We've come a long way in this course and learned:

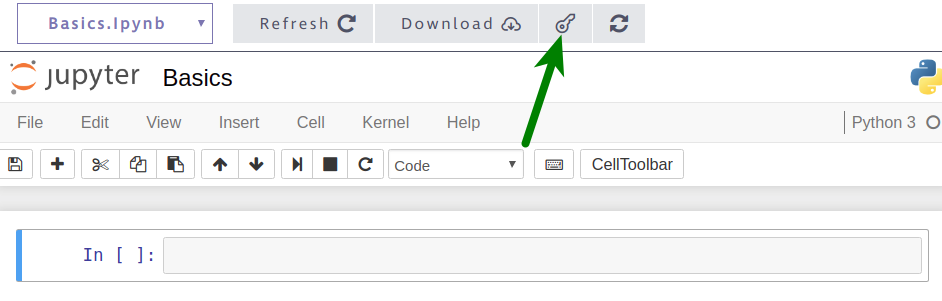
* The basics of programming in Python (arithmetical operations, variables, common data types, etc.)
* List and for loops
* Conditional statements
* Dictionaries and frequency tables
* Functions
* Jupyter Notebook

To make learning smoother and more efficient, we learned about each of these topics in isolation. In this **guided project**, we'll learn to combine these skills to perform practical data analysis. Guided projects are more involved compared to regular missions, so you should expect to spend more time working on them. After completing your work, you should be able to add a project like [this](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb) to your portfolio.

For this project, we'll pretend we're working as data analysts for a company that builds Android and iOS mobile apps. We make our apps available on Google Play and the App Store.

We only build apps that are free to download and install, and our main source of revenue consists of in-app ads. This means our revenue for any given app is mostly influenced by the number of users who use our app — the more users that see and engage with the ads, the better. Our goal for this project is to analyze data to help our developers understand what type of apps are likely to attract more users.

This is intended to be a portfolio project; you may want to share it with potential employers one day, so we'll start our work by helping readers understand what this project is about and what we're trying to achieve. You'll be able to check the solutions for this project in this [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb). The key icon at the top of our interface will also take you to the solution notebook.



To follow the instructions, you can use the Jupyter Notebook interface you see on the right of the screen.

Instructions

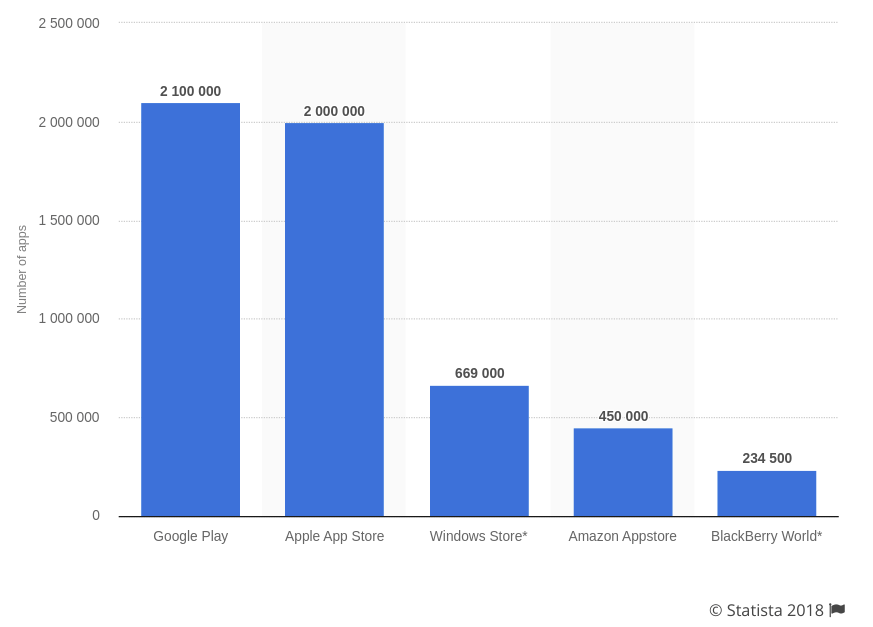
To help readers gain context into your project, use the first Markdown cell of the notebook to:

1. Add a title.
2. Write a short introduction where you explain (in no more than two paragraphs):
   * What the project is about.
   * What your goal is in this project.

The title and the introduction are tentative at this point, so don't spend too much time here — you can come back later to refine them.

In the previous step, we outlined that our aim is to help our developers understand what type of apps are likely to attract more users on Google Play and the App Store. To do this, we'll need to collect and analyze data about mobile apps available on Google Play and the App Store.

As of September 2018, there were approximately 2 million iOS apps available on the App Store, and 2.1 million Android apps on Google Play.

Source: [Statista](https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/)

Collecting data for over 4 million apps requires a significant amount of time and money, so we'll try to analyze a sample of the data instead. To avoid spending resources on collecting new data ourselves, we should first try to see if we can find any relevant existing data at no cost. Luckily, these are two data sets that seem suitable for our goals:

* [A data set](https://www.kaggle.com/lava18/google-play-store-apps) containing data about approximately 10,000 Android apps from Google Play; the data was collected in August 2018. You can download the data set directly from [this link](https://dq-content.s3.amazonaws.com/350/googleplaystore.csv).
* [A data set](https://www.kaggle.com/ramamet4/app-store-apple-data-set-10k-apps) containing data about approximately 7,000 iOS apps from the App Store; the data was collected in July 2017. You can download the data set directly from [this link](https://dq-content.s3.amazonaws.com/350/AppleStore.csv).

We'll start by opening and exploring these two data sets. To make them easier for you to explore, we created a function named explore\_data() that you can repeatedly use to print rows in a readable way.

def explore\_data(dataset, start, end, rows\_and\_columns=False):

   dataset\_slice = dataset[start:end]

   for row in dataset\_slice:

       print(row)

       print('\n') # adds a new (empty) line after each row

​

   if rows\_and\_columns:

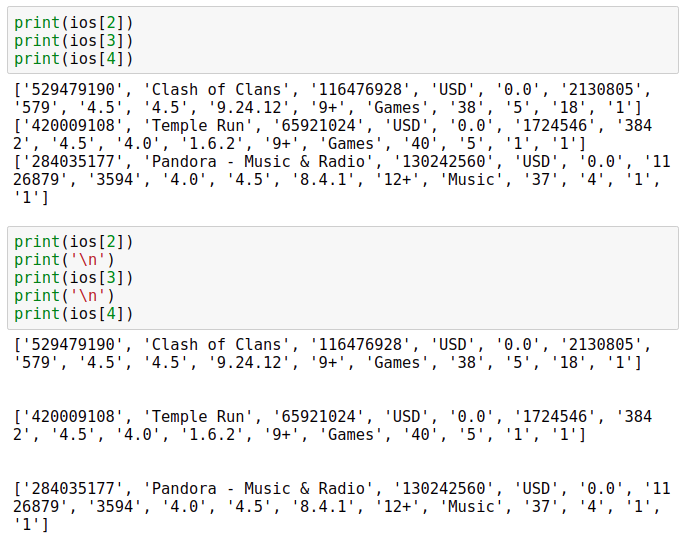
       print('Number of rows:', len(dataset))

       print('Number of columns:', len(dataset[0]))

The explore\_data() function:

* Takes in four parameters:
  + dataset, which is expected to be a list of lists.
  + start and end, which are both expected to be integers and represent the starting and the ending indices of a slice from the data set.
  + rows\_and\_columns, which is expected to be a Boolean and has False as a default argument.
* Slices the data set using dataset[start:end].
* Loops through the slice, and for each iteration, prints a row and adds a new line after that row using print('\n').
  + The \n in print('\n') is a special character and won't be printed. Instead, the \ncharacter adds a new line, and we use print('\n') to add some blank space between rows.
* Prints the number of rows and columns if rows\_and\_columns is True.
  + dataset shouldn't have a header row, otherwise the function will print the wrong number of rows (one more row compared to the actual length).

To help you better understand what print('\n') does, we printed three rows from the AppleStore.csv data set below. In the first code cell, we don't use print('\n') between rows, while in the second one we do:



Now let's open the two data sets and explore them.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Open the two data sets we mentioned above, and save both as lists of lists.
   * The App Store data set is stored in a CSV file named AppleStore.csv, and the Google Play data set is stored in a CSV file named googleplaystore.csv.
   * Both CSV files can be opened directly in the Jupyter Notebook interface you see on the right of the screen.
   * If you run into an error named UnicodeDecodeError, add encoding="utf8" to the open() function (for instance, use open('AppleStore.csv', encoding='utf8')).
2. Explore both data sets using the explore\_data() function.
   * Print the first few rows of each data set.
   * Find the number of rows and columns of each data set (recall that the function assumes the argument for the dataset parameter doesn't have a header row).
3. Print the column names and try to identify the columns that could help us with our analysis. Use the documentation of the data sets if you're having trouble understanding what a column describes. Add a link to the documentation for readers if you think the column names are not descriptive enough.

In the previous step, we opened the two data sets and performed a brief exploration of the data. Before beginning our analysis, we need to make sure the data we analyze is accurate, otherwise the results of our analysis will be wrong. This means that we need to:

* Detect inaccurate data, and correct or remove it.
* Detect duplicate data, and remove the duplicates.

Recall that at our company, we only build apps that are *free* to download and install, and that are directed toward an *English-speaking* audience. This means that we'll need to:

* Remove non-English apps like *爱奇艺PPS -《欢乐颂2》电视剧热播*.
* Remove apps that aren't free.

This process of preparing our data for analysis is called **data cleaning**. Data cleaning is done *before* the analysis; it includes removing or correcting wrong data, removing duplicate data, and modifying the data to fit the purpose of our analysis.

It's often said that data scientists spend around 80% of their time cleaning data, and only about 20% actually analyzing (cleaned) data. In this project, we'll see that this is not far from the truth.

Let's begin by detecting and deleting wrong data. For this guided project, we'll guide you throughout the entire data cleaning process. In later courses, we'll learn more about data cleaning, and you'll be able to perform data cleaning without any guidance.

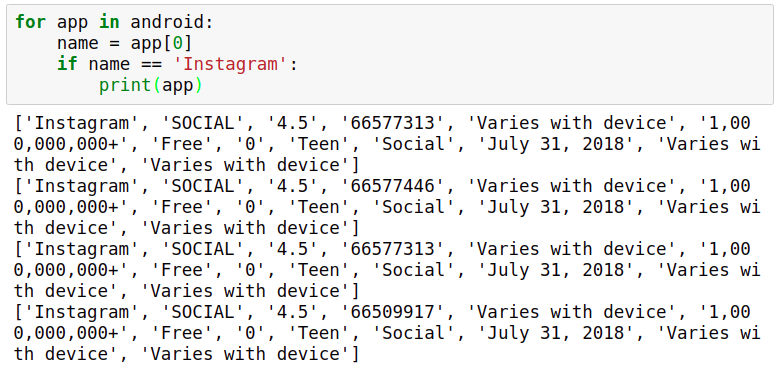
If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

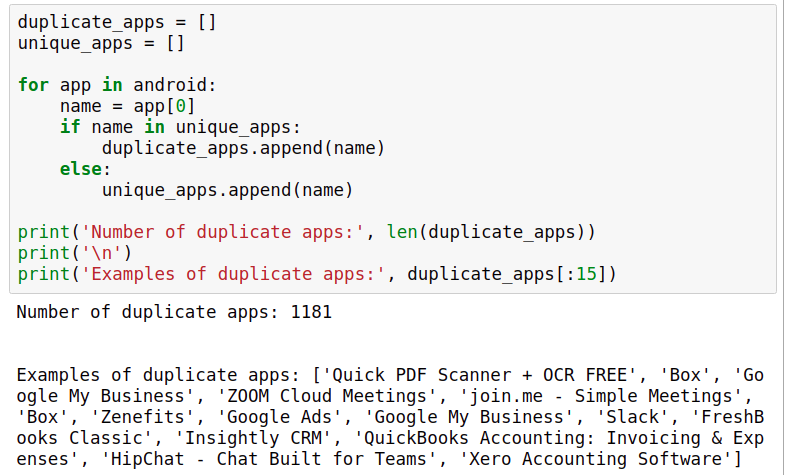
The Google Play data set has a dedicated [discussion section](https://www.kaggle.com/lava18/google-play-store-apps/discussion), and we can see that [one of the discussions](https://www.kaggle.com/lava18/google-play-store-apps/discussion/66015)describes an error for a certain row.

1. Read the discussion and find out what the index of the row is.
2. Print the row at that index to check whether it's indeed incorrect. Take into account the user reporting the error might or might have not removed the header row, so the index number might vary.
3. If the row has an error, remove the row using the [del statement](https://docs.python.org/3/reference/simple_stmts.html?highlight=del#the-del-statement). For instance, to remove the row with the index 149 from a data set data that is stored as a list of list, you can use the code del data[149].
4. Make sure you don't run the del statement more than once, otherwise you'll delete more than one row.
5. Read the [discussion section](https://www.kaggle.com/ramamet4/app-store-apple-data-set-10k-apps/discussion) for the App Store data set, and see whether you can find any reports of wrong data.

In the last step, we started the data cleaning process and deleted a row with incorrect data from the Google Play data set. If you explore the Google Play data set long enough or look at the [discussions](https://www.kaggle.com/lava18/google-play-store-apps/discussion) section, you'll notice some apps have duplicate entries. For instance, Instagram has four entries:



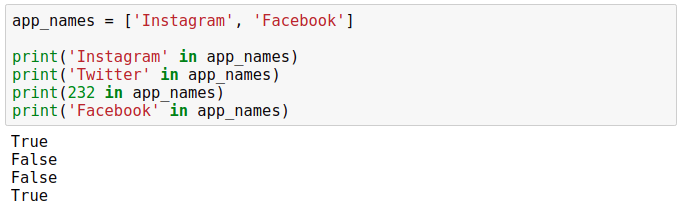
In total, there are 1,181 cases where an app occurs more than once:



Above, we:

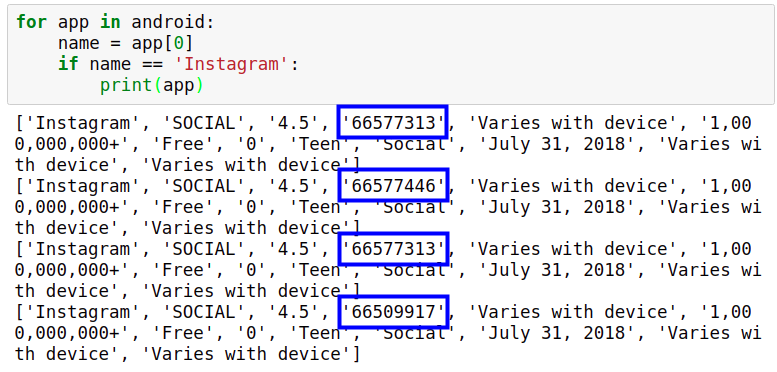
* Created two lists: one for storing the name of duplicate apps, and one for storing the name of unique apps.
* Looped through the android data set (the Google Play data set), and for each iteration:
  + We saved the app name to a variable named name.
  + If name was already in the unique\_apps list, we appended name to the duplicate\_appslist.
  + Else (if name wasn't already in the unique\_apps list), we appended name to the unique\_apps list.

(As a side note, you may notice we used the in operator above to check for membership in a list. We only learned to use in to check for membership in dictionaries, but in also works with lists):



We don't want to count certain apps more than once when we analyze data, so we need to remove the duplicate entries and keep only one entry per app. One thing we could do is remove the duplicate rows randomly, but we could probably find a better way.

If you examine the rows we printed for the Instagram app, the main difference happens on the fourth position of each row, which corresponds to the number of reviews. The different numbers show the data was collected at different times.



We can use this information to build a criterion for removing the duplicates. The higher the number of reviews, the more recent the data should be. Rather than removing duplicates randomly, we'll only keep the row with the highest number of reviews and remove the other entries for any given app.

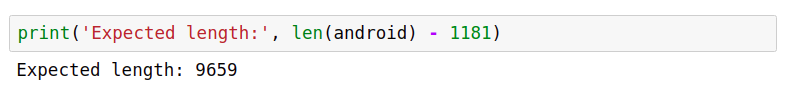
We'll remove the rows on the next screen. Now it's your turn to write some code and confirm the data has duplicate entries.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Using a combination of narrative and code, explain the reader that the Google Play data set has duplicate entries. Print a few duplicate rows to confirm.
2. Count the number of duplicates using the technique we learned above.
3. Explain that you won't remove the duplicates randomly. Describe the criterion you're going to use to remove the duplicates.
   * We already suggested a criterion above, but you can come up with another criterion if you want. Make sure you support your criterion with at least one argument.

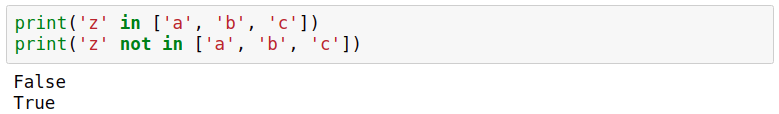
On the previous screen, we looped through the Google Play data set and found that there are 1,181 duplicates. After we remove the duplicates, we should be left with 9,659 rows:



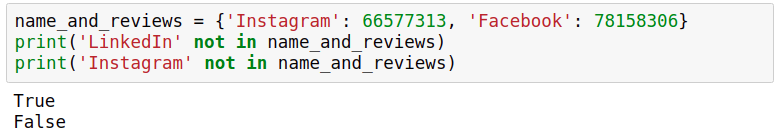
To remove the duplicates, we will:

* Create a dictionary, where each dictionary key is a unique app name and the corresponding dictionary value is the highest number of reviews of that app.
* Use the information stored in the dictionary and create a new data set, which will have only one entry per app (and for each app, we'll only select the entry with the highest number of reviews).

To turn the steps above into code, we'll need to use the [not in operator](https://docs.python.org/3/reference/expressions.html#not-in). The not in operator is the opposite of the in operator. For instance, 'z' in ['a', 'b', 'c'] returns False because 'z'is not in ['a', 'b', 'c'], but 'z' not in ['a', 'b', 'c'] returns True because it's true that 'z' is *not in* the list ['a', 'b', 'c'].



Essentially, we use both the in and not in operators to check for membership — we want to know whether a value belongs to some group of values or not. We can also use the not in operator to check for membership in a dictionary. Just like in the case of the in operator, the membership check is only done over the dictionary keys:



Now let's write the code to remove the duplicate entries.

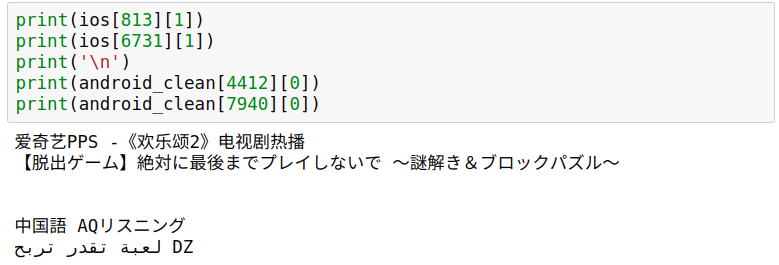
If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Create a dictionary where each key is a unique app name and the corresponding dictionary value is the highest number of reviews of that app.
   * Start by creating an empty dictionary named reviews\_max.
   * Loop through the Google Play data set (make sure you don't include the header row). For each iteration:
     + Assign the app name to a variable named name.
     + Convert the number of reviews to float. Assign it to a variable named n\_reviews.
     + If name already exists as a key in the reviews\_max dictionary **and** reviews\_max[name] < n\_reviews, update the number of reviews for that entry in the reviews\_max dictionary.
     + If name **is not in** the reviews\_max dictionary as a key, create a new entry in the dictionary where the key is the app name, and the value is the number of reviews. Make sure you don't use an else clause here, otherwise the number of reviews will be incorrectly updated whenever reviews\_max[name] < n\_reviews evaluates to False.
   * Inspect the dictionary to make sure everything went as expected. Measure the length of the dictionary — remember that the expected length is 9,659 entries.
2. Use the dictionary you created above to remove the duplicate rows:
   * Start by creating two empty lists: android\_clean (which will store our new cleaned data set) and already\_added (which will just store app names).
   * Loop through the Google Play data set (make sure you don't include the header row), and for each iteration:
     + Assign the app name to a variable named name.
     + Convert the number of reviews to float, and assign it to a variable named n\_reviews.
   * If n\_reviews is the same as the number of maximum reviews of the app name (the number can be found in the reviews\_max dictionary) **and** name is not already in the list already\_added (read the solution notebook to find out why we need this supplementary condition):
     + Append the entire row to the android\_clean list (which will eventually be a list of list and store our cleaned data set).
     + Append the name of the app name to the already\_added list — this helps us to keep track of apps that we already added.
3. Explore the android\_clean data set to ensure everything went as expected. The data set should have 9,659 rows. The two steps above are a bit more involved, so make sure you use Markdown to explain the readers the steps you took.

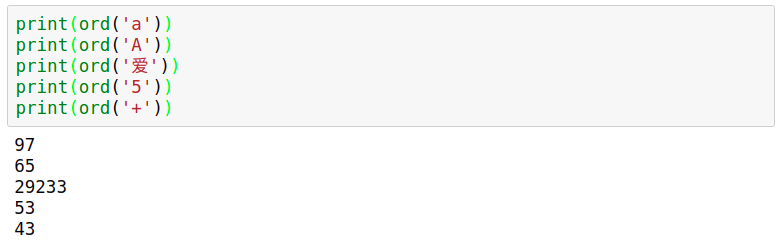
In the previous step, we managed to remove the duplicate app entries in the Google Play data set. We don't need to do the same for the App Store data because there are no duplicates — you can check that for yourself using the id column (not the track\_name column).

Remember we use English for the apps we develop at our company, and we'd like to analyze only the apps that are directed toward an English-speaking audience. However, if we explore the data long enough, we'll find that both data sets have apps with names that suggest they are not directed toward an English-speaking audience.



We're not interested in keeping these apps, so we'll remove them. One way to go about this is to remove each app with a name containing a symbol that is not commonly used in English text — English text usually includes letters from the English alphabet, numbers composed of digits from 0 to 9, punctuation marks (., !, ?, ;), and other symbols (+, \*, /).

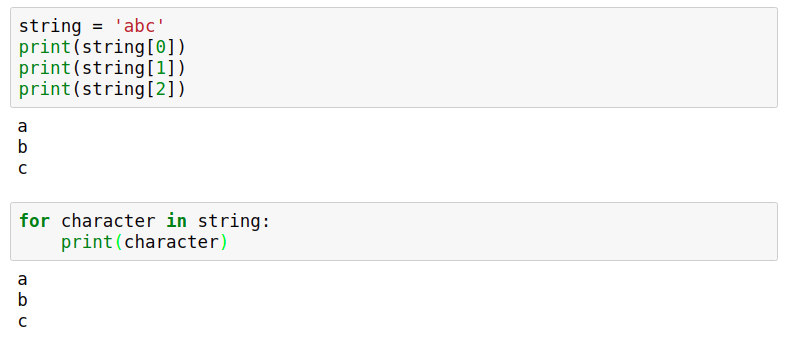
Behind the scenes, each character we use in a string has a corresponding number associated with it. For instance, the corresponding number for character 'a' is 97, character 'A' is 65, and character '爱'is 29,233. We can get the corresponding number of each character using the [ord() built-in function](https://docs.python.org/3/library/functions.html" \l "ord" \t "_blank).



The numbers corresponding to the characters we commonly use in an English text are all in the range 0 to 127, according to the [ASCII](https://en.wikipedia.org/wiki/ASCII) (American Standard Code for Information Interchange) system. Based on this number range, we can build a function that detects whether a character belongs to the set of common English characters or not. If the number is equal to or less than 127, then the character belongs to the set of common English characters.

If an app name contains a character that is greater than 127, then it probably means that the app has a non-English name. Our app names, however, are stored as strings, so how could we take each individual character of a string and check its corresponding number?

In Python, strings are indexable and iterable, which means we can use indexing to select an individual character, and we can also iterate on the string using a for loop.



Let's first try to write the function we talked about above, and in the next screen we'll remove the rows corresponding to the non-English apps.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Write a function that takes in a string and returns False if there's any character in the string that doesn't belong to the set of common English characters, otherwise it returns True.
   * Inside the function, iterate over the input string. For each iteration check whether the number associated with the character is greater than 127. When a character is greater than 127, the function should immediately return False — the app name is probably non-English since it contains a character that doesn't belong to the set of common English characters.
   * If the loop finishes running without the return statement being executed, then it means no character had a corresponding number over 127 — the app name is probably English, so the functions should return True.
2. Use your function to check whether these app names are detected as English or non-English:
   * 'Instagram'
   * '爱奇艺PPS -《欢乐颂2》电视剧热播'
   * 'Docs To Go™ Free Office Suite'
   * 'Instachat 😜'

On the previous screen, we wrote a function that detects non-English app names, but we saw that the function couldn't correctly identify certain English app names like 'Docs To Go™ Free Office Suite' and 'Instachat 😜'. This is because emojis and characters like ™ fall outside the ASCII range and have corresponding numbers over 127.



If we're going to use the function we've created, we'll lose useful data since many English apps will be incorrectly labeled as non-English. To minimize the impact of data loss, we'll only remove an app if its name has more than three characters with corresponding numbers falling outside the ASCII range. This means all English apps with up to three emoji or other special characters will still be labeled as English. Our filter function is still not perfect, but it should be fairly effective.

Let's edit the function we created in the previous screen, and then use it to filter out the non-English apps.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Change the function you created in the previous screen. If the input string has more than three characters that fall outside the ASCII range (0 - 127), then the function should return False (identify the string as non-English), otherwise it should return True.
2. Use the new function to check whether these app names are detected as English or non-English:
   * 'Docs To Go™ Free Office Suite'
   * 'Instachat 😜'
   * '爱奇艺PPS -《欢乐颂2》电视剧热播'
3. Use the new function to filter out non-English apps from both data sets. Loop through each data set. If an app name is identified as English, append the whole row to a separate list.
4. Explore the data sets and see how many rows you have remaining for each data set.

So far in the data cleaning process, we:

* Removed inaccurate data
* Removed duplicate app entries
* Removed non-English apps

As we mentioned in the introduction, we only build apps that are free to download and install, and our main source of revenue consists of in-app ads. Our data sets contain both free and non-free apps; we'll need to isolate only the free apps for our analysis.

Isolating the free apps will be our last step in the data cleaning process. On the next screen, we're going to start analyzing the data.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Loop through each data set to isolate the free apps in separate lists.
   * Make sure you identify the columns describing the app price correctly.
   * Prices come up as strings ('0', $0.99, $2.99, etc.), so make sure you're not checking an integer or a float in your conditional statements.
2. After you isolate the free apps, check the length of each data set to see how many apps you have remaining.

So far, we spent a good amount of time on cleaning data, and:

* Removed inaccurate data
* Removed duplicate app entries
* Removed non-English apps
* Isolated the free apps

As we mentioned in the introduction, our aim is to determine the kinds of apps that are likely to attract more users because our revenue is highly influenced by the number of people using our apps.

To minimize risks and overhead, our validation strategy for an app idea is comprised of three steps:

1. Build a minimal Android version of the app, and add it to Google Play.
2. If the app has a good response from users, we develop it further.
3. If the app is profitable after six months, we build an iOS version of the app and add it to the App Store.

Because our end goal is to add the app on both Google Play and the App Store, we need to find app profiles that are successful on both markets. For instance, a profile that works well for both markets might be a productivity app that makes use of gamification.

Let's begin the analysis by getting a sense of what are the most common genres for each market. For this, we'll need to build frequency tables for a few columns in our data sets.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Give readers more context into why we want to find an app profile that fits both the App Store and Google Play. Explain our validation strategy for an app idea.
2. Inspect both data sets and identify the columns you could use to generate frequency tables to find out what are the most common genres in each market.

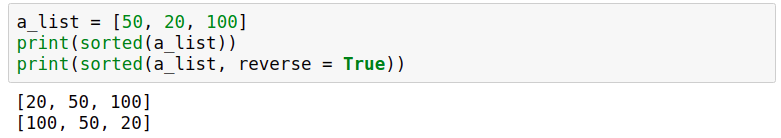
In the previous screen, we looked at our validation strategy for an app idea, and then we inspected the data sets to identify the columns that might be useful for finding out what the most common genres in each market are. Our conclusion was that we'll need to build a frequency table for the prime\_genrecolumn of the App Store data set, and for the Genres and Category columns of the Google Play data set.

We'll build two functions we can use to analyze the frequency tables:

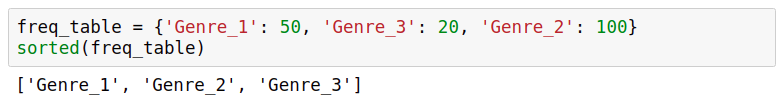
* One function to generate frequency tables that show percentages
* Another function we can use to display the percentages in a descending order

We already learned to generate frequency tables that show percentages, and we're going to build a function for that in the exercise below. However, dictionaries don't have order, and it will be very difficult to analyze the frequency tables. We'll need to build a second function which can help us display the entries in the frequency table in a descending order.

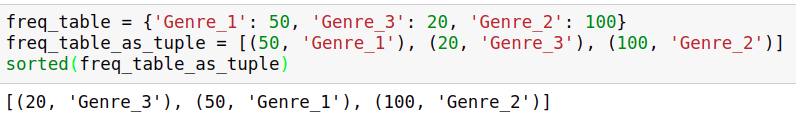
To do that, we'll need to make use of the built-in [sorted() function](https://docs.python.org/3/library/functions.html#sorted). This function takes in an iterable data type (like a list, dictionary, tuple, etc.), and returns a list of the elements of that iterable sorted in ascending or descending order (the reverse parameter controls whether the order is ascending or descending).



The sorted() function doesn't work too well with dictionaries because it only considers and returns the dictionary keys.



However, the sorted() function works well if we transform the dictionary into a list of tuples, where each tuple contains a dictionary key along with its corresponding dictionary value. To ensure the sorting works right, the dictionary value comes first, and the dictionary key comes second:



This is a bit overcomplicated to just sort a dictionary, but there are much simpler ways to do this once we learn more advanced techniques. Using the workaround above, we wrote a helper function for you named display\_table(), which you'll be able to combine with the function you're going to write in the next exercise. The display\_table() function you see below:

* Takes in two parameters: dataset and index. dataset is expected to be a list of lists, and index is expected to be an integer.
* Generates a frequency table using the freq\_table() function (which you're going to write as an exercise).
* Transforms the frequency table into a list of tuples, then sorts the list in a descending order.
* Prints the entries of the frequency table in descending order.

def display\_table(dataset, index):

   table = freq\_table(dataset, index)

   table\_display = []

   for key in table:

       key\_val\_as\_tuple = (table[key], key)

       table\_display.append(key\_val\_as\_tuple)

​

   table\_sorted = sorted(table\_display, reverse = True)

   for entry in table\_sorted:

       print(entry[1], ':', entry[0])

Let's now create a function for generating frequency tables, and use it in combination with the display\_table() function.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Create a function named freq\_table() that takes in two inputs: dataset (which is expected to be a list of lists) and index (which is expected to be an integer).
   * The function should return the frequency table (as a dictionary) for any column we want. The frequencies should also be expressed as percentages.
   * We already learned how to build frequency tables in [the mission on dictionaries](https://app.dataquest.io/m/314/dictionaries-and-frequency-tables/8/looping-over-dictionaries).
2. Copy the display\_table() function we wrote above. Use it to display the frequency table of the columns prime\_genre, Genres, and Category. We'll analyze the resulting tables on the next screen.

On the previous screen, we generated frequency tables for the columns prime\_genre, Genres, and Category. We'll now focus on analyzing these frequency tables.

Remember our data set only contains free English apps, so you should be careful not to extend your conclusions beyond that scope. If you find that gaming apps are the most numerous among the free English apps on Google Play, it doesn't mean we'll see the same pattern on Google Play as a whole.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Analyze the frequency table you generated for the prime\_genre column of the App Store data set.
   * What is the most common genre? What is the runner-up?
   * What other patterns do you see?
   * What is the general impression — are most of the apps designed for practical purposes (education, shopping, utilities, productivity, lifestyle) or more for entertainment (games, photo and video, social networking, sports, music)?
   * Can you recommend an app profile for the App Store market based on this frequency table alone? If there's a large number of apps for a particular genre, does that also imply that apps of that genre generally have a large number of users?
2. Analyze the frequency table you generated for the Category and Genres column of the Google Play data set.
   * What are the most common genres?
   * What other patterns do you see?
   * Compare the patterns you see for the Google Play market with those you saw for the App Store market.
   * Can you recommend an app profile based on what you found so far? Do the frequency tables you generated reveal the most frequent app genres or what genres have the most users?

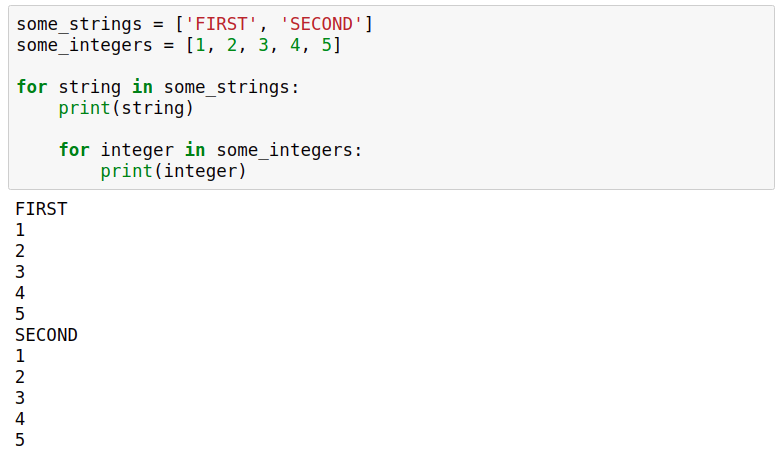
The frequency tables we analyzed on the previous screen showed us that the App Store is dominated by apps designed for fun, while Google Play shows a more balanced landscape of both practical and fun apps. Now, we'd like to get an idea about the kind of apps with the most users.

One way to find out what genres are the most popular (have the most users) is to calculate the average number of installs for each app genre. For the Google Play data set, we can find this information in the Installs column, but this information is missing for the App Store data set. As a workaround, we'll take the total number of user ratings as a proxy, which we can find in the rating\_count\_tot app.

Let's start with calculating the average number of user ratings per app genre on the App Store. To do that, we'll need to:

* Isolate the apps of each genre.
* Sum up the user ratings for the apps of that genre.
* Divide the sum by the number of apps belonging to that genre (not by the total number of apps).

To calculate the average number of user ratings for each genre, we'll use a for loop inside of another for loop. This is an example of a for loop used inside another for loop:



Above, we can see that:

* We first iterate over the some\_strings list, and for each iteration:
  + We print string (iteration variable).
  + We start another iteration over the list some\_integers.
    - For each iteration over this list, we print integer (iteration variable).

We can see that for each of the two iterations over the list some\_strings (there are two iterations because some\_strings only has two list elements), there's another inner iteration happening over the list some\_integers.

The second iteration over some\_strings begins only when the iteration over some\_integers is done completely. Notice that all the elements of the list some\_integers are printed for each of the two iterations over the list some\_strings.

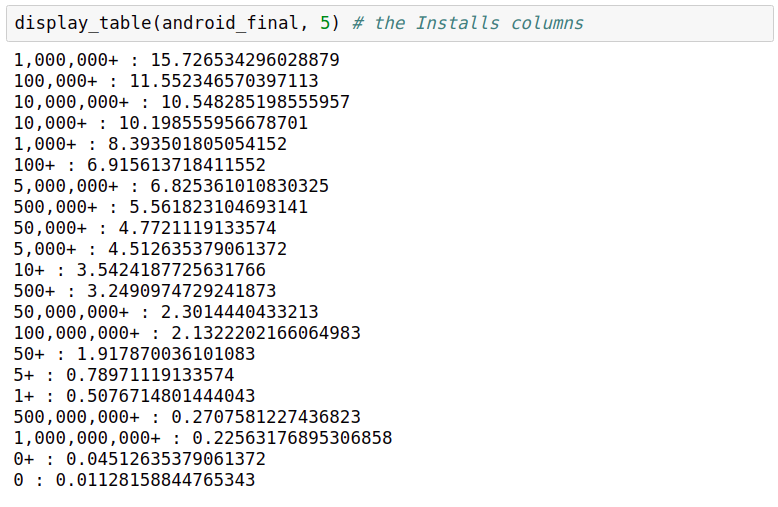
A loop inside another loop is called a **nested loop**. We'll use a nested loop to compute the averages we mentioned above.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Start by generating a frequency table for the prime\_genre column to get the unique app genres (below, we'll need to loop over the unique genres). You can use the freq\_table() function you wrote in a previous screen.
2. Loop over the unique genres of the App Store data set. For each iteration (below, we'll assume that the iteration variable is named genre):
   * Initiate a variable named total with a value of 0. This variable will store the sum of user ratings (the number of ratings, not the actual ratings) specific to each genre.
   * Initiate a variable named len\_genre with a value of 0. This variable will store the number of apps specific to each genre.
   * Loop over the App Store data set, and for each iteration:
     + Save the app genre to a variable named genre\_app.
     + If genre\_app is the same as genre (the iteration variable of the main loop), then:
       - Save the number of user ratings of the app as a float.
       - Add up the number of user ratings to the total variable.
       - Increment the len\_genre variable by 1.
   * Compute the average number of user ratings by dividing total by len\_genre. This should be done outside the nested loop.
   * Print the app genre and the average number of user ratings. This should also be done outside the nested loop.
3. Analyze the results and try to come up with at least one app profile recommendation for the App Store. Note that there's no fixed answer here, and it's perfectly fine if the app profile you recommended is different than the one recommended in the solution notebook.

In the previous screen, we came up with an app profile recommendation for the App Store based on the number of user ratings. We have data about the number of installs for the Google Play market, so we should be able to get a clearer picture about genre popularity. However, the install numbers don't seem precise enough — we can see that most values are open-ended (100+, 1,000+, 5,000+, etc.):



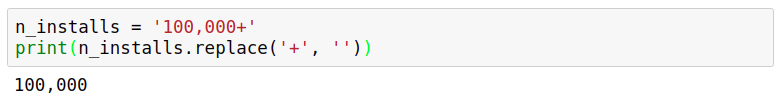
For instance, we don't know whether an app with 100,000+ installs has 100,000 installs, 200,000, or 350,000. However, we don't need very precise data for our purposes — we only want to find out which app genres attract the most users, and we don't need perfect precision with respect to the number of users.

We're going to leave the numbers as they are, which means that we'll consider that an app with 100,000+ installs has 100,000 installs, and an app with 1,000,000+ installs has 1,000,000 installs, and so on. To perform computations, however, we'll need to convert each install number from string to float. This means we need to remove the commas and the plus characters, otherwise the conversion will fail and raise an error.

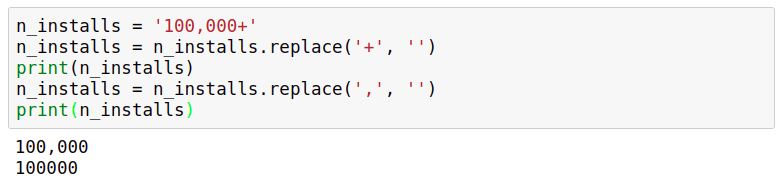
To remove characters from strings, we can use [str.replace(old, new) method](https://docs.python.org/3/library/stdtypes.html?" \l "str.replace" \t "_blank) (just like list.append() or list.copy(), str.replace() is a special kind of function called method — we'll learn more about this early in the next course). str.replace() takes in two parameters, oldand new, and replaces all occurrences of old within a string with new:



To remove certain characters, we can replace them with the empty string '':



Note that we'll need to reassign to n\_installs if we want our changes saved:



Now let's calculate the average number of installs per app genre for the Google Play data set. We'll need to use a nested loop, just like in the previous screen.

If you get stuck during the following exercise, you can check the [solution notebook](https://github.com/dataquestio/solutions/blob/master/Mission350Solutions.ipynb).

Instructions

1. Start by generating a frequency table for the Category column of the Google Play data set to get the unique app genres (below, we'll need to loop over the unique genres). You can use the freq\_table() function you wrote in a previous screen.
2. Loop over the unique genres of the Google Play data set. For each iteration (below, we'll assume that the iteration variable is named category):
   * Initiate a variable named total with a value of 0. This variable will store the sum of installs specific to each genre.
   * Initiate a variable named len\_category with a value of 0. This variable will store the number of apps specific to each genre.
   * Loop over the Google Play data set, and for each iteration:
     + Save the app genre to a variable named category\_app.
     + If category\_app is the same as category (the iteration variable of the main loop), then:
       - Save the number of installs.
       - Remove any + or , character, and then convert the string to a float.
       - Add up the number of installs to the total variable.
       - Increment the len\_category variable by 1.
   * Compute the average number of installs by dividing total by len\_category. This should be done outside the nested loop.
   * Print the app genre and the average number of installs. This should also be done outside the nested loop.
3. Analyze the results and try to come up with at least one app profile recommendation for Google Play. Remember, our aim is to recommend an app genre that shows potential for being profitable on both the App Store and Google Play. Note that there's no fixed answer here, and it's perfectly fine if the app profile you recommended is different than the one recommended in the solution notebook.

In this project, we went through a complete data science workflow:

* We started by clarifying the goal of our project.
* We collected relevant data.
* We cleaned the data to prepare it for analysis.
* We analyzed the cleaned data.

In the solution notebook, we concluded that taking a very popular book (perhaps a more recent book) and turning it into an app could be profitable for both the Google Play and the App Store market. The markets are already full of libraries, so we need to add some special features besides the raw version of the book. This might include daily quotes from the book, an audio version of the book, quizzes on the book, a forum where people can discuss the book, etc. You might have reached a different conclusion, which is perfectly fine as long as you managed to build a data-driven argumentation for your recommendation.

Curious to see what other students have done on this project? [Head over to our Community to check them out](https://community.dataquest.io/tags/c/social/share/49/350). While you are there, please remember to show some love and give your own feedback!

And of course, we welcome you to share your own project and show off your hard work. Head over to our Community to [share your finished Guided Project](https://community.dataquest.io/tags/c/social/share/49/350)!

These are a few next steps you could take:

* Analyze the frequency table for the Genre column of the Google Play data set, and see whether you can find useful patterns.
* Assume we could also make revenue via in-app purchases and subscriptions, and try to find out which genres seem to be liked the most by users — you could examine app ratings here.
* Refine your project using our data science project [style guide](https://www.dataquest.io/blog/data-science-project-style-guide/).

If you're going to work on the next steps above independently, you'll almost inevitably face some problems like not knowing how to fix an error, or not knowing what code to write to perform a certain task. In situations like these, the best thing to do is to start with a Google search (or any other search engine). In most situations, there will always be people who already ran into the same kind of problem, and you'll be able to piggyback on the solution they came up with.

As you search for solutions to your problems, you'll notice that one particular site will constantly show up in the first few results of your query —  [Stack Overflow](https://stackoverflow.com/). The community on Stack Overflow is very active, and the answers you'll find there are almost always accurate and up-to-date. One important tip when you're searching on Google is to start with the word "python". For instance, if you want to find out how to remove the characters from a string, search for "**python** how to remove a character from a string" (not just "how to remove a character from a string") — otherwise you'll most likely get results for other programming languages.

Congratulations, this is the end of the course! We'll continue our data science journey in the next course, where we'll keep focusing on Python and learn about object-oriented programming, dates and times, and many other concepts that are essential for a data scientist.