/Springboard2/blob/752307a894bc05dc095427868c7c103ea4ce6153/Notebooks/#2.13_Summary)

**2.2 Introduction**

This step focuses on collecting your data, organizing it, and making sure it's well defined. Paying attention to these tasks will pay off greatly later on. Some data cleaning can be done at this stage, but it's important not to be overzealous in your cleaning before you've explored the data to better understand it.

**2.2.1 Recap Of Data Science Problem**

The purpose of this data science project is to come up with a pricing model for ski resort tickets in our market segment. Big Mountain suspects it may not be maximizing its returns, relative to its position in the market. It also does not have a strong sense of what facilities matter most to visitors, particularly which ones they're most likely to pay more for. This project aims to build a predictive model for ticket price based on a number of facilities, or properties, boasted by resorts (*at the resorts).* This model will be used to provide guidance for Big Mountain's pricing and future facility investment plans.

**2.2.2 Introduction To Notebook**

Notebooks grow organically as we explore our data. If you used paper notebooks, you could discover a mistake and cross out or revise some earlier work. Later work may give you a reason to revisit earlier work and explore it further. The great thing about Jupyter notebooks is that you can edit, add, and move cells around without needing to cross out figures or scrawl in the margin. However, this means you can lose track of your changes easily. If you worked in a regulated environment, the company may have a a policy of always dating entries and clearly crossing out any mistakes, with your initials and the date.

**Best practice here is to commit your changes using a version control system such as Git.** Try to get into the habit of adding and committing your files to the Git repository you're working in after you save them. You're are working in a Git repository, right? If you make a significant change, save the notebook and commit it to Git. In fact, if you're about to make a significant change, it's a good idea to commit before as well. Then if the change is a mess, you've got the previous version to go back to.

**Another best practice with notebooks is to try to keep them organized with helpful headings and comments.** Not only can a good structure, but associated headings help you keep track of what you've done and your current focus. Anyone reading your notebook will have a much easier time following the flow of work. Remember, that 'anyone' will most likely be you. Be kind to future you!

In this notebook, note how we try to use well structured, helpful headings that frequently are self-explanatory, and we make a brief note after any results to highlight key takeaways. This is an immense help to anyone reading your notebook and it will greatly help you when you come to summarise your findings. **Top tip: jot down key findings in a final summary at the end of the notebook as they arise. You can tidy this up later.** This is a great way to ensure important results don't get lost in the middle of your notebooks.

In this, and subsequent notebooks, there are coding tasks marked with #Code task n# with code to complete. The \_\_\_ will guide you to where you need to insert code.

**2.3 Imports**

Placing your imports all together at the start of your notebook means you only need to consult one place to check your notebook's dependencies. By all means import something 'in situ' later on when you're experimenting, but if the imported dependency ends up being kept, you should subsequently move the import statement here with the rest.

#Code task 1#

#Import pandas, matplotlib.pyplot, and seaborn in the correct lines below

import \_\_\_ as pd

import \_\_\_ as plt

import \_\_\_ as sns

import os

from library.sb\_utils import save\_file

**2.4 Objectives**

There are some fundamental questions to resolve in this notebook before you move on.

* Do you think you may have the data you need to tackle the desired question?
  + Have you identified the required target value?
  + Do you have potentially useful features?
* Do you have any fundamental issues with the data?

**2.5 Load The Ski Resort Data**

# the supplied CSV data file is the raw\_data directory

ski\_data = pd.read\_csv('../raw\_data/ski\_resort\_data.csv')

Good first steps in auditing the data are the info method and displaying the first few records with head.

#Code task 2#

#Call the info method on ski\_data to see a summary of the data

ski\_data.\_\_\_

AdultWeekday is the price of an adult weekday ticket. AdultWeekend is the price of an adult weekend ticket. The other columns are potential features.

This immediately raises the question of what quantity will you want to model? You know you want to model the ticket price, but you realise there are two kinds of ticket price!

#Code task 3#

#Call the head method on ski\_data to print the first several rows of the data

ski\_data.\_\_\_

The output above suggests you've made a good start getting the ski resort data organized. You have plausible column headings. You can already see you have a missing value in the fastEight column

**2.6 Explore The Data**

**2.6.1 Find Your Resort Of Interest**

Your resort of interest is called Big Mountain Resort. Check it's in the data:

#Code task 4#

#Filter the ski\_data dataframe to display just the row for our resort with the name 'Big Mountain Resort'

#Hint: you will find that the transpose of the row will give a nicer output. DataFrame's do have a

#transpose method, but you can access this conveniently with the `T` property.

ski\_data[ski\_data.Name == \_\_\_].\_\_\_

It's good that your resort doesn't appear to have any missing values.

**2.6.2 Number Of Missing Values By Column**

Count the number of missing values in each column and sort them.

#Code task 5#

#Count (using `.sum()`) the number of missing values (`.isnull()`) in each column of

#ski\_data as well as the percentages (using `.mean()` instead of `.sum()`).

#Order them (increasing or decreasing) using sort\_values

#Call `pd.concat` to present these in a single table (DataFrame) with the helpful column names 'count' and '%'

missing = \_\_\_([ski\_data.\_\_\_.\_\_\_, 100 \* ski\_data.\_\_\_.\_\_\_], axis=1)

missing.columns=[\_\_\_, \_\_\_]

missing.\_\_\_(by=\_\_\_)

fastEight has the most missing values, at just over 50%. Unfortunately, you see you're also missing quite a few of your desired target quantity, the ticket price, which is missing 15-16% of values. AdultWeekday is missing in a few more records than AdultWeekend. What overlap is there in these missing values? This is a question you'll want to investigate. You should also point out that isnull() is not the only indicator of missing data. Sometimes 'missingness' can be encoded, perhaps by a -1 or 999. Such values are typically chosen because they are "obviously" not genuine values. If you were capturing data on people's heights and weights but missing someone's height, you could certainly encode that as a 0 because no one has a height of zero (in any units). Yet such entries would not be revealed by isnull(). Here, you need a data dictionary and/or to spot such values as part of looking for outliers. Someone with a height of zero should definitely show up as an outlier!

**2.6.3 Categorical Features**

So far you've examined only the numeric features. Now you inspect categorical ones such as resort name and state. These are discrete entities. 'Alaska' is a name. Although names can be sorted alphabetically, it makes no sense to take the average of 'Alaska' and 'Arizona'. Similarly, 'Alaska' is before 'Arizona' only lexicographically; it is neither 'less than' nor 'greater than' 'Arizona'. As such, they tend to require different handling than strictly numeric quantities. Note, a feature *can* be numeric but also categorical. For example, instead of giving the number of fastEight lifts, a feature might be has\_fastEights and have the value 0 or 1 to denote absence or presence of such a lift. In such a case it would not make sense to take an average of this or perform other mathematical calculations on it. Although you digress a little to make a point, month numbers are also, strictly speaking, categorical features. Yes, when a month is represented by its number (1 for January, 2 for Februrary etc.) it provides a convenient way to graph trends over a year. And, arguably, there is some logical interpretation of the average of 1 and 3 (January and March) being 2 (February). However, clearly December of one years precedes January of the next and yet 12 as a number is not less than 1. The numeric quantities in the section above are truly numeric; they are the number of feet in the drop, or acres or years open or the amount of snowfall etc.

#Code task 6#

#Use ski\_data's `select\_dtypes` method to select columns of dtype 'object'

ski\_data.\_\_\_(\_\_\_)

You saw earlier on that these three columns had no missing values. But are there any other issues with these columns? Sensible questions to ask here include:

* Is Name (or at least a combination of Name/Region/State) unique?
* Is Region always the same as state?

**2.6.3.1 Unique Resort Names**

#Code task 7#

#Use pandas' Series method `value\_counts` to find any duplicated resort names

ski\_data['Name'].\_\_\_.head()

You have a duplicated resort name: Crystal Mountain.

**Q: 1** Is this resort duplicated if you take into account Region and/or state as well?

#Code task 8#

#Concatenate the string columns 'Name' and 'Region' and count the values again (as above)

(ski\_data[\_\_\_] + ', ' + ski\_data[\_\_\_]).\_\_\_.head()

#Code task 9#

#Concatenate 'Name' and 'state' and count the values again (as above)

(ski\_data[\_\_\_] + ', ' + ski\_data[\_\_\_]).\_\_\_.head()

\*\*NB\*\* because you know `value\_counts()` sorts descending, you can use the `head()` method and know the rest of the counts must be 1.

**A: 1** Your answer here

ski\_data[ski\_data['Name'] == 'Crystal Mountain']

|  | **Name** | **Region** | **state** | **summit\_elev** | **vertical\_drop** | **base\_elev** | **trams** | **fastEight** | **fastSixes** | **fastQuads** | **...** | **LongestRun\_mi** | **SkiableTerrain\_ac** | **Snow Making\_ac** | **daysOpenLastYear** | **yearsOpen** | **averageSnowfall** | **AdultWeekday** | **AdultWeekend** | **projectedDaysOpen** | **NightSkiing\_ac** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **104** | Crystal Mountain | Michigan | Michigan | 1132 | 375 | 757 | 0 | 0.0 | 0 | 1 | ... | 0.3 | 102.0 | 96.0 | 120.0 | 63.0 | 132.0 | 54.0 | 64.0 | 135.0 | 56.0 |
| **295** | Crystal Mountain | Washington | Washington | 7012 | 3100 | 4400 | 1 | NaN | 2 | 2 | ... | 2.5 | 2600.0 | 10.0 | NaN | 57.0 | 486.0 | 99.0 | 99.0 | NaN | NaN |

2 rows × 27 columns

So there are two Crystal Mountain resorts, but they are clearly two different resorts in two different states. This is a powerful signal that you have unique records on each row.

**2.6.3.2 Region And State**

What's the relationship between region and state?

You know they are the same in many cases (e.g. both the Region and the state are given as 'Michigan'). In how many cases do they differ?

#Code task 10#

#Calculate the number of times Region does not equal state

(ski\_data.Region \_\_\_ ski\_data.state).\_\_\_

You know what a state is. What is a region? You can tabulate the distinct values along with their respective frequencies using value\_counts().

ski\_data['Region'].value\_counts()

New York 33

Michigan 29

Sierra Nevada 22

Colorado 22

Pennsylvania 19

Wisconsin 16

New Hampshire 16

Vermont 15

Minnesota 14

Montana 12

Idaho 12

Massachusetts 11

Washington 10

Maine 9

New Mexico 9

Wyoming 8

Utah 7

Oregon 6

Salt Lake City 6

North Carolina 6

Connecticut 5

Ohio 5

West Virginia 4

Virginia 4

Mt. Hood 4

Illinois 4

Alaska 3

Iowa 3

Missouri 2

Arizona 2

Indiana 2

South Dakota 2

New Jersey 2

Nevada 2

Rhode Island 1

Maryland 1

Tennessee 1

Northern California 1

Name: Region, dtype: int64

A casual inspection by eye reveals some non-state names such as Sierra Nevada, Salt Lake City, and Northern California. Tabulate the differences between Region and state. On a note regarding scaling to larger data sets, you might wonder how you could spot such cases when presented with millions of rows. This is an interesting point. Imagine you have access to a database with a Region and state column in a table and there are millions of rows. You wouldn't eyeball all the rows looking for differences! Bear in mind that our first interest lies in establishing the answer to the question "Are they always the same?" One approach might be to ask the database to return records where they differ, but limit the output to 10 rows. If there were differences, you'd only get up to 10 results, and so you wouldn't know whether you'd located all differences, but you'd know that there were 'a nonzero number' of differences. If you got an empty result set back, then you would know that the two columns always had the same value. At the risk of digressing, some values in one column only might be NULL (missing) and different databases treat NULL differently, so be aware that on many an occasion a seamingly 'simple' question gets very interesting to answer very quickly!

#Code task 11#

#Filter the ski\_data dataframe for rows where 'Region' and 'state' are different,

#group that by 'state' and perform `value\_counts` on the 'Region'

(ski\_data[ski\_data.\_\_\_ \_\_\_ ski\_data.\_\_\_]

.groupby(\_\_\_)[\_\_\_]

.value\_counts())

The vast majority of the differences are in California, with most Regions being called Sierra Nevada and just one referred to as Northern California.

**2.6.3.3 Number of distinct regions and states**

#Code task 12#

#Select the 'Region' and 'state' columns from ski\_data and use the `nunique` method to calculate

#the number of unique values in each

ski\_data[[\_\_\_, \_\_\_]].\_\_\_

Because a few states are split across multiple named regions, there are slightly more unique regions than states.

**2.6.3.4 Distribution Of Resorts By Region And State**

If this is your first time using [matplotlib](https://matplotlib.org/3.2.2/index.html)'s [subplots](https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.subplots.html), you may find the online documentation useful.

#Code task 13#

#Create two subplots on 1 row and 2 columns with a figsize of (12, 8)

fig, ax = plt.subplots(\_\_\_, \_\_\_, figsize=(\_\_\_))

#Specify a horizontal barplot ('barh') as kind of plot (kind=)

ski\_data.Region.value\_counts().plot(kind=\_\_\_, ax=ax[0])

#Give the plot a helpful title of 'Region'

ax[0].set\_title(\_\_\_)

#Label the xaxis 'Count'

ax[0].set\_xlabel(\_\_\_)

#Specify a horizontal barplot ('barh') as kind of plot (kind=)

ski\_data.state.value\_counts().plot(kind=\_\_\_, ax=ax[1])

#Give the plot a helpful title of 'state'

ax[1].set\_title(\_\_\_)

#Label the xaxis 'Count'

ax[1].set\_xlabel(\_\_\_)

#Give the subplots a little "breathing room" with a wspace of 0.5

plt.subplots\_adjust(wspace=\_\_\_);

#You're encouraged to explore a few different figure sizes, orientations, and spacing here

# as the importance of easy-to-read and informative figures is frequently understated

# and you will find the ability to tweak figures invaluable later on

How's your geography? Looking at the distribution of States, you see New York accounting for the majority of resorts. Our target resort is in Montana, which comes in at 13th place. You should think carefully about how, or whether, you use this information. Does New York command a premium because of its proximity to population? Even if a resort's State were a useful predictor of ticket price, your main interest lies in Montana. Would you want a model that is skewed for accuracy by New York? Should you just filter for Montana and create a Montana-specific model? This would slash your available data volume. Your problem task includes the contextual insight that the data are for resorts all belonging to the same market share. This suggests one might expect prices to be similar amongst them. You can look into this. A boxplot grouped by State is an ideal way to quickly compare prices. Another side note worth bringing up here is that, in reality, the best approach here definitely would include consulting with the client or other domain expert. They might know of good reasons for treating states equivalently or differently. The data scientist is rarely the final arbiter of such a decision. But here, you'll see if we can find any supporting evidence for treating states the same or differently.

**2.6.3.5 Distribution Of Ticket Price By State**

Our primary focus is our Big Mountain resort, in Montana. Does the state give you any clues to help decide what your primary target response feature should be (weekend or weekday ticket prices)?

**2.6.3.5.1 Average weekend and weekday price by state**

#Code task 14#

# Calculate average weekday and weekend price by state and sort by the average of the two

# Hint: use the pattern dataframe.groupby(<grouping variable>)[<list of columns>].mean()

state\_price\_means = ski\_data.\_\_\_(\_\_\_)[[\_\_\_, \_\_\_]].mean()

state\_price\_means.head()

# The next bit simply reorders the index by increasing average of weekday and weekend prices

# Compare the index order you get from

# state\_price\_means.index

# with

# state\_price\_means.mean(axis=1).sort\_values(ascending=False).index

# See how this expression simply sits within the reindex()

(state\_price\_means.reindex(index=state\_price\_means.mean(axis=1)

.sort\_values(ascending=False)

.index)

.plot(kind='barh', figsize=(10, 10), title='Average ticket price by State'))

plt.xlabel('Price ($)');

Chart

Description automatically generated

The figure above represents a dataframe with two columns, one for the average prices of each kind of ticket. This tells you how the average ticket price varies from state to state. But can you get more insight into the difference in the distributions between states?

**2.6.3.5.2 Distribution of weekday and weekend price by state**

Next, you can transform the data into a single column for price with a new categorical column that represents the ticket type.

#Code task 15#

#Use the pd.melt function, pass in the ski\_data columns 'state', 'AdultWeekday', and 'Adultweekend' only,

#specify 'state' for `id\_vars`

#gather the ticket prices from the 'Adultweekday' and 'AdultWeekend' columns using the `value\_vars` argument,

#call the resultant price column 'Price' via the `value\_name` argument,

#name the weekday/weekend indicator column 'Ticket' via the `var\_name` argument

ticket\_prices = pd.melt(ski\_data[[\_\_\_, \_\_\_, \_\_\_]],

id\_vars=\_\_\_,

var\_name=\_\_\_,

value\_vars=[\_\_\_, \_\_\_],

value\_name=\_\_\_)

ticket\_prices.head()

|  | **state** | **Ticket** | **Price** |
| --- | --- | --- | --- |
| **0** | Alaska | AdultWeekday | 65.0 |
| **1** | Alaska | AdultWeekday | 47.0 |
| **2** | Alaska | AdultWeekday | 30.0 |
| **3** | Arizona | AdultWeekday | 89.0 |
| **4** | Arizona | AdultWeekday | 74.0 |

This is now in a format we can pass to [seaborn](https://seaborn.pydata.org/)'s [boxplot](https://seaborn.pydata.org/generated/seaborn.boxplot.html) function to create boxplots of the ticket price distributions for each ticket type for each state.

#Code task 16#

#Create a seaborn boxplot of the ticket price dataframe we created above,

#with 'state' on the x-axis, 'Price' as the y-value, and a hue that indicates 'Ticket'

#This will use boxplot's x, y, hue, and data arguments.

plt.subplots(figsize=(12, 8))

sns.boxplot(x=\_\_\_, y=\_\_\_, hue=\_\_\_, data=ticket\_prices)

plt.xticks(rotation='vertical')

plt.ylabel('Price ($)')

plt.xlabel('State');

Aside from some relatively expensive ticket prices in California, Colorado, and Utah, most prices appear to lie in a broad band from around 25 to over 100 dollars. Some States show more variability than others. Montana and South Dakota, for example, both show fairly small variability as well as matching weekend and weekday ticket prices. Nevada and Utah, on the other hand, show the most range in prices. Some States, notably North Carolina and Virginia, have weekend prices far higher than weekday prices. You could be inspired from this exploration to consider a few potential groupings of resorts, those with low spread, those with lower averages, and those that charge a premium for weekend tickets. However, you're told that you are taking all resorts to be part of the same market share, you could argue against further segment the resorts. Nevertheless, ways to consider using the State information in your modelling include:

* disregard State completely
* retain all State information
* retain State in the form of Montana vs not Montana, as our target resort is in Montana

You've also noted another effect above: some States show a marked difference between weekday and weekend ticket prices. It may make sense to allow a model to take into account not just State but also weekend vs weekday.

Thus we currently have two main questions you want to resolve:

* What do you do about the two types of ticket price?
* What do you do about the state information?

**2.6.4 Numeric Features**

Having decided to reserve judgement on how exactly you utilize the State, turn your attention to cleaning the numeric features.

**2.6.4.1 Numeric data summary**

#Code task 17#

#Call ski\_data's `describe` method for a statistical summary of the numerical columns

#Hint: there are fewer summary stat columns than features, so displaying the transpose

#will be useful again

ski\_data.\_\_\_.\_\_\_

Recall you're missing the ticket prices for some 16% of resorts. This is a fundamental problem that means you simply lack the required data for those resorts and will have to drop those records. But you may have a weekend price and not a weekday price, or vice versa. You want to keep any price you have.

missing\_price = ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum(axis=1)

missing\_price.value\_counts()/len(missing\_price) \* 100

0 82.424242

2 14.242424

1 3.333333

dtype: float64

Just over 82% of resorts have no missing ticket price, 3% are missing one value, and 14% are missing both. You will definitely want to drop the records for which you have no price information, however you will not do so just yet. There may still be useful information about the distributions of other features in that 14% of the data.

**2.6.4.2 Distributions Of Feature Values**

Note that, although we are still in the 'data wrangling and cleaning' phase rather than exploratory data analysis, looking at distributions of features is immensely useful in getting a feel for whether the values look sensible and whether there are any obvious outliers to investigate. Some exploratory data analysis belongs here, and data wrangling will inevitably occur later on. It's more a matter of emphasis. Here, we're interesting in focusing on whether distributions look plausible or wrong. Later on, we're more interested in relationships and patterns.

#Code task 18#

#Call ski\_data's `hist` method to plot histograms of each of the numeric features

#Try passing it an argument figsize=(15,10)

#Try calling plt.subplots\_adjust() with an argument hspace=0.5 to adjust the spacing

#It's important you create legible and easy-to-read plots

ski\_data.\_\_\_(\_\_\_)

#plt.subplots\_adjust(hspace=\_\_\_);

#Hint: notice how the terminating ';' "swallows" some messy output and leads to a tidier notebook

What features do we have possible cause for concern about and why?

* SkiableTerrain\_ac because values are clustered down the low end,
* Snow Making\_ac for the same reason,
* fastEight because all but one value is 0 so it has very little variance, and half the values are missing,
* fastSixes raises an amber flag; it has more variability, but still mostly 0,
* trams also may get an amber flag for the same reason,
* yearsOpen because most values are low but it has a maximum of 2019, which strongly suggests someone recorded calendar year rather than number of years.

**2.6.4.2.1 SkiableTerrain\_ac**

#Code task 19#

#Filter the 'SkiableTerrain\_ac' column to print the values greater than 10000

ski\_data.\_\_\_[ski\_data.\_\_\_ > \_\_\_]

**Q: 2** One resort has an incredibly large skiable terrain area! Which is it?

#Code task 20#

#Now you know there's only one, print the whole row to investigate all values, including seeing the resort name

#Hint: don't forget the transpose will be helpful here

ski\_data[ski\_data.\_\_\_ > \_\_\_].\_\_\_

**A: 2** Your answer here

But what can you do when you have one record that seems highly suspicious?

You can see if your data are correct. Search for "silverton mountain skiable area". If you do this, you get some [useful information](https://www.google.com/search?q=silverton+mountain+skiable+area).

You can spot check data. You see your top and base elevation values agree, but the skiable area is very different. Your suspect value is 26819, but the value you've just looked up is 1819. The last three digits agree. This sort of error could have occured in transmission or some editing or transcription stage. You could plausibly replace the suspect value with the one you've just obtained. Another cautionary note to make here is that although you're doing this in order to progress with your analysis, this is most definitely an issue that should have been raised and fed back to the client or data originator as a query. You should view this "data correction" step as a means to continue (documenting it carefully as you do in this notebook) rather than an ultimate decision as to what is correct.

#Code task 21#

#Use the .loc accessor to print the 'SkiableTerrain\_ac' value only for this resort

ski\_data.\_\_\_[39, 'SkiableTerrain\_ac']

#Code task 22#

#Use the .loc accessor again to modify this value with the correct value of 1819

ski\_data.\_\_\_[39, 'SkiableTerrain\_ac'] = \_\_\_

#Code task 23#

#Use the .loc accessor a final time to verify that the value has been modified

ski\_data.\_\_\_[39, 'SkiableTerrain\_ac']

**NB whilst you may become suspicious about your data quality, and you know you have missing values, you will not here dive down the rabbit hole of checking all values or web scraping to replace missing values.**

What does the distribution of skiable area look like now?

ski\_data.SkiableTerrain\_ac.hist(bins=30)

plt.xlabel('SkiableTerrain\_ac')

plt.ylabel('Count')

plt.title('Distribution of skiable area (acres) after replacing erroneous value');

Histogram

Description automatically generated with low confidence

You now see a rather long tailed distribution. You may wonder about the now most extreme value that is above 8000, but similarly you may also wonder about the value around 7000. If you wanted to spend more time manually checking values you could, but leave this for now. The above distribution is plausible.

**2.6.4.2.2 Snow Making\_ac**

ski\_data['Snow Making\_ac'][ski\_data['Snow Making\_ac'] > 1000]

11 3379.0

18 1500.0

Name: Snow Making\_ac, dtype: float64

ski\_data[ski\_data['Snow Making\_ac'] > 3000].T

|  | **11** |
| --- | --- |
| **Name** | Heavenly Mountain Resort |
| **Region** | Sierra Nevada |
| **state** | California |
| **summit\_elev** | 10067 |
| **vertical\_drop** | 3500 |
| **base\_elev** | 7170 |
| **trams** | 2 |
| **fastEight** | 0 |
| **fastSixes** | 2 |
| **fastQuads** | 7 |
| **quad** | 1 |
| **triple** | 5 |
| **double** | 3 |
| **surface** | 8 |
| **total\_chairs** | 28 |
| **Runs** | 97 |
| **TerrainParks** | 3 |
| **LongestRun\_mi** | 5.5 |
| **SkiableTerrain\_ac** | 4800 |
| **Snow Making\_ac** | 3379 |
| **daysOpenLastYear** | 155 |
| **yearsOpen** | 64 |
| **averageSnowfall** | 360 |
| **AdultWeekday** | NaN |
| **AdultWeekend** | NaN |
| **projectedDaysOpen** | 157 |
| **NightSkiing\_ac** | NaN |

You can adopt a similar approach as for the suspect skiable area value and do some spot checking. To save time, here is a link to the website for [Heavenly Mountain Resort](https://www.skiheavenly.com/the-mountain/about-the-mountain/mountain-info.aspx). From this you can glean that you have values for skiable terrain that agree. Furthermore, you can read that snowmaking covers 60% of the trails.

What, then, is your rough guess for the area covered by snowmaking?

.6 \* 4800

2880.0

This is less than the value of 3379 in your data so you may have a judgement call to make. However, notice something else. You have no ticket pricing information at all for this resort. Any further effort spent worrying about values for this resort will be wasted. You'll simply be dropping the entire row!

**2.6.4.2.3 fastEight**

Look at the different fastEight values more closely:

ski\_data.fastEight.value\_counts()

0.0 163

1.0 1

Name: fastEight, dtype: int64

Drop the fastEight column in its entirety; half the values are missing and all but the others are the value zero. There is essentially no information in this column.

#Code task 24#

#Drop the 'fastEight' column from ski\_data. Use inplace=True

ski\_data.drop(columns=\_\_\_, inplace=\_\_\_)

What about yearsOpen? How many resorts have purportedly been open for more than 100 years?

#Code task 25#

#Filter the 'yearsOpen' column for values greater than 100

ski\_data.\_\_\_[ski\_data.\_\_\_ > \_\_\_]

Okay, one seems to have been open for 104 years. But beyond that, one is down as having been open for 2019 years. This is wrong! What shall you do about this?

What does the distribution of yearsOpen look like if you exclude just the obviously wrong one?

#Code task 26#

#Call the hist method on 'yearsOpen' after filtering for values under 1000

#Pass the argument bins=30 to hist(), but feel free to explore other values

ski\_data.\_\_\_[ski\_data.\_\_\_ < \_\_\_].hist(\_\_\_)

plt.xlabel('Years open')

plt.ylabel('Count')

plt.title('Distribution of years open excluding 2019');

The above distribution of years seems entirely plausible, including the 104 year value. You can certainly state that no resort will have been open for 2019 years! It likely means the resort opened in 2019. It could also mean the resort is due to open in 2019. You don't know when these data were gathered!

Let's review the summary statistics for the years under 1000.

ski\_data.yearsOpen[ski\_data.yearsOpen < 1000].describe()

count 328.000000

mean 57.695122

std 16.841182

min 6.000000

25% 50.000000

50% 58.000000

75% 68.250000

max 104.000000

Name: yearsOpen, dtype: float64

The smallest number of years open otherwise is 6. You can't be sure whether this resort in question has been open zero years or one year and even whether the numbers are projections or actual. In any case, you would be adding a new youngest resort so it feels best to simply drop this row.

ski\_data = ski\_data[ski\_data.yearsOpen < 1000]

**2.6.4.2.4 fastSixes and Trams**

The other features you had mild concern over, you will not investigate further. Perhaps take some care when using these features.

**2.7 Derive State-wide Summary Statistics For Our Market Segment**

You have, by this point removed one row, but it was for a resort that may not have opened yet, or perhaps in its first season. Using your business knowledge, you know that state-wide supply and demand of certain skiing resources may well factor into pricing strategies. Does a resort dominate the available night skiing in a state? Or does it account for a large proportion of the total skiable terrain or days open?

If you want to add any features to your data that captures the state-wide market size, you should do this now, before dropping any more rows. In the next section, you'll drop rows with missing price information. Although you don't know what those resorts charge for their tickets, you do know the resorts exists and have been open for at least six years. Thus, you'll now calculate some state-wide summary statistics for later use.

Many features in your data pertain to chairlifts, that is for getting people around each resort. These aren't relevant, nor are the features relating to altitudes. Features that you may be interested in are:

* TerrainParks
* SkiableTerrain\_ac
* daysOpenLastYear
* NightSkiing\_ac

When you think about it, these are features it makes sense to sum: the total number of terrain parks, the total skiable area, the total number of days open, and the total area available for night skiing. You might consider the total number of ski runs, but understand that the skiable area is more informative than just a number of runs.

A fairly new groupby behaviour is [named aggregation](https://pandas-docs.github.io/pandas-docs-travis/whatsnew/v0.25.0.html). This allows us to clearly perform the aggregations you want whilst also creating informative output column names.

#Code task 27#

#Add named aggregations for the sum of 'daysOpenLastYear', 'TerrainParks', and 'NightSkiing\_ac'

#call them 'state\_total\_days\_open', 'state\_total\_terrain\_parks', and 'state\_total\_nightskiing\_ac',

#respectively

#Finally, add a call to the reset\_index() method (we recommend you experiment with and without this to see

#what it does)

state\_summary = ski\_data.groupby('state').agg(

resorts\_per\_state=pd.NamedAgg(column='Name', aggfunc='size'), #could pick any column here

state\_total\_skiable\_area\_ac=pd.NamedAgg(column='SkiableTerrain\_ac', aggfunc='sum'),

state\_total\_days\_open=pd.NamedAgg(column=\_\_, aggfunc='sum'),

\_\_\_=pd.NamedAgg(column=\_\_\_, aggfunc=\_\_\_),

\_\_\_=pd.NamedAgg(column=\_\_\_, aggfunc=\_\_\_)

).\_\_\_

state\_summary.head()

**2.8 Drop Rows With No Price Data**

You know there are two columns that refer to price: 'AdultWeekend' and 'AdultWeekday'. You can calculate the number of price values missing per row. This will obviously have to be either 0, 1, or 2, where 0 denotes no price values are missing and 2 denotes that both are missing.

missing\_price = ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum(axis=1)

missing\_price.value\_counts()/len(missing\_price) \* 100

0 82.317073

2 14.329268

1 3.353659

dtype: float64

About 14% of the rows have no price data. As the price is your target, these rows are of no use. Time to lose them.

#Code task 28#

#Use `missing\_price` to remove rows from ski\_data where both price values are missing

ski\_data = ski\_data[\_\_\_ != 2]

**2.9 Review distributions**

ski\_data.hist(figsize=(15, 10))

plt.subplots\_adjust(hspace=0.5);

A picture containing crossword puzzle

Description automatically generated

These distributions are much better. There are clearly some skewed distributions, so keep an eye on fastQuads, fastSixes, and perhaps trams. These lack much variance away from 0 and may have a small number of relatively extreme values. Models failing to rate a feature as important when domain knowledge tells you it should be is an issue to look out for, as is a model being overly influenced by some extreme values. If you build a good machine learning pipeline, hopefully it will be robust to such issues, but you may also wish to consider nonlinear transformations of features.

**2.10 Population data**

Population and area data for the US states can be obtained from [wikipedia](https://simple.wikipedia.org/wiki/List_of_U.S._states). Listen, you should have a healthy concern about using data you "found on the Internet". Make sure it comes from a reputable source. This table of data is useful because it allows you to easily pull and incorporate an external data set. It also allows you to proceed with an analysis that includes state sizes and populations for your 'first cut' model. Be explicit about your source (we documented it here in this workflow) and ensure it is open to inspection. All steps are subject to review, and it may be that a client has a specific source of data they trust that you should use to rerun the analysis.

#Code task 29#

#Use pandas' `read\_html` method to read the table from the URL below

states\_url = 'https://simple.wikipedia.org/w/index.php?title=List\_of\_U.S.\_states&oldid=7168473'

usa\_states = pd.\_\_\_(\_\_\_)

type(usa\_states)

list

len(usa\_states)

1

usa\_states = usa\_states[0]

usa\_states.head()

|  | **Name &postal abbs. [1]** | | **Cities** | | **Established[upper-alpha 1]** | **Population[upper-alpha 2][3]** | **Total area[4]** | | **Land area[4]** | | **Water area[4]** | | **Numberof Reps.** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Name &postal abbs. [1]** | **Name &postal abbs. [1].1** | **Capital** | **Largest[5]** | **Established[upper-alpha 1]** | **Population[upper-alpha 2][3]** | **mi2** | **km2** | **mi2** | **km2** | **mi2** | **km2** | **Numberof Reps.** |
| **0** | Alabama | AL | Montgomery | Birmingham | Dec 14, 1819 | 4903185 | 52420 | 135767 | 50645 | 131171 | 1775 | 4597 | 7 |
| **1** | Alaska | AK | Juneau | Anchorage | Jan 3, 1959 | 731545 | 665384 | 1723337 | 570641 | 1477953 | 94743 | 245384 | 1 |
| **2** | Arizona | AZ | Phoenix | Phoenix | Feb 14, 1912 | 7278717 | 113990 | 295234 | 113594 | 294207 | 396 | 1026 | 9 |
| **3** | Arkansas | AR | Little Rock | Little Rock | Jun 15, 1836 | 3017804 | 53179 | 137732 | 52035 | 134771 | 1143 | 2961 | 4 |
| **4** | California | CA | Sacramento | Los Angeles | Sep 9, 1850 | 39512223 | 163695 | 423967 | 155779 | 403466 | 7916 | 20501 | 53 |

Note, in even the last year, the capability of pd.read\_html() has improved. The merged cells you see in the web table are now handled much more conveniently, with 'Phoenix' now being duplicated so the subsequent columns remain aligned. But check this anyway. If you extract the established date column, you should just get dates. Recall previously you used the .loc accessor, because you were using labels. Now you want to refer to a column by its index position and so use .iloc. For a discussion on the difference use cases of .loc and .iloc refer to the [pandas documentation](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html).

#Code task 30#

#Use the iloc accessor to get the pandas Series for column number 4 from `usa\_states`

#It should be a column of dates

established = usa\_sates.\_\_\_[:, 4]

established

0 Dec 14, 1819

1 Jan 3, 1959

2 Feb 14, 1912

3 Jun 15, 1836

4 Sep 9, 1850

5 Aug 1, 1876

6 Jan 9, 1788

7 Dec 7, 1787

8 Mar 3, 1845

9 Jan 2, 1788

10 Aug 21, 1959

11 Jul 3, 1890

12 Dec 3, 1818

13 Dec 11, 1816

14 Dec 28, 1846

15 Jan 29, 1861

16 Jun 1, 1792

17 Apr 30, 1812

18 Mar 15, 1820

19 Apr 28, 1788

20 Feb 6, 1788

21 Jan 26, 1837

22 May 11, 1858

23 Dec 10, 1817

24 Aug 10, 1821

25 Nov 8, 1889

26 Mar 1, 1867

27 Oct 31, 1864

28 Jun 21, 1788

29 Dec 18, 1787

30 Jan 6, 1912

31 Jul 26, 1788

32 Nov 21, 1789

33 Nov 2, 1889

34 Mar 1, 1803

35 Nov 16, 1907

36 Feb 14, 1859

37 Dec 12, 1787

38 May 29, 1790

39 May 23, 1788

40 Nov 2, 1889

41 Jun 1, 1796

42 Dec 29, 1845

43 Jan 4, 1896

44 Mar 4, 1791

45 Jun 25, 1788

46 Nov 11, 1889

47 Jun 20, 1863

48 May 29, 1848

49 Jul 10, 1890

Name: (Established[upper-alpha 1], Established[upper-alpha 1]), dtype: object

Extract the state name, population, and total area (square miles) columns.

#Code task 31#

#Now use the iloc accessor again to extract columns 0, 5, and 6 and the dataframe's `copy()` method

#Set the names of these extracted columns to 'state', 'state\_population', and 'state\_area\_sq\_miles',

#respectively.

usa\_states\_sub = usa\_states.\_\_\_[:, [\_\_\_]].copy()

usa\_states\_sub.columns = [\_\_\_]

usa\_states\_sub.head()

Do you have all the ski data states accounted for?

#Code task 32#

#Find the states in `state\_summary` that are not in `usa\_states\_sub`

#Hint: set(list1) - set(list2) is an easy way to get items in list1 that are not in list2

missing\_states = \_\_\_(state\_summary.state) - \_\_\_(usa\_states\_sub.state)

missing\_states

No??

If you look at the table on the web, you can perhaps start to guess what the problem is. You can confirm your suspicion by pulling out state names that *contain* 'Massachusetts', 'Pennsylvania', or 'Virginia' from usa\_states\_sub:

usa\_states\_sub.state[usa\_states\_sub.state.str.contains('Massachusetts|Pennsylvania|Rhode Island|Virginia')]

20 Massachusetts[upper-alpha 3]

37 Pennsylvania[upper-alpha 3]

38 Rhode Island[upper-alpha 4]

45 Virginia[upper-alpha 3]

47 West Virginia

Name: state, dtype: object

Delete square brackets and their contents and try again:

#Code task 33#

#Use pandas' Series' `replace()` method to replace anything within square brackets (including the brackets)

#with the empty string. Do this inplace, so you need to specify the arguments:

#to\_replace='\[.\*\]' #literal square bracket followed by anything or nothing followed by literal closing bracket

#value='' #empty string as replacement

#regex=True #we used a regex in our `to\_replace` argument

#inplace=True #Do this "in place"

usa\_states\_sub.state.\_\_\_(to\_replace=\_\_\_, value=\_\_, regex=\_\_\_, inplace=\_\_\_)

usa\_states\_sub.state[usa\_states\_sub.state.str.contains('Massachusetts|Pennsylvania|Rhode Island|Virginia')]

#Code task 34#

#And now verify none of our states are missing by checking that there are no states in

#state\_summary that are not in usa\_states\_sub (as earlier using `set()`)

missing\_states = \_\_\_(state\_summary.state) - \_\_\_(usa\_states\_sub.state)

missing\_states

Better! You have an empty set for missing states now. You can confidently add the population and state area columns to the ski resort data.

#Code task 35#

#Use 'state\_summary's `merge()` method to combine our new data in 'usa\_states\_sub'

#specify the arguments how='left' and on='state'

state\_summary = state\_summary.\_\_\_(usa\_states\_sub, \_\_\_=\_\_\_, \_\_\_=\_\_\_)

state\_summary.head()

Having created this data frame of summary statistics for various states, it would seem obvious to join this with the ski resort data to augment it with this additional data. You will do this, but not now. In the next notebook you will be exploring the data, including the relationships between the states. For that you want a separate row for each state, as you have here, and joining the data this soon means you'd need to separate and eliminate redundances in the state data when you wanted it.

**2.11 Target Feature**

Finally, what will your target be when modelling ticket price? What relationship is there between weekday and weekend prices?

#Code task 36#

#Use ski\_data's `plot()` method to create a scatterplot (kind='scatter') with 'AdultWeekday' on the x-axis and

#'AdultWeekend' on the y-axis

ski\_data.\_\_\_(x=\_\_\_, y=\_\_\_, kind=\_\_\_);

A couple of observations can be made. Firstly, there is a clear line where weekend and weekday prices are equal. Weekend prices being higher than weekday prices seem restricted to sub $100 resorts. Recall from the boxplot earlier that the distribution for weekday and weekend prices in Montana seemed equal. Is this confirmed in the actual data for each resort? Big Mountain resort is in Montana, so the relationship between these quantities in this state are particularly relevant.

#Code task 37#

#Use the loc accessor on ski\_data to print the 'AdultWeekend' and 'AdultWeekday' columns for Montana only

ski\_data.\_\_\_[ski\_data.state == \_\_\_, [\_\_\_, \_\_\_]]

Is there any reason to prefer weekend or weekday prices? Which is missing the least?

ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum()

AdultWeekend 4

AdultWeekday 7

dtype: int64

Weekend prices have the least missing values of the two, so drop the weekday prices and then keep just the rows that have weekend price.

ski\_data.drop(columns='AdultWeekday', inplace=True)

ski\_data.dropna(subset=['AdultWeekend'], inplace=True)

ski\_data.shape

(277, 25)

Perform a final quick check on the data.

**2.11.1 Number Of Missing Values By Row - Resort**

Having dropped rows missing the desired target ticket price, what degree of missingness do you have for the remaining rows?

missing = pd.concat([ski\_data.isnull().sum(axis=1), 100 \* ski\_data.isnull().mean(axis=1)], axis=1)

missing.columns=['count', '%']

missing.sort\_values(by='count', ascending=False).head(10)

|  | **count** | **%** |
| --- | --- | --- |
| **329** | 5 | 20.0 |
| **62** | 5 | 20.0 |
| **141** | 5 | 20.0 |
| **86** | 5 | 20.0 |
| **74** | 5 | 20.0 |
| **146** | 5 | 20.0 |
| **184** | 4 | 16.0 |
| **108** | 4 | 16.0 |
| **198** | 4 | 16.0 |
| **39** | 4 | 16.0 |

These seem possibly curiously quantized...

missing['%'].unique()

array([ 0., 4., 8., 12., 16., 20.])

Yes, the percentage of missing values per row appear in multiples of 4.

missing['%'].value\_counts()

0.0 107

4.0 94

8.0 45

12.0 15

16.0 10

20.0 6

Name: %, dtype: int64

This is almost as if values have been removed artificially... Nevertheless, what you don't know is how useful the missing features are in predicting ticket price. You shouldn't just drop rows that are missing several useless features.

ski\_data.info()

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

Int64Index: 277 entries, 0 to 329

Data columns (total 25 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name 277 non-null object

1 Region 277 non-null object

2 state 277 non-null object

3 summit\_elev 277 non-null int64

4 vertical\_drop 277 non-null int64

5 base\_elev 277 non-null int64

6 trams 277 non-null int64

7 fastSixes 277 non-null int64

8 fastQuads 277 non-null int64

9 quad 277 non-null int64

10 triple 277 non-null int64

11 double 277 non-null int64

12 surface 277 non-null int64

13 total\_chairs 277 non-null int64

14 Runs 274 non-null float64

15 TerrainParks 233 non-null float64

16 LongestRun\_mi 272 non-null float64

17 SkiableTerrain\_ac 275 non-null float64

18 Snow Making\_ac 240 non-null float64

19 daysOpenLastYear 233 non-null float64

20 yearsOpen 277 non-null float64

21 averageSnowfall 268 non-null float64

22 AdultWeekend 277 non-null float64

23 projectedDaysOpen 236 non-null float64

24 NightSkiing\_ac 163 non-null float64

dtypes: float64(11), int64(11), object(3)

memory usage: 56.3+ KB

There are still some missing values, and it's good to be aware of this, but leave them as is for now.

**2.12 Save data**

ski\_data.shape

(277, 25)

Save this to your data directory, separately. Note that you were provided with the data in raw\_data and you should saving derived data in a separate location. This guards against overwriting our original data.

# save the data to a new csv file

datapath = '../data'

save\_file(ski\_data, 'ski\_data\_cleaned.csv', datapath)

# save the state\_summary separately.

datapath = '../data'

save\_file(state\_summary, 'state\_summary.csv', datapath)

**2.13 Summary**

**Q: 3** Write a summary statement that highlights the key processes and findings from this notebook. This should include information such as the original number of rows in the data, whether our own resort was actually present etc. What columns, if any, have been removed? Any rows? Summarise the reasons why. Were any other issues found? What remedial actions did you take? State where you are in the project. Can you confirm what the target feature is for your desire to predict ticket price? How many rows were left in the data? Hint: this is a great opportunity to reread your notebook, check all cells have been executed in order and from a "blank slate" (restarting the kernel will do this), and that your workflow makes sense and follows a logical pattern. As you do this you can pull out salient information for inclusion in this summary. Thus, this section will provide an important overview of "what" and "why" without having to dive into the "how" or any unproductive or inconclusive steps along the way.

**A: 3** Your answer here

[Jupyter Notebook](http://localhost:8890/tree?token=34101dfefd012e8bd0b705731d01b2a4049a6063cf628fdf)

02\_data\_wrangling Last Checkpoint: 21 hours ago (autosaved)

Python 3 (ipykernel)

* [File](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Edit](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [View](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Insert](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Cell](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Kernel](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Widgets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)
* [Help](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb)

**2 Data wrangling**

**2.1 Contents**

* [2 Data wrangling](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2_Data_wrangling)
  + [2.1 Contents](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.1_Contents)
  + [2.2 Introduction](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.2_Introduction)
    - [2.2.1 Recap Of Data Science Problem](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.2.1_Recap_Of_Data_Science_Problem)
    - [2.2.2 Introduction To Notebook](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.2.2_Introduction_To_Notebook)
  + [2.3 Imports](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.3_Imports)
  + [2.4 Objectives](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.4_Objectives)
  + [2.5 Load The Ski Resort Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.5_Load_The_Ski_Resort_Data)
  + [2.6 Explore The Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6_Explore_The_Data)
    - [2.6.1 Find Your Resort Of Interest](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.1_Find_Your_Resort_Of_Interest)
    - [2.6.2 Number Of Missing Values By Column](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.2_Number_Of_Missing_Values_By_Column)
    - [2.6.3 Categorical Features](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3_Categorical_Features)
      * [2.6.3.1 Unique Resort Names](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.1_Unique_Resort_Names)
      * [2.6.3.2 Region And State](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.2_Region_And_State)
      * [2.6.3.3 Number of distinct regions and states](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.3_Number_of_distinct_regions_and_states)
      * [2.6.3.4 Distribution Of Resorts By Region And State](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.4_Distribution_Of_Resorts_By_Region_And_State)
      * [2.6.3.5 Distribution Of Ticket Price By State](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.5_Distribution_Of_Ticket_Price_By_State)
        + [2.6.3.5.1 Average weekend and weekday price by state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.5.1_Average_weekend_and_weekday_price_by_state)
        + [2.6.3.5.2 Distribution of weekday and weekend price by state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.3.5.2_Distribution_of_weekday_and_weekend_price_by_state)
    - [2.6.4 Numeric Features](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4_Numeric_Features)
      * [2.6.4.1 Numeric data summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.1_Numeric_data_summary)
      * [2.6.4.2 Distributions Of Feature Values](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.2_Distributions_Of_Feature_Values)
        + [2.6.4.2.1 SkiableTerrain\_ac](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.2.1_SkiableTerrain_ac)
        + [2.6.4.2.2 Snow Making\_ac](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.2.2_Snow_Making_ac)
        + [2.6.4.2.3 fastEight](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.2.3_fastEight)
        + [2.6.4.2.4 fastSixes and Trams](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.6.4.2.4_fastSixes_and_Trams)
  + [2.7 Derive State-wide Summary Statistics For Our Market Segment](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.7_Derive_State-wide_Summary_Statistics_For_Our_Market_Segment)
  + [2.8 Drop Rows With No Price Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.8_Drop_Rows_With_No_Price_Data)
  + [2.9 Review distributions](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.9_Review_distributions)
  + [2.10 Population data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.10_Population_data)
  + [2.11 Target Feature](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.11_Target_Feature)
    - [2.11.1 Number Of Missing Values By Row - Resort](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.11.1_Number_Of_Missing_Values_By_Row_-_Resort)
  + [2.12 Save data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.12_Save_data)
  + [2.13 Summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/02_data_wrangling.ipynb#2.13_Summary)

**2.2 Introduction**

This step focuses on collecting your data, organizing it, and making sure it's well defined. Paying attention to these tasks will pay off greatly later on. Some data cleaning can be done at this stage, but it's important not to be overzealous in your cleaning before you've explored the data to better understand it.

**2.2.1 Recap Of Data Science Problem**

The purpose of this data science project is to come up with a pricing model for ski resort tickets in our market segment. Big Mountain suspects it may not be maximizing its returns, relative to its position in the market. It also does not have a strong sense of what facilities matter most to visitors, particularly which ones they're most likely to pay more for. This project aims to build a predictive model for ticket price based on a number of facilities, or properties, boasted by resorts (*at the resorts).* This model will be used to provide guidance for Big Mountain's pricing and future facility investment plans.

**2.2.2 Introduction To Notebook**

Notebooks grow organically as we explore our data. If you used paper notebooks, you could discover a mistake and cross out or revise some earlier work. Later work may give you a reason to revisit earlier work and explore it further. The great thing about Jupyter notebooks is that you can edit, add, and move cells around without needing to cross out figures or scrawl in the margin. However, this means you can lose track of your changes easily. If you worked in a regulated environment, the company may have a a policy of always dating entries and clearly crossing out any mistakes, with your initials and the date.

**Best practice here is to commit your changes using a version control system such as Git.** Try to get into the habit of adding and committing your files to the Git repository you're working in after you save them. You're are working in a Git repository, right? If you make a significant change, save the notebook and commit it to Git. In fact, if you're about to make a significant change, it's a good idea to commit before as well. Then if the change is a mess, you've got the previous version to go back to.

**Another best practice with notebooks is to try to keep them organized with helpful headings and comments.** Not only can a good structure, but associated headings help you keep track of what you've done and your current focus. Anyone reading your notebook will have a much easier time following the flow of work. Remember, that 'anyone' will most likely be you. Be kind to future you!

In this notebook, note how we try to use well structured, helpful headings that frequently are self-explanatory, and we make a brief note after any results to highlight key takeaways. This is an immense help to anyone reading your notebook and it will greatly help you when you come to summarise your findings. **Top tip: jot down key findings in a final summary at the end of the notebook as they arise. You can tidy this up later.** This is a great way to ensure important results don't get lost in the middle of your notebooks.

In this, and subsequent notebooks, there are coding tasks marked with #Code task n# with code to complete. The \_\_\_ will guide you to where you need to insert code.

**2.3 Imports**

Placing your imports all together at the start of your notebook means you only need to consult one place to check your notebook's dependencies. By all means import something 'in situ' later on when you're experimenting, but if the imported dependency ends up being kept, you should subsequently move the import statement here with the rest.



#Code task 1#

#Import pandas, matplotlib.pyplot, and seaborn in the correct lines below

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import os

​

from library.sb\_utils import save\_file

​

**2.4 Objectives**

There are some fundamental questions to resolve in this notebook before you move on.

* Do you think you may have the data you need to tackle the desired question?
  + Have you identified the required target value?
  + Do you have potentially useful features?
* Do you have any fundamental issues with the data?

**2.5 Load The Ski Resort Data**



# the supplied CSV data file is the raw\_data directory

ski\_data = pd.read\_csv('../raw\_data/ski\_resort\_data.csv')

Good first steps in auditing the data are the info method and displaying the first few records with head.



#Code task 2#

#Call the info method on ski\_data to see a summary of the data

ski\_data.info()

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

RangeIndex: 330 entries, 0 to 329

Data columns (total 27 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name 330 non-null object

1 Region 330 non-null object

2 state 330 non-null object

3 summit\_elev 330 non-null int64

4 vertical\_drop 330 non-null int64

5 base\_elev 330 non-null int64

6 trams 330 non-null int64

7 fastEight 164 non-null float64

8 fastSixes 330 non-null int64

9 fastQuads 330 non-null int64

10 quad 330 non-null int64

11 triple 330 non-null int64

12 double 330 non-null int64

13 surface 330 non-null int64

14 total\_chairs 330 non-null int64

15 Runs 326 non-null float64

16 TerrainParks 279 non-null float64

17 LongestRun\_mi 325 non-null float64

18 SkiableTerrain\_ac 327 non-null float64

19 Snow Making\_ac 284 non-null float64

20 daysOpenLastYear 279 non-null float64

21 yearsOpen 329 non-null float64

22 averageSnowfall 316 non-null float64

23 AdultWeekday 276 non-null float64

24 AdultWeekend 279 non-null float64

25 projectedDaysOpen 283 non-null float64

26 NightSkiing\_ac 187 non-null float64

dtypes: float64(13), int64(11), object(3)

memory usage: 69.7+ KB

AdultWeekday is the price of an adult weekday ticket. AdultWeekend is the price of an adult weekend ticket. The other columns are potential features.

This immediately raises the question of what quantity will you want to model? You know you want to model the ticket price, but you realise there are two kinds of ticket price!



#Code task 3#

#Call the head method on ski\_data to print the first several rows of the data

ski\_data.head()

|  | **Name** | **Region** | **state** | **summit\_elev** | **vertical\_drop** | **base\_elev** | **trams** | **fastEight** | **fastSixes** | **fastQuads** | **...** | **LongestRun\_mi** | **SkiableTerrain\_ac** | **Snow Making\_ac** | **daysOpenLastYear** | **yearsOpen** | **averageSnowfall** | **AdultWeekday** | **AdultWeekend** | **projectedDaysOpen** | **NightSkiing\_ac** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alyeska Resort | Alaska | Alaska | 3939 | 2500 | 250 | 1 | 0.0 | 0 | 2 | ... | 1.0 | 1610.0 | 113.0 | 150.0 | 60.0 | 669.0 | 65.0 | 85.0 | 150.0 | 550.0 |
| **1** | Eaglecrest Ski Area | Alaska | Alaska | 2600 | 1540 | 1200 | 0 | 0.0 | 0 | 0 | ... | 2.0 | 640.0 | 60.0 | 45.0 | 44.0 | 350.0 | 47.0 | 53.0 | 90.0 | NaN |
| **2** | Hilltop Ski Area | Alaska | Alaska | 2090 | 294 | 1796 | 0 | 0.0 | 0 | 0 | ... | 1.0 | 30.0 | 30.0 | 150.0 | 36.0 | 69.0 | 30.0 | 34.0 | 152.0 | 30.0 |
| **3** | Arizona Snowbowl | Arizona | Arizona | 11500 | 2300 | 9200 | 0 | 0.0 | 1 | 0 | ... | 2.0 | 777.0 | 104.0 | 122.0 | 81.0 | 260.0 | 89.0 | 89.0 | 122.0 | NaN |
| **4** | Sunrise Park Resort | Arizona | Arizona | 11100 | 1800 | 9200 | 0 | NaN | 0 | 1 | ... | 1.2 | 800.0 | 80.0 | 115.0 | 49.0 | 250.0 | 74.0 | 78.0 | 104.0 | 80.0 |

5 rows × 27 columns

The output above suggests you've made a good start getting the ski resort data organized. You have plausible column headings. You can already see you have a missing value in the fastEight column

**2.6 Explore The Data**

**2.6.1 Find Your Resort Of Interest**

Your resort of interest is called Big Mountain Resort. Check it's in the data:



#Code task 4#

#Filter the ski\_data dataframe to display just the row for our resort with the name 'Big Mountain Resort'

#Hint: you will find that the transpose of the row will give a nicer output. DataFrame's do have a

#transpose method, but you can access this conveniently with the `T` property.

ski\_data[ski\_data.Name == 'Big Mountain Resort'].T

|  | **151** |
| --- | --- |
| **Name** | Big Mountain Resort |
| **Region** | Montana |
| **state** | Montana |
| **summit\_elev** | 6817 |
| **vertical\_drop** | 2353 |
| **base\_elev** | 4464 |
| **trams** | 0 |
| **fastEight** | 0.0 |
| **fastSixes** | 0 |
| **fastQuads** | 3 |
| **quad** | 2 |
| **triple** | 6 |
| **double** | 0 |
| **surface** | 3 |
| **total\_chairs** | 14 |
| **Runs** | 105.0 |
| **TerrainParks** | 4.0 |
| **LongestRun\_mi** | 3.3 |
| **SkiableTerrain\_ac** | 3000.0 |
| **Snow Making\_ac** | 600.0 |
| **daysOpenLastYear** | 123.0 |
| **yearsOpen** | 72.0 |
| **averageSnowfall** | 333.0 |
| **AdultWeekday** | 81.0 |
| **AdultWeekend** | 81.0 |
| **projectedDaysOpen** | 123.0 |
| **NightSkiing\_ac** | 600.0 |

It's good that your resort doesn't appear to have any missing values.

**2.6.2 Number Of Missing Values By Column**

Count the number of missing values in each column and sort them.



#Code task 5#

#Count (using `.sum()`) the number of missing values (`.isnull()`) in each column of

#ski\_data as well as the percentages (using `.mean()` instead of `.sum()`).

#Order them (increasing or decreasing) using sort\_values

#Call `pd.concat` to present these in a single table (DataFrame) with the helpful column names 'count' and '%'

missing = pd.concat([ski\_data.isnull().sum(), 100 \* ski\_data.isnull().mean()], axis=1)

missing.columns=['count','%']

missing.sort\_values(by=['count','%'])

|  | **count** | **%** |
| --- | --- | --- |
| **Name** | 0 | 0.000000 |
| **Region** | 0 | 0.000000 |
| **state** | 0 | 0.000000 |
| **summit\_elev** | 0 | 0.000000 |
| **vertical\_drop** | 0 | 0.000000 |
| **base\_elev** | 0 | 0.000000 |
| **trams** | 0 | 0.000000 |
| **fastSixes** | 0 | 0.000000 |
| **fastQuads** | 0 | 0.000000 |
| **quad** | 0 | 0.000000 |
| **triple** | 0 | 0.000000 |
| **double** | 0 | 0.000000 |
| **surface** | 0 | 0.000000 |
| **total\_chairs** | 0 | 0.000000 |
| **yearsOpen** | 1 | 0.303030 |
| **SkiableTerrain\_ac** | 3 | 0.909091 |
| **Runs** | 4 | 1.212121 |
| **LongestRun\_mi** | 5 | 1.515152 |
| **averageSnowfall** | 14 | 4.242424 |
| **Snow Making\_ac** | 46 | 13.939394 |
| **projectedDaysOpen** | 47 | 14.242424 |
| **TerrainParks** | 51 | 15.454545 |
| **daysOpenLastYear** | 51 | 15.454545 |
| **AdultWeekend** | 51 | 15.454545 |
| **AdultWeekday** | 54 | 16.363636 |
| **NightSkiing\_ac** | 143 | 43.333333 |
| **fastEight** | 166 | 50.303030 |

fastEight has the most missing values, at just over 50%. Unfortunately, you see you're also missing quite a few of your desired target quantity, the ticket price, which is missing 15-16% of values. AdultWeekday is missing in a few more records than AdultWeekend. What overlap is there in these missing values? This is a question you'll want to investigate. You should also point out that isnull() is not the only indicator of missing data. Sometimes 'missingness' can be encoded, perhaps by a -1 or 999. Such values are typically chosen because they are "obviously" not genuine values. If you were capturing data on people's heights and weights but missing someone's height, you could certainly encode that as a 0 because no one has a height of zero (in any units). Yet such entries would not be revealed by isnull(). Here, you need a data dictionary and/or to spot such values as part of looking for outliers. Someone with a height of zero should definitely show up as an outlier!

**2.6.3 Categorical Features**

So far you've examined only the numeric features. Now you inspect categorical ones such as resort name and state. These are discrete entities. 'Alaska' is a name. Although names can be sorted alphabetically, it makes no sense to take the average of 'Alaska' and 'Arizona'. Similarly, 'Alaska' is before 'Arizona' only lexicographically; it is neither 'less than' nor 'greater than' 'Arizona'. As such, they tend to require different handling than strictly numeric quantities. Note, a feature *can* be numeric but also categorical. For example, instead of giving the number of fastEight lifts, a feature might be has\_fastEights and have the value 0 or 1 to denote absence or presence of such a lift. In such a case it would not make sense to take an average of this or perform other mathematical calculations on it. Although you digress a little to make a point, month numbers are also, strictly speaking, categorical features. Yes, when a month is represented by its number (1 for January, 2 for Februrary etc.) it provides a convenient way to graph trends over a year. And, arguably, there is some logical interpretation of the average of 1 and 3 (January and March) being 2 (February). However, clearly December of one years precedes January of the next and yet 12 as a number is not less than 1. The numeric quantities in the section above are truly numeric; they are the number of feet in the drop, or acres or years open or the amount of snowfall etc.



#Code task 6#

#Use ski\_data's `select\_dtypes` method to select columns of dtype 'object'

ski\_data.select\_dtypes('object')

|  | **Name** | **Region** | **state** |
| --- | --- | --- | --- |
| **0** | Alyeska Resort | Alaska | Alaska |
| **1** | Eaglecrest Ski Area | Alaska | Alaska |
| **2** | Hilltop Ski Area | Alaska | Alaska |
| **3** | Arizona Snowbowl | Arizona | Arizona |
| **4** | Sunrise Park Resort | Arizona | Arizona |
| **...** | ... | ... | ... |
| **325** | Meadowlark Ski Lodge | Wyoming | Wyoming |
| **326** | Sleeping Giant Ski Resort | Wyoming | Wyoming |
| **327** | Snow King Resort | Wyoming | Wyoming |
| **328** | Snowy Range Ski & Recreation Area | Wyoming | Wyoming |
| **329** | White Pine Ski Area | Wyoming | Wyoming |

330 rows × 3 columns

You saw earlier on that these three columns had no missing values. But are there any other issues with these columns? Sensible questions to ask here include:

* Is Name (or at least a combination of Name/Region/State) unique?
* Is Region always the same as state?

**2.6.3.1 Unique Resort Names**



#Code task 7#

#Use pandas' Series method `value\_counts` to find any duplicated resort names

ski\_data['Name'].value\_counts().head()

Crystal Mountain 2

Alyeska Resort 1

Brandywine 1

Boston Mills 1

Alpine Valley 1

Name: Name, dtype: int64

You have a duplicated resort name: Crystal Mountain.

**Q: 1** Is this resort duplicated if you take into account Region and/or state as well?



#Code task 8#

#Concatenate the string columns 'Name' and 'Region' and count the values again (as above)

(ski\_data['Name'] + ', ' + ski\_data['Region']).value\_counts().head()

Alyeska Resort, Alaska 1

Snow Trails, Ohio 1

Brandywine, Ohio 1

Boston Mills, Ohio 1

Alpine Valley, Ohio 1

dtype: int64



#Code task 9#

#Concatenate 'Name' and 'state' and count the values again (as above)

(ski\_data['Name'] + ', ' + ski\_data['state']).value\_counts().head()

Alyeska Resort, Alaska 1

Snow Trails, Ohio 1

Brandywine, Ohio 1

Boston Mills, Ohio 1

Alpine Valley, Ohio 1

dtype: int64

**NB** because you know value\_counts() sorts descending, you can use the head() method and know the rest of the counts must be 1.

**A: 1** No, there are 2 different resorts with same Name of Crystal Mountain but in different Region and state. Therefore, unique.



ski\_data[ski\_data['Name'] == 'Crystal Mountain']

|  | **Name** | **Region** | **state** | **summit\_elev** | **vertical\_drop** | **base\_elev** | **trams** | **fastEight** | **fastSixes** | **fastQuads** | **...** | **LongestRun\_mi** | **SkiableTerrain\_ac** | **Snow Making\_ac** | **daysOpenLastYear** | **yearsOpen** | **averageSnowfall** | **AdultWeekday** | **AdultWeekend** | **projectedDaysOpen** | **NightSkiing\_ac** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **104** | Crystal Mountain | Michigan | Michigan | 1132 | 375 | 757 | 0 | 0.0 | 0 | 1 | ... | 0.3 | 102.0 | 96.0 | 120.0 | 63.0 | 132.0 | 54.0 | 64.0 | 135.0 | 56.0 |
| **295** | Crystal Mountain | Washington | Washington | 7012 | 3100 | 4400 | 1 | NaN | 2 | 2 | ... | 2.5 | 2600.0 | 10.0 | NaN | 57.0 | 486.0 | 99.0 | 99.0 | NaN | NaN |

2 rows × 27 columns

So there are two Crystal Mountain resorts, but they are clearly two different resorts in two different states. This is a powerful signal that you have unique records on each row.

**2.6.3.2 Region And State**

What's the relationship between region and state?

You know they are the same in many cases (e.g. both the Region and the state are given as 'Michigan'). In how many cases do they differ?



#Code task 10#

#Calculate the number of times Region does not equal state

(ski\_data.Region != ski\_data.state).value\_counts()

False 297

True 33

dtype: int64

You know what a state is. What is a region? You can tabulate the distinct values along with their respective frequencies using value\_counts().



ski\_data['Region'].value\_counts()

New York 33

Michigan 29

Sierra Nevada 22

Colorado 22

Pennsylvania 19

Wisconsin 16

New Hampshire 16

Vermont 15

Minnesota 14

Idaho 12

Montana 12

Massachusetts 11

Washington 10

New Mexico 9

Maine 9

Wyoming 8

Utah 7

Salt Lake City 6

North Carolina 6

Oregon 6

Connecticut 5

Ohio 5

Virginia 4

West Virginia 4

Illinois 4

Mt. Hood 4

Alaska 3

Iowa 3

South Dakota 2

Arizona 2

Nevada 2

Missouri 2

Indiana 2

New Jersey 2

Rhode Island 1

Tennessee 1

Maryland 1

Northern California 1

Name: Region, dtype: int64

A casual inspection by eye reveals some non-state names such as Sierra Nevada, Salt Lake City, and Northern California. Tabulate the differences between Region and state. On a note regarding scaling to larger data sets, you might wonder how you could spot such cases when presented with millions of rows. This is an interesting point. Imagine you have access to a database with a Region and state column in a table and there are millions of rows. You wouldn't eyeball all the rows looking for differences! Bear in mind that our first interest lies in establishing the answer to the question "Are they always the same?" One approach might be to ask the database to return records where they differ, but limit the output to 10 rows. If there were differences, you'd only get up to 10 results, and so you wouldn't know whether you'd located all differences, but you'd know that there were 'a nonzero number' of differences. If you got an empty result set back, then you would know that the two columns always had the same value. At the risk of digressing, some values in one column only might be NULL (missing) and different databases treat NULL differently, so be aware that on many an occasion a seamingly 'simple' question gets very interesting to answer very quickly!



#Code task 11#

#Filter the ski\_data dataframe for rows where 'Region' and 'state' are different,

#group that by 'state' and perform `value\_counts` on the 'Region'

(ski\_data[ski\_data.Region != ski\_data.state]

.groupby('state')['Region']

.value\_counts())

state Region

California Sierra Nevada 20

Northern California 1

Nevada Sierra Nevada 2

Oregon Mt. Hood 4

Utah Salt Lake City 6

Name: Region, dtype: int64

The vast majority of the differences are in California, with most Regions being called Sierra Nevada and just one referred to as Northern California.

**2.6.3.3 Number of distinct regions and states**



#Code task 12#

#Select the 'Region' and 'state' columns from ski\_data and use the `nunique` method to calculate

#the number of unique values in each

ski\_data[['Region', 'state']].nunique()

Region 38

state 35

dtype: int64

Because a few states are split across multiple named regions, there are slightly more unique regions than states.

**2.6.3.4 Distribution Of Resorts By Region And State**

If this is your first time using [matplotlib](https://matplotlib.org/3.2.2/index.html)'s [subplots](https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.subplots.html), you may find the online documentation useful.



#Code task 13#

#Create two subplots on 1 row and 2 columns with a figsize of (12, 8)

fig, ax = plt.subplots(1, 2, figsize=(12,8))

#Specify a horizontal barplot ('barh') as kind of plot (kind=)

ski\_data.Region.value\_counts().plot(kind='barh', ax=ax[0])

#Give the plot a helpful title of 'Region'

ax[0].set\_title('Region')

#Label the xaxis 'Count'

ax[0].set\_xlabel('Count')

#Specify a horizontal barplot ('barh') as kind of plot (kind=)

ski\_data.state.value\_counts().plot(kind='barh', ax=ax[1])

#Give the plot a helpful title of 'state'

ax[1].set\_title('state')

#Label the xaxis 'Count'

ax[1].set\_xlabel('Count')

#Give the subplots a little "breathing room" with a wspace of 0.5

plt.subplots\_adjust(wspace=0.5);

#You're encouraged to explore a few different figure sizes, orientations, and spacing here

# as the importance of easy-to-read and informative figures is frequently understated

# and you will find the ability to tweak figures invaluable later on

Chart, bar chart, histogram

Description automatically generated

How's your geography? Looking at the distribution of States, you see New York accounting for the majority of resorts. Our target resort is in Montana, which comes in at 13th place. You should think carefully about how, or whether, you use this information. Does New York command a premium because of its proximity to population? Even if a resort's State were a useful predictor of ticket price, your main interest lies in Montana. Would you want a model that is skewed for accuracy by New York? Should you just filter for Montana and create a Montana-specific model? This would slash your available data volume. Your problem task includes the contextual insight that the data are for resorts all belonging to the same market share. This suggests one might expect prices to be similar amongst them. You can look into this. A boxplot grouped by State is an ideal way to quickly compare prices. Another side note worth bringing up here is that, in reality, the best approach here definitely would include consulting with the client or other domain expert. They might know of good reasons for treating states equivalently or differently. The data scientist is rarely the final arbiter of such a decision. But here, you'll see if we can find any supporting evidence for treating states the same or differently.

**2.6.3.5 Distribution Of Ticket Price By State**

Our primary focus is our Big Mountain resort, in Montana. Does the state give you any clues to help decide what your primary target response feature should be (weekend or weekday ticket prices)?

**2.6.3.5.1 Average weekend and weekday price by state**



#Code task 14#

# Calculate average weekday and weekend price by state and sort by the average of the two

# Hint: use the pattern dataframe.groupby(<grouping variable>)[<list of columns>].mean()

state\_price\_means = ski\_data.groupby('state')[['AdultWeekday', 'AdultWeekend']].mean()

state\_price\_means.head()

|  | **AdultWeekday** | **AdultWeekend** |
| --- | --- | --- |
| **state** |  |  |
| **Alaska** | 47.333333 | 57.333333 |
| **Arizona** | 81.500000 | 83.500000 |
| **California** | 78.214286 | 81.416667 |
| **Colorado** | 90.714286 | 90.714286 |
| **Connecticut** | 47.800000 | 56.800000 |



# The next bit simply reorders the index by increasing average of weekday and weekend prices

# Compare the index order you get from

# state\_price\_means.index

# with

# state\_price\_means.mean(axis=1).sort\_values(ascending=False).index

# See how this expression simply sits within the reindex()

(state\_price\_means.reindex(index=state\_price\_means.mean(axis=1)

.sort\_values(ascending=False)

.index)

.plot(kind='barh', figsize=(10, 10), title='Average ticket price by State'))

plt.xlabel('Price ($)');

Chart, bar chart

Description automatically generated



The figure above represents a dataframe with two columns, one for the average prices of each kind of ticket. This tells you how the average ticket price varies from state to state. But can you get more insight into the difference in the distributions between states?

Object `states` not found.



The figure above represents a dataframe with two columns, one for the average prices of each kind of ticket. This tells you how the average ticket price varies from state to state. But can you get more insight into the difference in the distributions between states

**2.6.3.5.2 Distribution of weekday and weekend price by state**

Next, you can transform the data into a single column for price with a new categorical column that represents the ticket type.



#Code task 15#

#Use the pd.melt function, pass in the ski\_data columns 'state', 'AdultWeekday', and 'Adultweekend' only,

#specify 'state' for `id\_vars`

#gather the ticket prices from the 'Adultweekday' and 'AdultWeekend' columns using the `value\_vars` argument,

#call the resultant price column 'Price' via the `value\_name` argument,

#name the weekday/weekend indicator column 'Ticket' via the `var\_name` argument

ticket\_prices = pd.melt(ski\_data[['state','AdultWeekday','AdultWeekend']],

id\_vars='state',

var\_name='Ticket',

value\_vars=['AdultWeekday','AdultWeekend'],

value\_name='Price')



ticket\_prices.head()

|  | **state** | **Ticket** | **Price** |
| --- | --- | --- | --- |
| **0** | Alaska | AdultWeekday | 65.0 |
| **1** | Alaska | AdultWeekday | 47.0 |
| **2** | Alaska | AdultWeekday | 30.0 |
| **3** | Arizona | AdultWeekday | 89.0 |
| **4** | Arizona | AdultWeekday | 74.0 |

This is now in a format we can pass to [seaborn](https://seaborn.pydata.org/)'s [boxplot](https://seaborn.pydata.org/generated/seaborn.boxplot.html) function to create boxplots of the ticket price distributions for each ticket type for each state.



#Code task 16#

#Create a seaborn boxplot of the ticket price dataframe we created above,

#with 'state' on the x-axis, 'Price' as the y-value, and a hue that indicates 'Ticket'

#This will use boxplot's x, y, hue, and data arguments.

plt.subplots(figsize=(12, 8))

sns.boxplot(x='state', y='Price', hue='Ticket', data=ticket\_prices)

plt.xticks(rotation='vertical')

plt.ylabel('Price ($)')

plt.xlabel('State');

Chart, bar chart

Description automatically generated

Aside from some relatively expensive ticket prices in California, Colorado, and Utah, most prices appear to lie in a broad band from around 25 to over 100 dollars. Some States show more variability than others. Montana and South Dakota, for example, both show fairly small variability as well as matching weekend and weekday ticket prices. Nevada and Utah, on the other hand, show the most range in prices. Some States, notably North Carolina and Virginia, have weekend prices far higher than weekday prices. You could be inspired from this exploration to consider a few potential groupings of resorts, those with low spread, those with lower averages, and those that charge a premium for weekend tickets. However, you're told that you are taking all resorts to be part of the same market share, you could argue against further segment the resorts. Nevertheless, ways to consider using the State information in your modelling include:

* disregard State completely
* retain all State information
* retain State in the form of Montana vs not Montana, as our target resort is in Montana

You've also noted another effect above: some States show a marked difference between weekday and weekend ticket prices. It may make sense to allow a model to take into account not just State but also weekend vs weekday.

Thus we currently have two main questions you want to resolve:

* What do you do about the two types of ticket price?
* What do you do about the state information?

**2.6.4 Numeric Features**



Having decided to reserve judgement on how exactly you utilize the State, turn your attention to cleaning the numeric features.

File "C:\Users\charrie\AppData\Local\Temp\ipykernel\_15976\2281747816.py", line 1

Having decided to reserve judgement on how exactly you utilize the State, turn your attention to cleaning the numeric features.

^

SyntaxError: invalid syntax

**2.6.4.1 Numeric data summary**



#Code task 17#

#Call ski\_data's `describe` method for a statistical summary of the numerical columns

#Hint: there are fewer summary stat columns than features, so displaying the transpose

#will be useful again

ski\_data.describe().T

Recall you're missing the ticket prices for some 16% of resorts. This is a fundamental problem that means you simply lack the required data for those resorts and will have to drop those records. But you may have a weekend price and not a weekday price, or vice versa. You want to keep any price you have.



missing\_price = ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum(axis=1)

missing\_price.value\_counts()/len(missing\_price) \* 100

Just over 82% of resorts have no missing ticket price, 3% are missing one value, and 14% are missing both. You will definitely want to drop the records for which you have no price information, however you will not do so just yet. There may still be useful information about the distributions of other features in that 14% of the data.

**2.6.4.2 Distributions Of Feature Values**

Note that, although we are still in the 'data wrangling and cleaning' phase rather than exploratory data analysis, looking at distributions of features is immensely useful in getting a feel for whether the values look sensible and whether there are any obvious outliers to investigate. Some exploratory data analysis belongs here, and data wrangling will inevitably occur later on. It's more a matter of emphasis. Here, we're interesting in focusing on whether distributions look plausible or wrong. Later on, we're more interested in relationships and patterns.



#Code task 18#

#Call ski\_data's `hist` method to plot histograms of each of the numeric features

#Try passing it an argument figsize=(15,10)

#Try calling plt.subplots\_adjust() with an argument hspace=0.5 to adjust the spacing

#It's important you create legible and easy-to-read plots

ski\_data.hist(figsize=(15,10))

plt.subplots\_adjust(hspace=0.5);

#Hint: notice how the terminating ';' "swallows" some messy output and leads to a tidier notebook

What features do we have possible cause for concern about and why?

* SkiableTerrain\_ac because values are clustered down the low end,
* Snow Making\_ac for the same reason,
* fastEight because all but one value is 0 so it has very little variance, and half the values are missing,
* fastSixes raises an amber flag; it has more variability, but still mostly 0,
* trams also may get an amber flag for the same reason,
* yearsOpen because most values are low but it has a maximum of 2019, which strongly suggests someone recorded calendar year rather than number of years.

**2.6.4.2.1 SkiableTerrain\_ac**



#Code task 19#

#Filter the 'SkiableTerrain\_ac' column to print the values greater than 10000

ski\_data.SkiableTerrain\_ac[ski\_data.SkiableTerrain\_ac > 10000]

**Q: 2** One resort has an incredibly large skiable terrain area! Which is it?



#Code task 20#

#Now you know there's only one, print the whole row to investigate all values, including seeing the resort name

#Hint: don't forget the transpose will be helpful here

ski\_data[ski\_data.SkiableTerrain\_ac > 10000].T

**A: 2** Silverton Mountain resort has an incredibly large skiable terrain area.

But what can you do when you have one record that seems highly suspicious?

You can see if your data are correct. Search for "silverton mountain skiable area". If you do this, you get some [useful information](https://www.google.com/search?q=silverton+mountain+skiable+area).

You can spot check data. You see your top and base elevation values agree, but the skiable area is very different. Your suspect value is 26819, but the value you've just looked up is 1819. The last three digits agree. This sort of error could have occured in transmission or some editing or transcription stage. You could plausibly replace the suspect value with the one you've just obtained. Another cautionary note to make here is that although you're doing this in order to progress with your analysis, this is most definitely an issue that should have been raised and fed back to the client or data originator as a query. You should view this "data correction" step as a means to continue (documenting it carefully as you do in this notebook) rather than an ultimate decision as to what is correct.



#Code task 21#

#Use the .loc accessor to print the 'SkiableTerrain\_ac' value only for this resort

ski\_data.loc[39, 'SkiableTerrain\_ac']



#Code task 22#

#Use the .loc accessor again to modify this value with the correct value of 1819

ski\_data.loc[39, 'SkiableTerrain\_ac'] = 1819



#Code task 23#

#Use the .loc accessor a final time to verify that the value has been modified

ski\_data.loc[39, 'SkiableTerrain\_ac']

**NB whilst you may become suspicious about your data quality, and you know you have missing values, you will not here dive down the rabbit hole of checking all values or web scraping to replace missing values.**

What does the distribution of skiable area look like now?



ski\_data.SkiableTerrain\_ac.hist(bins=30)

plt.xlabel('SkiableTerrain\_ac')

plt.ylabel('Count')

plt.title('Distribution of skiable area (acres) after replacing erroneous value');

You now see a rather long tailed distribution. You may wonder about the now most extreme value that is above 8000, but similarly you may also wonder about the value around 7000. If you wanted to spend more time manually checking values you could, but leave this for now. The above distribution is plausible.

**2.6.4.2.2 Snow Making\_ac**



ski\_data['Snow Making\_ac'][ski\_data['Snow Making\_ac'] > 1000]



ski\_data[ski\_data['Snow Making\_ac'] > 3000].T

You can adopt a similar approach as for the suspect skiable area value and do some spot checking. To save time, here is a link to the website for [Heavenly Mountain Resort](https://www.skiheavenly.com/the-mountain/about-the-mountain/mountain-info.aspx). From this you can glean that you have values for skiable terrain that agree. Furthermore, you can read that snowmaking covers 60% of the trails.

What, then, is your rough guess for the area covered by snowmaking?



.6 \* 4800

This is less than the value of 3379 in your data so you may have a judgement call to make. However, notice something else. You have no ticket pricing information at all for this resort. Any further effort spent worrying about values for this resort will be wasted. You'll simply be dropping the entire row!

**2.6.4.2.3 fastEight**

Look at the different fastEight values more closely:



ski\_data.fastEight.value\_counts()

Drop the fastEight column in its entirety; half the values are missing and all but the others are the value zero. There is essentially no information in this column.



#Code task 24#

#Drop the 'fastEight' column from ski\_data. Use inplace=True

ski\_data.drop(columns='fastEight', inplace=True)

What about yearsOpen? How many resorts have purportedly been open for more than 100 years?



#Code task 25#

#Filter the 'yearsOpen' column for values greater than 100

ski\_data.yearsOpen[ski\_data.yearsOpen > 100]

Okay, one seems to have been open for 104 years. But beyond that, one is down as having been open for 2019 years. This is wrong! What shall you do about this?

What does the distribution of yearsOpen look like if you exclude just the obviously wrong one?



#Code task 26#

#Call the hist method on 'yearsOpen' after filtering for values under 1000

#Pass the argument bins=30 to hist(), but feel free to explore other values

ski\_data.yearsOpen[ski\_data.yearsOpen < 1000].hist(bins=30)

plt.xlabel('Years open')

plt.ylabel('Count')

plt.title('Distribution of years open excluding 2019');

The above distribution of years seems entirely plausible, including the 104 year value. You can certainly state that no resort will have been open for 2019 years! It likely means the resort opened in 2019. It could also mean the resort is due to open in 2019. You don't know when these data were gathered!

Let's review the summary statistics for the years under 1000.



ski\_data.yearsOpen[ski\_data.yearsOpen < 1000].describe()

The smallest number of years open otherwise is 6. You can't be sure whether this resort in question has been open zero years or one year and even whether the numbers are projections or actual. In any case, you would be adding a new youngest resort so it feels best to simply drop this row.



ski\_data = ski\_data[ski\_data.yearsOpen < 1000]

**2.6.4.2.4 fastSixes and Trams**

The other features you had mild concern over, you will not investigate further. Perhaps take some care when using these features.

**2.7 Derive State-wide Summary Statistics For Our Market Segment**

You have, by this point removed one row, but it was for a resort that may not have opened yet, or perhaps in its first season. Using your business knowledge, you know that state-wide supply and demand of certain skiing resources may well factor into pricing strategies. Does a resort dominate the available night skiing in a state? Or does it account for a large proportion of the total skiable terrain or days open?

If you want to add any features to your data that captures the state-wide market size, you should do this now, before dropping any more rows. In the next section, you'll drop rows with missing price information. Although you don't know what those resorts charge for their tickets, you do know the resorts exists and have been open for at least six years. Thus, you'll now calculate some state-wide summary statistics for later use.

Many features in your data pertain to chairlifts, that is for getting people around each resort. These aren't relevant, nor are the features relating to altitudes. Features that you may be interested in are:

* TerrainParks
* SkiableTerrain\_ac
* daysOpenLastYear
* NightSkiing\_ac

When you think about it, these are features it makes sense to sum: the total number of terrain parks, the total skiable area, the total number of days open, and the total area available for night skiing. You might consider the total number of ski runs, but understand that the skiable area is more informative than just a number of runs.

A fairly new groupby behaviour is [named aggregation](https://pandas-docs.github.io/pandas-docs-travis/whatsnew/v0.25.0.html). This allows us to clearly perform the aggregations you want whilst also creating informative output column names.



#Code task 27#

#Add named aggregations for the sum of 'daysOpenLastYear', 'TerrainParks', and 'NightSkiing\_ac'

#call them 'state\_total\_days\_open', 'state\_total\_terrain\_parks', and 'state\_total\_nightskiing\_ac',

#respectively

#Finally, add a call to the reset\_index() method (we recommend you experiment with and without this to see

#what it does)

state\_summary = ski\_data.groupby('state').agg(

resorts\_per\_state=pd.NamedAgg(column='Name', aggfunc='size'), #could pick any column here

state\_total\_skiable\_area\_ac=pd.NamedAgg(column='SkiableTerrain\_ac', aggfunc='sum'),

state\_total\_days\_open=pd.NamedAgg(column='daysOpenLastYear', aggfunc='sum'),

state\_total\_terrain\_parks=pd.NamedAgg(column='TerrainParks', aggfunc='sum'),

state\_total\_nightskiing\_ac=pd.NamedAgg(column='NightSkiing\_ac', aggfunc='sum')

).reset\_index()

state\_summary.head()

**2.8 Drop Rows With No Price Data**

You know there are two columns that refer to price: 'AdultWeekend' and 'AdultWeekday'. You can calculate the number of price values missing per row. This will obviously have to be either 0, 1, or 2, where 0 denotes no price values are missing and 2 denotes that both are missing.



missing\_price = ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum(axis=1)

missing\_price.value\_counts()/len(missing\_price) \* 100

About 14% of the rows have no price data. As the price is your target, these rows are of no use. Time to lose them.



#Code task 28#

#Use `missing\_price` to remove rows from ski\_data where both price values are missing

ski\_data = ski\_data[missing\_price!= 2]

**2.9 Review distributions**



ski\_data.hist(figsize=(15, 10))

plt.subplots\_adjust(hspace=0.5);

These distributions are much better. There are clearly some skewed distributions, so keep an eye on fastQuads, fastSixes, and perhaps trams. These lack much variance away from 0 and may have a small number of relatively extreme values. Models failing to rate a feature as important when domain knowledge tells you it should be is an issue to look out for, as is a model being overly influenced by some extreme values. If you build a good machine learning pipeline, hopefully it will be robust to such issues, but you may also wish to consider nonlinear transformations of features.

**2.10 Population data**

Population and area data for the US states can be obtained from [wikipedia](https://simple.wikipedia.org/wiki/List_of_U.S._states). Listen, you should have a healthy concern about using data you "found on the Internet". Make sure it comes from a reputable source. This table of data is useful because it allows you to easily pull and incorporate an external data set. It also allows you to proceed with an analysis that includes state sizes and populations for your 'first cut' model. Be explicit about your source (we documented it here in this workflow) and ensure it is open to inspection. All steps are subject to review, and it may be that a client has a specific source of data they trust that you should use to rerun the analysis.



#Code task 29#

#Use pandas' `read\_html` method to read the table from the URL below

states\_url = 'https://simple.wikipedia.org/w/index.php?title=List\_of\_U.S.\_states&oldid=7168473'

usa\_states = pd.read\_html(states\_url)



type(usa\_states)



len(usa\_states)



usa\_states = usa\_states[0]

usa\_states.head()

Note, in even the last year, the capability of pd.read\_html() has improved. The merged cells you see in the web table are now handled much more conveniently, with 'Phoenix' now being duplicated so the subsequent columns remain aligned. But check this anyway. If you extract the established date column, you should just get dates. Recall previously you used the .loc accessor, because you were using labels. Now you want to refer to a column by its index position and so use .iloc. For a discussion on the difference use cases of .loc and .iloc refer to the [pandas documentation](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html).



#Code task 30#

#Use the iloc accessor to get the pandas Series for column number 4 from `usa\_states`

#It should be a column of dates

established = usa\_states.iloc[:, 4]



established

Extract the state name, population, and total area (square miles) columns.



#Code task 31#

#Now use the iloc accessor again to extract columns 0, 5, and 6 and the dataframe's `copy()` method

#Set the names of these extracted columns to 'state', 'state\_population', and 'state\_area\_sq\_miles',

#respectively.

usa\_states\_sub = usa\_states.iloc[:, [0,5,6]].copy()

usa\_states\_sub.columns = ['state', 'state\_population', 'state\_area\_sq\_miles']

usa\_states\_sub.head()

Do you have all the ski data states accounted for?



#Code task 32#

#Find the states in `state\_summary` that are not in `usa\_states\_sub`

#Hint: set(list1) - set(list2) is an easy way to get items in list1 that are not in list2

missing\_states = set(state\_summary.state) - set(usa\_states\_sub.state)

missing\_states

No??

If you look at the table on the web, you can perhaps start to guess what the problem is. You can confirm your suspicion by pulling out state names that *contain* 'Massachusetts', 'Pennsylvania', or 'Virginia' from usa\_states\_sub:



usa\_states\_sub.state[usa\_states\_sub.state.str.contains('Massachusetts|Pennsylvania|Rhode Island|Virginia')]

Delete square brackets and their contents and try again:



#Code task 33#

#Use pandas' Series' `replace()` method to replace anything within square brackets (including the brackets)

#with the empty string. Do this inplace, so you need to specify the arguments:

#to\_replace='\[.\*\]' #literal square bracket followed by anything or nothing followed by literal closing bracket

#value='' #empty string as replacement

#regex=True #we used a regex in our `to\_replace` argument

#inplace=True #Do this "in place"

usa\_states\_sub.state.replace(to\_replace='\[.\*\]', value='', regex=True, inplace=True)

usa\_states\_sub.state[usa\_states\_sub.state.str.contains('Massachusetts|Pennsylvania|Rhode Island|Virginia')]



#Code task 34#

#And now verify none of our states are missing by checking that there are no states in

#state\_summary that are not in usa\_states\_sub (as earlier using `set()`)

missing\_states = set(state\_summary.state) - set(usa\_states\_sub.state)

missing\_states

Better! You have an empty set for missing states now. You can confidently add the population and state area columns to the ski resort data.



#Code task 35#

#Use 'state\_summary's `merge()` method to combine our new data in 'usa\_states\_sub'

#specify the arguments how='left' and on='state'

state\_summary = state\_summary.merge(usa\_states\_sub, how='left', on='state')

state\_summary.head()

Having created this data frame of summary statistics for various states, it would seem obvious to join this with the ski resort data to augment it with this additional data. You will do this, but not now. In the next notebook you will be exploring the data, including the relationships between the states. For that you want a separate row for each state, as you have here, and joining the data this soon means you'd need to separate and eliminate redundances in the state data when you wanted it.

**2.11 Target Feature**

Finally, what will your target be when modelling ticket price? What relationship is there between weekday and weekend prices?



#Code task 36#

#Use ski\_data's `plot()` method to create a scatterplot (kind='scatter') with 'AdultWeekday' on the x-axis and

#'AdultWeekend' on the y-axis

ski\_data.plot(x='AdultWeekday', y='AdultWeekend', kind='scatter');

A couple of observations can be made. Firstly, there is a clear line where weekend and weekday prices are equal. Weekend prices being higher than weekday prices seem restricted to sub $100 resorts. Recall from the boxplot earlier that the distribution for weekday and weekend prices in Montana seemed equal. Is this confirmed in the actual data for each resort? Big Mountain resort is in Montana, so the relationship between these quantities in this state are particularly relevant.



#Code task 37#

#Use the loc accessor on ski\_data to print the 'AdultWeekend' and 'AdultWeekday' columns for Montana only

ski\_data.loc[ski\_data.state == 'Montana', ['AdultWeekend','AdultWeekday']]

Is there any reason to prefer weekend or weekday prices? Which is missing the least?



ski\_data[['AdultWeekend', 'AdultWeekday']].isnull().sum()

Weekend prices have the least missing values of the two, so drop the weekday prices and then keep just the rows that have weekend price.



ski\_data.drop(columns='AdultWeekday', inplace=True)

ski\_data.dropna(subset=['AdultWeekend'], inplace=True)



ski\_data.shape

Perform a final quick check on the data.

**2.11.1 Number Of Missing Values By Row - Resort**

Having dropped rows missing the desired target ticket price, what degree of missingness do you have for the remaining rows?



missing = pd.concat([ski\_data.isnull().sum(axis=1), 100 \* ski\_data.isnull().mean(axis=1)], axis=1)

missing.columns=['count', '%']

missing.sort\_values(by='count', ascending=False).head(10)

These seem possibly curiously quantized...



missing['%'].unique()

Yes, the percentage of missing values per row appear in multiples of 4.



missing['%'].value\_counts()

This is almost as if values have been removed artificially... Nevertheless, what you don't know is how useful the missing features are in predicting ticket price. You shouldn't just drop rows that are missing several useless features.



ski\_data.info()

There are still some missing values, and it's good to be aware of this, but leave them as is for now.

**2.12 Save data**



ski\_data.shape

Save this to your data directory, separately. Note that you were provided with the data in raw\_data and you should saving derived data in a separate location. This guards against overwriting our original data.



# save the data to a new csv file

datapath = '../data'

save\_file(ski\_data, 'ski\_data\_cleaned.csv', datapath)



# save the state\_summary separately.

datapath = '../data'

save\_file(state\_summary, 'state\_summary.csv', datapath)

**2.13 Summary**

**Q: 3** Write a summary statement that highlights the key processes and findings from this notebook. This should include information such as the original number of rows in the data, whether our own resort was actually present etc. What columns, if any, have been removed? Any rows? Summarise the reasons why. Were any other issues found? What remedial actions did you take? State where you are in the project. Can you confirm what the target feature is for your desire to predict ticket price? How many rows were left in the data? Hint: this is a great opportunity to reread your notebook, check all cells have been executed in order and from a "blank slate" (restarting the kernel will do this), and that your workflow makes sense and follows a logical pattern. As you do this you can pull out salient information for inclusion in this summary. Thus, this section will provide an important overview of "what" and "why" without having to dive into the "how" or any unproductive or inconclusive steps along the way.

**A: 3** Your answer here