Introduction

Pokemon is a giant video game franchise owned by Nintendo based around building a team of monsters called pokemon. Over the years, the collection of monsters have grown, and each new game they release gives us pokemon fans a new generation of unique monsters to battle with. However, it seems that with each new pokemon release, the Nintendo team is making the new generation of pokemon more powerful than previous ones, disappointing long time fans of the series since their older, favorite pokemon are being outshined by the new ones. In this paper I'm going to look into how Nintendo has changed the stats of the pokemon over the different generations to find out whether or not old pokemon are rendered useless compared to the new ones.

Scraping for data

To start off my code, I figured it would be the easiest to include all of the imports that I thought might come in handy for manipulating and visualizing the data. The first dataframe I created was the pokedex, which is basically Pokemon's list of all the different pokemon and their combat statistics.

In [882]:
```
import pandas as pd
import numpy as np
import requests
import re
import sqlite3 as sql
from bs4 import BeautifulSoup
# Grabs the table from the html page,
# Creating a dataframe from the <table> element
r = requests.get("https://pokemondb.net/pokedex/all")
soup = BeautifulSoup(r.content, "html.parser")\
    .find("table")\
    .prettify()
pokedex = pd.read_html(soup)[0]

pokedex
```

Out[882]:

	#	Name	Type	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed
0	1	Bulbasaur	Grass Poison	318	45	49	49	65	65	45
1	2	Ivysaur	Grass Poison	405	60	62	63	80	80	60
2	3	Venusaur	Grass Poison	525	80	82	83	100	100	80
3	3	Venusaur Mega Venusaur	Grass Poison	625	80	100	123	122	120	80
4	4	Charmander	Fire	309	39	52	43	60	50	65
...	...	...	...	...	...	...	...	...	...	...
1185	1004	Chi-Yu	Dark Fire	570	55	80	80	135	120	100
1186	1005	Roaring Moon	Dragon Dark	590	105	139	71	55	101	119
1187	1006	Iron Valiant	Fairy Fighting	590	74	130	90	120	60	116
1188	1007	Koraidon	Fighting Dragon	670	100	135	115	85	100	135
1189	1008	Miraidon	Electric Dragon	670	100	85	100	135	115	135

1190 rows × 10 columns

Tidying up Pokedex Dataframe

Since the pokedex contains some duplicates of the same pokemon with new concepts being introduced to the game such as mega evolutions and regional types, I found it best to get rid of all of the duplicate special versions of the pokemon and just use the base ones. This is because it might get confusing later down the line when we have to compare the generations to each other. I also found that it would be useful to separate the types into two categories instead of just one single one, because some pokemon have multiple types and it is a big factor in how each pokemon performs vs. others. I also added a generation column so that when we compare and contrast the different generations, it will be easier to determine which pokemon belongs to which generation.

In [883]:
```
# New columns to store specific type information
pokedex['Type1'] = ''
pokedex['Type2'] = ''
# Converts the single type column into type1 and type2 columns
# If pokemon only has one type, then set type2 to None
for index, row in pokedex.iterrows():
    types = row['Type'].split(" ")
    if(len(types) == 1):
        pokedex.at[index, 'Type1'] = types[0]
        pokedex.at[index, 'Type2'] = None
    else:
        pokedex.at[index, 'Type1'] = types[0]
        pokedex.at[index, 'Type2'] = types[2]

# Removes the original single type column from the dataframe
pokedex = pokedex.drop(pokedex.columns[2], axis=1)
# Gets rid of duplicate pokemon that have a regional form or mega evolution
# This is so we can only compare the new pokemon of each generation, and not remakes
# Since this could lead to some confusion during comparisons
pokedex.drop_duplicates(subset='#', keep='first', inplace=True)
pokedex = pokedex.reset_index(drop=True)
# Adds a generation column indicating which generation the pokemon is from
# There wasn't an easier/more accurate way to do this with online databases,
# so I simply had to manually seperate the pokemon by their generations
generations = []
for i in range(0,1008):
     if(i < 151):
          generations.append(1)
     elif(i < 251):
          generations.append(2)
     elif(i < 386):
          generations.append(3)
     elif(i < 493):
          generations.append(4)
     elif(i < 649):
          generations.append(5)
     elif(i < 721):
          generations.append(6)
     elif(i < 809):
          generations.append(7)
     elif(i < 905):
          generations.append(8)
     else:
          generations.append(9)
pokedex['Generation'] = generations
pokedex
```

Out[883]:

	#	Name	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Type1	Type2	Generation
0	1	Bulbasaur	318	45	49	49	65	65	45	Grass	Poison	1
1	2	Ivysaur	405	60	62	63	80	80	60	Grass	Poison	1
2	3	Venusaur	525	80	82	83	100	100	80	Grass	Poison	1
3	4	Charmander	309	39	52	43	60	50	65	Fire	None	1
4	5	Charmeleon	405	58	64	58	80	65	80	Fire	None	1
...	...	...	...	...	...	...	...	...	...	...	...	...
1003	1004	Chi-Yu	570	55	80	80	135	120	100	Dark	Fire	9
1004	1005	Roaring Moon	590	105	139	71	55	101	119	Dragon	Dark	9
1005	1006	Iron Valiant	590	74	130	90	120	60	116	Fairy	Fighting	9
1006	1007	Koraidon	670	100	135	115	85	100	135	Fighting	Dragon	9
1007	1008	Miraidon	670	100	85	100	135	115	135	Electric	Dragon	9

1008 rows × 12 columns

Creating/Tidying up Type Matchup Dataframe

As mentioned earlier, the "type" of a pokemon is very influential in determining its strength against its opponents. The types of pokemon usually represent some sort of mythical element in the pokemon world, and some pokemon only have one type while others have two. This dataframe is organized with the attacking pokemon type as the rows, and the defending pokemon type as the columns. For each row-column pair, there is a corresponding damage multiplier which represents how much to multiply the attack against the defending pokemon. I then took the average of each attack multiplier for each attacking type against each defending type, and ranked the types in the data frame accordingly.

In [884]:
```
# Here, we are creating a dataframe for the type matchups in pokemon
r = requests.get("https://pokemondb.net/type")
soup = BeautifulSoup(r.content, "html.parser")\
    .find("table")\
    .prettify()
# Unfilitered dataframe storing our type matchups
type_matchups = pd.read_html(soup)[0]
# Replace all NaN values with 1, for the damage/defense multiplier would be 1
type_matchups = type_matchups.replace('½', 0.5)
type_matchups = type_matchups.replace(np.nan, 1)
# Changes the names of the columns
type_names = ['Normal', 'Fire', 'Water', 'Electric', 'Grass', 'Ice', 'Fighting', 'Poison', 'Ground', 'Flying',\
    'Psychic', 'Bug', 'Rock', 'Ghost', 'Dragon', 'Dark', 'Steel', 'Fairy']
names = ['Attacking Type (Rows) vs Defending Type (Cols)'] + type_names
type_matchups.set_axis(names, axis=1,inplace=True)
# Converts all multipliers to floats
type_matchups[type_names] = type_matchups[type_names].astype('float64')
type_matchups['Mean Attack Multiplier'] = type_matchups[type_names].mean(axis=1)
# Ranks each type by the mean attack multiplier
type_matchups = type_matchups.sort_values('Mean Attack Multiplier', ascending=False)
type_matchups
```

```
/var/folders/t_/fw_ctfbj3pl01ch4zw3zt49r0000gn/T/ipykernel_24960/2729016999.py:15: FutureWarning: DataFrame.set_axis
'inplace' keyword is deprecated and will be removed in a future version. Use `obj = obj.set_axis(..., copy=False)` in
stead
  type_matchups.set_axis(names, axis=1,inplace=True)
```

Out[884]:

	Attacking Type (Rows) vs Defending Type (Cols)	Normal	Fire	Water	Electric	Grass	Ice	Fighting	Poison	Ground	Flying	Psychic	Bug	Rock	Ghost	Dragon	Dark	Steel	Fairy	Mean Attack Multiplier
8	Ground	1.0	2.0	1.0	2.0	0.5	1.0	1.0	2.0	1.0	0.0	1.0	0.5	2.0	1.0	1.0	1.0	2.0	1.0	1.166667
12	Rock	1.0	2.0	1.0	1.0	1.0	2.0	0.5	1.0	0.5	2.0	1.0	2.0	1.0	1.0	1.0	1.0	0.5	1.0	1.138889
1	Fire	1.0	0.5	0.5	1.0	2.0	2.0	1.0	1.0	1.0	1.0	1.0	2.0	0.5	1.0	0.5	1.0	2.0	1.0	1.111111
5	Ice	1.0	0.5	0.5	1.0	2.0	0.5	1.0	1.0	2.0	2.0	1.0	1.0	1.0	1.0	2.0	1.0	0.5	1.0	1.111111
9	Flying	1.0	1.0	1.0	0.5	2.0	1.0	2.0	1.0	1.0	1.0	1.0	2.0	0.5	1.0	1.0	1.0	0.5	1.0	1.083333
6	Fighting	2.0	1.0	1.0	1.0	1.0	2.0	1.0	0.5	1.0	0.5	0.5	0.5	2.0	0.0	1.0	2.0	2.0	0.5	1.083333
17	Fairy	1.0	0.5	1.0	1.0	1.0	1.0	2.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0	2.0	2.0	0.5	1.0	1.083333
2	Water	1.0	2.0	0.5	1.0	0.5	1.0	1.0	1.0	2.0	1.0	1.0	1.0	2.0	1.0	0.5	1.0	1.0	1.0	1.083333
16	Steel	1.0	0.5	0.5	0.5	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	0.5	2.0	1.055556
13	Ghost	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	2.0	1.0	0.5	1.0	1.0	1.027778
15	Dark	1.0	1.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0	2.0	1.0	1.0	2.0	1.0	0.5	1.0	0.5	1.027778
10	Psychic	1.0	1.0	1.0	1.0	1.0	1.0	2.0	2.0	1.0	1.0	0.5	1.0	1.0	1.0	1.0	0.0	0.5	1.0	1.000000
4	Grass	1.0	0.5	2.0	1.0	0.5	1.0	1.0	0.5	2.0	0.5	1.0	0.5	2.0	1.0	0.5	1.0	0.5	1.0	0.972222
3	Electric	1.0	1.0	2.0	0.5	0.5	1.0	1.0	1.0	0.0	2.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0	0.972222
11	Bug	1.0	0.5	1.0	1.0	2.0	1.0	0.5	0.5	1.0	0.5	2.0	1.0	1.0	0.5	1.0	2.0	0.5	0.5	0.972222
14	Dragon	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	0.5	0.0	0.972222
7	Poison	1.0	1.0	1.0	1.0	2.0	1.0	1.0	0.5	0.5	1.0	1.0	1.0	0.5	0.5	1.0	1.0	0.0	2.0	0.944444
0	Normal	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.0	1.0	1.0	0.5	1.0	0.888889

Computing Each Pokemon's Average Attack Multiplier

Since some pokemon have two types, it's important to look at not just the attack multiplier of one of its types, but the average of both of them. I decided to add the attack multiplier for each pokemon to our Pokedex database, so we can compare the average attack multipliers of each generation

In [885]:
```
# Creates new Attack Multiplier column for each pokemon in our database
pokedex['Attack Multiplier'] = ""
for index, row in pokedex.iterrows():
    # Temp variable to store the pokemon's average attack multiplier
    temp_attack_mult = 0
    # Searches through the Type Matchup dataframe in order to extract the
    for index1, row1 in type_matchups.iterrows():
        # Searches for the Average Attack Multipliers for both of the pokemon's types
        if(row1['Attacking Type (Rows) vs Defending Type (Cols)'] == row['Type1']):
            temp_attack_mult += type_matchups.at[index1, 'Mean Attack Multiplier']
        if(row1['Attacking Type (Rows) vs Defending Type (Cols)'] == row['Type2']):
            temp_attack_mult += type_matchups.at[index1, 'Mean Attack Multiplier']
    # If pokemon only has one type, return the Attack Multiplier as is
    if(row['Type2'] == None):
        pokedex.at[index, 'Attack Multiplier'] = temp_attack_mult
    # If pokemon has two types, return average of attack multipliers for its two types
    else:
        pokedex.at[index, 'Attack Multiplier'] = temp_attack_mult / 2
pokedex
```

Out[885]:

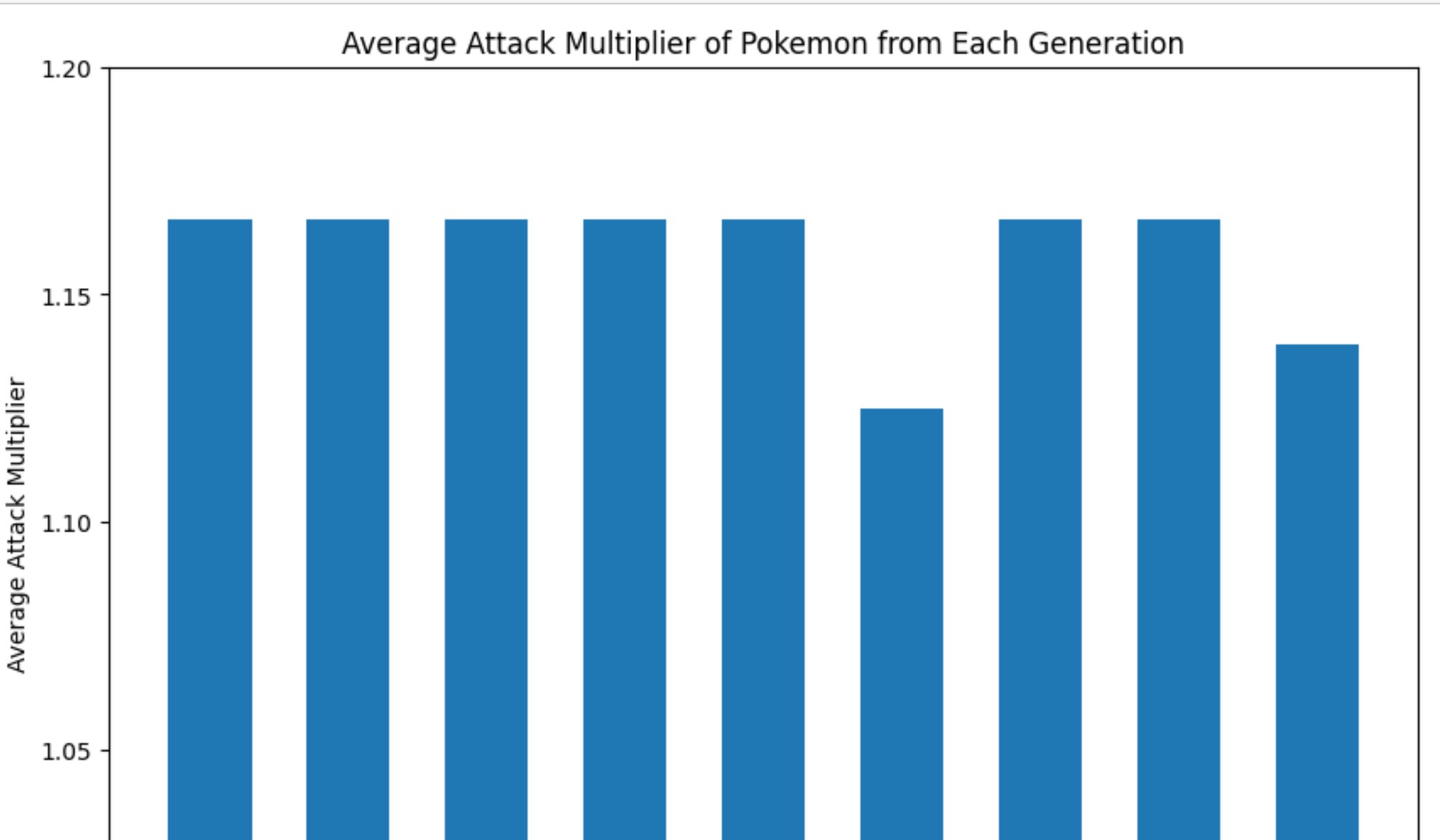
	#	Name	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Type1	Type2	Generation	Attack Multiplier
0	1	Bulbasaur	318	45	49	49	65	65	45	Grass	Poison	1	0.958333
1	2	Ivysaur	405	60	62	63	80	80	60	Grass	Poison	1	0.958333
2	3	Venusaur	525	80	82	83	100	100	80	Grass	Poison	1	0.958333
3	4	Charmander	309	39	52	43	60	50	65	Fire	None	1	1.111111
4	5	Charmeleon	405	58	64	58	80	65	80	Fire	None	1	1.111111
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1003	1004	Chi-Yu	570	55	80	80	135	120	100	Dark	Fire	9	1.069444
1004	1005	Roaring Moon	590	105	139	71	55	101	119	Dragon	Dark	9	1.0
1005	1006	Iron Valiant	590	74	130	90	120	60	116	Fairy	Fighting	9	1.083333
1006	1007	Koraidon	670	100	135	115	85	100	135	Fighting	Dragon	9	1.027778
1007	1008	Miraidon	670	100	85	100	135	115	135	Electric	Dragon	9	0.972222

1008 rows × 13 columns

Comparing Attack Multipliers of Each Generation

In order to compare the relative power of each generation, I thought it would first be a good idea to look at the averages of all the attack multipliers for each generation. From the results, it seems that Nintendo has made sure that the average attack multiplier of each generation has remained within ±.05 of each other. Contrary to what we expected, generations 6 and 9(some of the newer generations) are actually the only generations that are inconsistent with other generations' average attack multiplier. However, when it comes to pokemon, the deciding factor of whether or not a pokemon is more powerful than others is a combination of its attack stats as well as type matchups

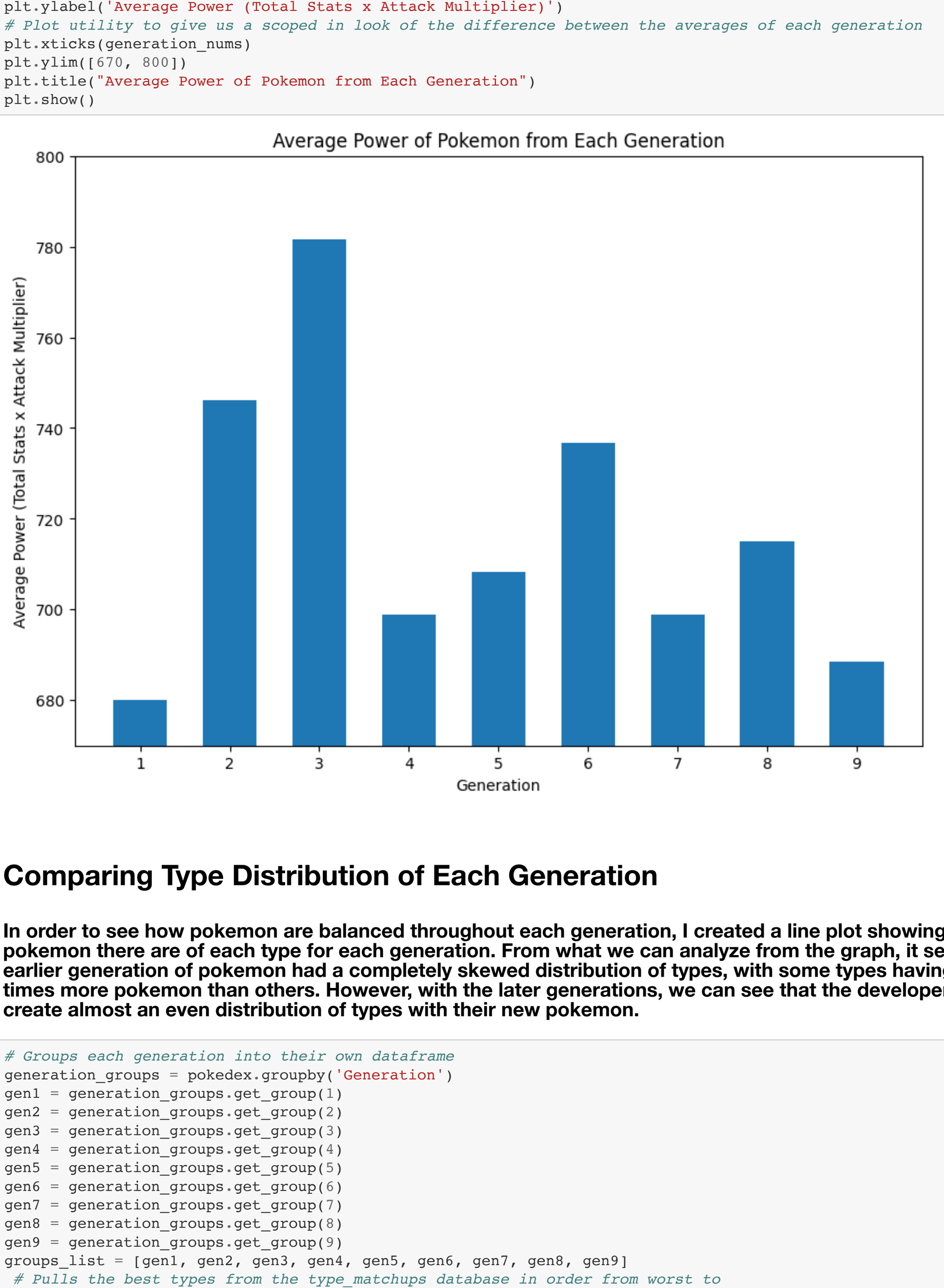
In [886]:
```
import matplotlib.pyplot as plt
# Pulls dataframe series for each of our axis
gen = pokedex['Generation']
mean_attack = pokedex['Attack Multiplier']
# Creates a reasonably size bar plot of the mean attack multiplier for each generation
fig = plt.figure(figsize =(10, 7))
plt.bar(gen, mean_attack, width = 0.6)
plt.xlabel('Generation')
plt.ylabel('Average Attack Multiplier')
# Plot utility to give us a scoped in look of the difference between the averages of each generation
generation_nums = [1,2,3,4,5,6,7,8,9]
plt.title('Average Attack Multiplier of Pokemon from Each Generation')
plt.xticks(generation_nums)
plt.yticks([1.0, 1.05, 1.1, 1.15, 1.2])
plt.ylim([1.0, 1.2])
plt.show()
```



Comparing Relative Power Level of Each Generation

A better method to compare a pokemon's relative strength is to look at not only it's average attack multiplier, but its base stats as well. Each pokemon's strength is determined from 6 different stats: Attack, Special Attack, Defense, Special Defense, Health, and Speed. Since these different stats make up how strong a pokemon is, a good gauge is to look at the total stats times the attack multipliers. I decided to create a new column named 'Power', which is the combination of the pokemon's stats and average attack multiplier. As we can see from the graph, it seems that there is a recurring pattern of gradual increase in power every 3 generations, with a sudden drop in power when it comes to the next generation. The gradual increase in the average power of every 3 generations is probably what makes some of the older pokemons feel unpolished compared to the new ones, but it looks like instead of increasing the power for every generation consecutively, Nintendo drops the power levels of the first generation in each trilogy so that pokemon of the previous generations aren't rendered useless

In [887]:
```
# Creates graph of pokemon's power(total stats * attack multiplier)
pokedex['Power'] = ""
for index, row in pokedex.iterrows():
    pokedex.at[index, 'Power'] = pokedex.at[index, 'Attack Multiplier'] * pokedex.at[index, 'Total']
# Pulls dataframe series for each of our axis
gen = pokedex['Generation']
mean_power = pokedex['Power']
# Creates a reasonably size bar plot of the mean attack multiplier for each generation
fig = plt.figure(figsize =(10, 7))
plt.bar(gen, mean_power, width = 0.6)
plt.xlabel('Generation')
plt.ylabel('Average Power (Total Stats x Attack Multiplier)')
# Plot utility to give us a scoped in look of the difference between the averages of each generation
plt.xticks(generation_nums)
plt.ylim([670, 800])
plt.title("Average Power of Pokemon from Each Generation")
plt.show()
```



Comparing Type Distribution of Each Generation

In order to see how pokemon are balanced throughout each generation, I created a line plot showing how many pokemon there are of each type for each generation. From what we can analyze from the graph, it seems as if the earlier generation of pokemon had a completely skewed distribution of types, with some types having almost 6-7 times more pokemon than others. However, with the later generations, we can see that the developers started to create almost an even distribution of types with their new pokemon.

In [888]:
```
# Groups each generation into their own dataframe
generation_groups = pokedex.groupby('Generation')
gen1 = generation_groups.get_group(1)
gen2 = generation_groups.get_group(2)
gen3 = generation_groups.get_group(3)
gen4 = generation_groups.get_group(4)
gen5 = generation_groups.get_group(5)
gen6 = generation_groups.get_group(6)
gen7 = generation_groups.get_group(7)
gen8 = generation_groups.get_group(8)
gen9 = generation_groups.get_group(9)
groups_list = [gen1, gen2, gen3, gen4, gen5, gen6, gen7, gen8, gen9]
 # Pulls the best types from the type_matchups database in order from worst to
# best based on mean attack multiplier
types_list = type_matchups['Attacking Type (Rows) vs Defending Type (Cols)'].tolist()
types_list.reverse()
# Creates a 2d list which will store the number of types for each gen
# Format: [generation_index][type_index]
gen_list = []
for i in range(len(generation_nums)):
    gen_list.append([0] * len(types_list))
# Updates the array for each group
for group in groups_list:
    for index, row in group.iterrows():
        for i in range(len(types_list)):
            if(row['Type1'] == types_list[i]):
                gen_list[row['Generation'] - 1][i] += 1
            if(row['Type2'] == types_list[i]):
                gen_list[row['Generation'] - 1][i] += 1
# Num x steps
x = np.arange(18)
# Create graph of type distribution
fig, ax = plt.subplots(figsize=(21,7))
plt.plot(x, gen_list[0], color='red', label = 'Gen 1',)
plt.plot(x, gen_list[1], color='blue', label = 'Gen 2',)
plt.plot(x, gen_list[2], color='cyan', label = 'Gen 3',)
plt.plot(x, gen_list[3], color='green', label = 'Gen 4',)
plt.plot(x, gen_list[4],  color='purple', label = 'Gen 5',)
plt.plot(x, gen_list[5], color='orange', label = 'Gen 6',)
plt.plot(x, gen_list[6], color='pink', label = 'Gen 7',)
plt.plot(x, gen_list[7], color='brown', label = 'Gen 8',)
plt.plot(x, gen_list[8], color='black', label = 'Gen 9',)
plt.xticks(x, types_list)
plt.xlabel('Types')
plt.ylabel('Number of Pokemon')
generation_labels = ['Gen 1', 'Gen 2','Gen 3','Gen 4','Gen 5','Gen 6','Gen 7','Gen 8','Gen 9',]
plt.legend(generation_labels)
```

Out[888]: <matplotlib.legend.Legend at 0x14d6e1690>

Analysis

By comparing the average attack multiplier and power levels of each generation of pokemon, I'm able to conclude that the developers over at Nintendo have definitely improved at balancing their game over time, and created a balanced system where older pokemon won't lose their luster over new ones. Even though the power level of the first trilogy of the Pokemon franchise is much more skewed than the later 2 trilogies, we can see that over time, the power level of each generation has more or less remained pretty balanced. We can see that for each trilogy, the average power levels increase, and with each new trilogy, it drops back down, suggesting that Nintendo attempts to balance their new generations of pokemon by resetting the maximum threshold of the average pokemon's power. From the line graph, we can also see that over time, the distributions of types for each generation has become more even, suggesting that they are attempting to make each new generation have a balanced set of pokemon types, which would in turn create a similar model of average attack multipliers for their future games. In conclusion, pokemon fans shouldn't have to worry about their favorite pokemon becoming outdated/irrelevant, for it is apparent that Nintendo is going in the right step to make each new generation balanced and each of our beloved pokemon viable