Data Visualization

Denaco Analytics

You've learned to use `NumPy` and `Pandas` to read and manipulate your data from a statistical and mathematical standpoint. Now, you'll visualize your data in the form of graphs/charts, to get insights that the statistics alone may not completely convey.

The current and the next lesson will help you learn to draw a variety of informative statistical visualizations using the `Matplotlib` and `Seaborn` packages.

The current lesson will focus on introducing univariate visualizations: bar charts, and histograms. By the end of this lesson, you will be able to:

1. Create bar charts for qualitative variables, for example, the amount (number) of eggs consumed in a meal (categories: {breakfast, lunch, or dinner}). In general, bar chart maps categories to numbers.

2. Create Pie charts. A pie chart is a common univariate plot type that is used to depict relative frequencies for levels of a categorical variable. A pie chart is preferably used when the number of categories is less, and you'd like to see the proportion of each category.

3. Create histograms for quantitative variables. A histogram splits the (tabular) data into evenly sized intervals and displays the count of rows in each interval with bars. A histogram is similar to a bar chart, except that the "category" here is a range of values.

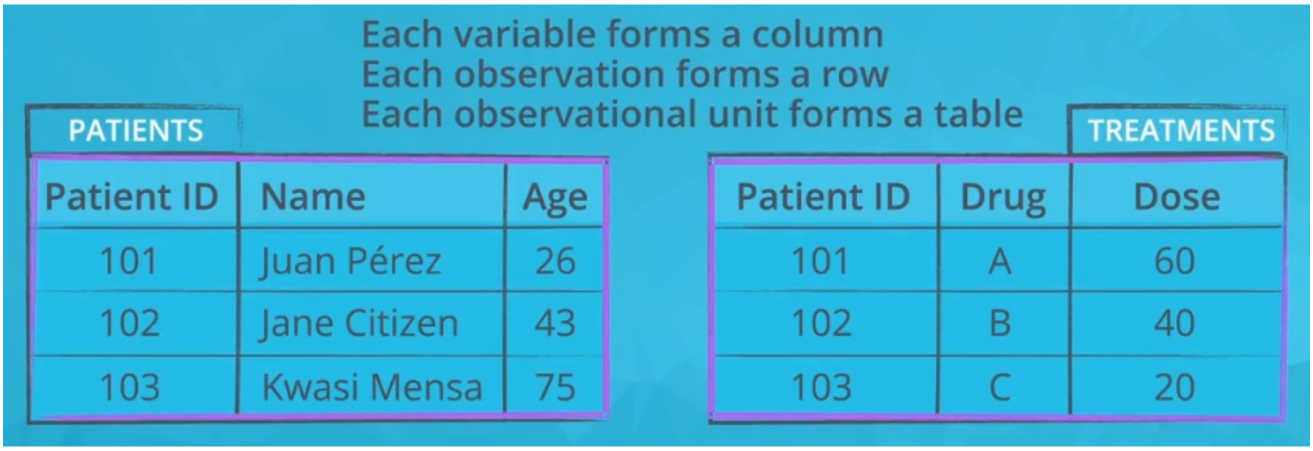
4. Analyze the bar charts and histograms.

Once you have the foundational knowledge of Matplotlib and Seaborn, we will move on to the next lesson (`part-2`), where you'll learn advanced visualizations such as heat map, scatter plot, violin plots, box plots, clustered bar charts, and many others.

What is Tidy Data?

In this course, it is expected that your data is organized in some kind of tidy format. In short, a tidy dataset is a tabular dataset where:

1. each variable is a column
2. each observation is a row
3. each type of observational unit is a table



Each variable forms a column
Each observation forms a row
Each observational unit forms a table

PATIENTS

Patient ID	Name	Age
101	Juan Pérez	26
102	Jane Citizen	43
103	Kwasi Mensa	75

TREATMENTS

Patient ID	Drug	Dose
101	A	60
102	B	40
103	C	20

A **bar chart** depicts the distribution of a categorical variable. In a bar chart, each level of the categorical variable is depicted with a bar, whose height indicates the frequency of data points that take on that level.

Bar Chart using Seaborn

A basic bar chart of frequencies can be created through the use of seaborn's countplot function.

```
seaborn.countplot(*, x=None, y=None, data=None, order=None, orient=None, color=None)
```

We will see the usage of a few of the arguments of the `countplot()` function.

Example 1. Create a vertical bar chart using Seaborn, with default colors

In [2]:

```
# Necessary imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline
```

In [3]:

```
# Read the csv file, and check its top 10 rows
pokemon = pd.read_csv('pokemon.csv')
print(pokemon.shape)
pokemon.head(10)
```

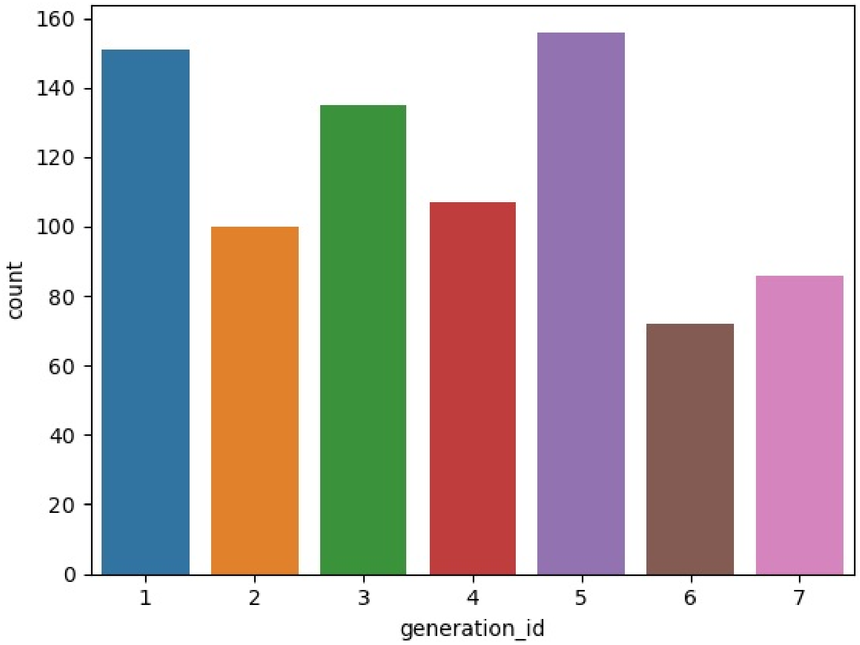
```
(807, 14)
```

Out[3]:

	id	species	generation_id	height	weight	base_experience	type_1	type_2	hp	attack	defense	speed	special-attack	special-defense
0	1	bulbasaur	1	0.7	6.9	64	grass	poison	45	49	49	45	65	65
1	2	ivysaur	1	1.0	13.0	142	grass	poison	60	62	63	60	80	80
2	3	venusaur	1	2.0	100.0	236	grass	poison	80	82	83	80	100	100
3	4	charmander	1	0.6	8.5	62	fire	NaN	39	52	43	65	60	50
4	5	charmeleon	1	1.1	19.0	142	fire	NaN	58	64	58	80	80	65
5	6	charizard	1	1.7	90.5	240	fire	flying	78	84	78	100	109	85
6	7	squirtle	1	0.5	9.0	63	water	NaN	44	48	65	43	50	64
7	8	wartortle	1	1.0	22.5	142	water	NaN	59	63	80	58	65	80
8	9	blastoise	1	1.6	85.5	239	water	NaN	79	83	100	78	85	105
9	10	caterpie	1	0.3	2.9	39	bug	NaN	45	30	35	45	20	20

In [8]:

```
# A semicolon (;) at the end of the statement will supress printing the plotting information
sb.countplot(data=pokemon, x='generation_id');
```



In the example above, all the bars have a different color. This might come in handy for building associations between these category labels and encodings in plots with more variables. Otherwise, it's a good idea to simplify the plot and reduce unnecessary distractions by plotting all bars in the same color. You can choose to have a uniform color across all bars, by using the `color` argument, as shown in the example below:

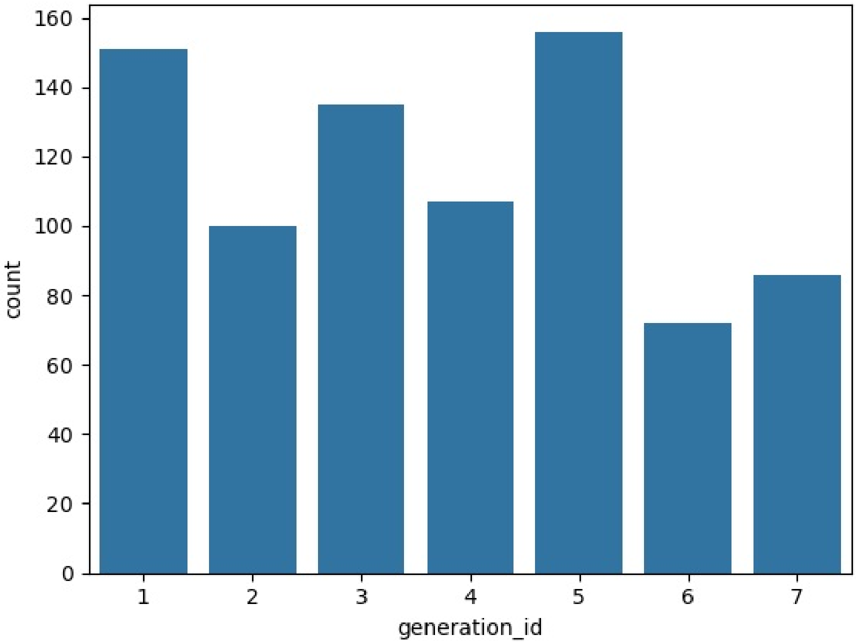
Example 2. Create a vertical bar chart using Seaborn, with a uniform single color

In [12]:

```
# The `color_palette()` returns the the current / default palette as a list of RGB tuples.
# Each tuple consists of three digits specifying the red, green, and blue channel values to specify a color.
# Choose the first tuple of RGB colors
base_color = sb.color_palette()[0]

# Use the `color` argument
sb.countplot(data=pokemon, x='generation_id', color=base_color)
```

Out[12]:

```
<AxesSubplot:xlabel='generation_id', ylabel='count'>
```



Bar Chart using the Matplotlib

You can even create a similar bar chart using the Matplotlib, instead of Seaborn. We will use the `matplotlib.pyplot.bar()` function to plot the chart. The syntax is:

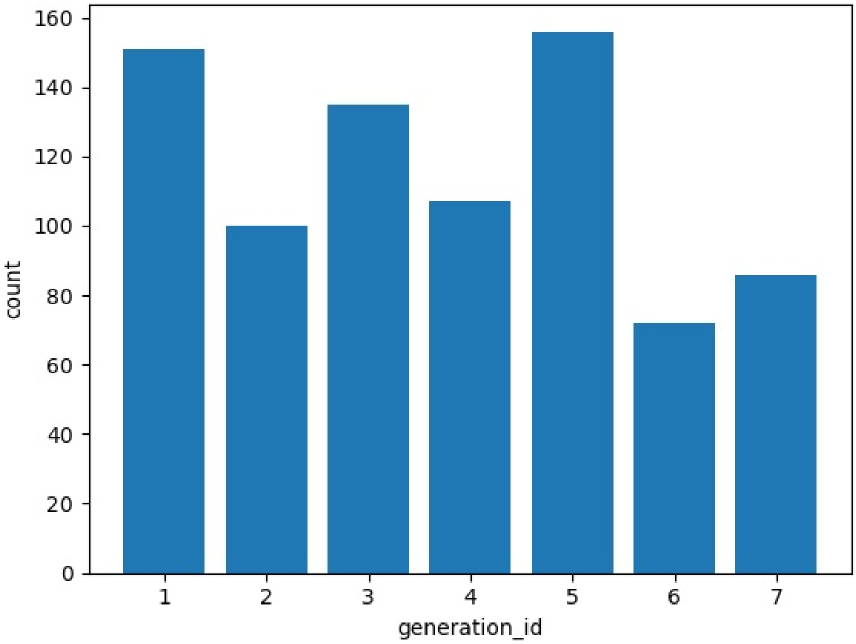
```
matplotlib.pyplot.bar(x, y, width=0.8, bottom=None, *, align='center', data=None)
```

Refer to the documentation for the details of optional arguments. In the example below, we will use `Series.value_counts()` to extract a Series from the given DataFrame object.

Example 3. Create a vertical bar chart using Matplotlib, with a uniform single color

In [23]:

```
# Return the Series having unique values
x = pokemon['generation_id'].unique()

# Return the Series having frequency count of each unique value
y = pokemon['generation_id'].value_counts(sort=False)

plt.bar(x, y)

# Labeling the axes
plt.xlabel('generation_id')
plt.ylabel('count')

# Dsiplay the plot
plt.show()
```



There is a lot more you can do with both Seaborn and Matplotlib bar charts. The remaining examples will experiment with seaborn's `countplot()` function.

For nominal-type data, one common operation is to sort the data in terms of frequency. In the examples shown above, you can even order the bars as desirable. With our data in a pandas DataFrame, we can use various DataFrame methods to compute and extract an ordering, then set that ordering on the "order" parameter:

This can be done by using the `order` argument of the `countplot()` function.

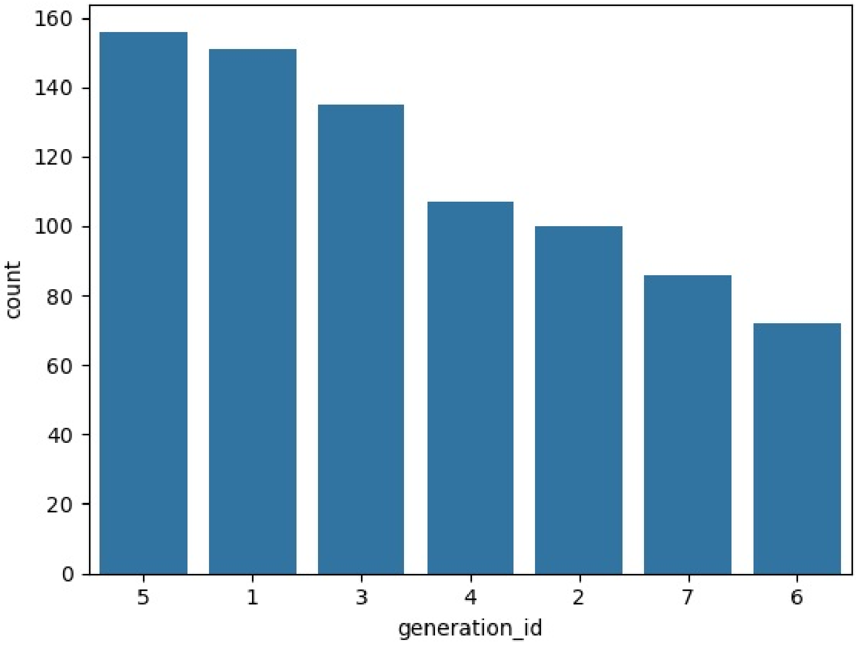
Example 4. Static and dynamic ordering of the bars in a bar chart using `seaborn.countplot()`

In [25]:

```python
# Static-ordering the bars
sb.countplot(data=pokemon, x='generation_id', color=base_color, order=[5,1,3,4,2,7,6]);

# Dynamic-ordering the bars
# The order of the display of the bars can be computed with the following logic.
# Count the frequency of each unique value in the 'generation_id' column, and sort it in descending order
# Returns a Series

freq = pokemon['generation_id'].value_counts()

# Get the indexes of the Series
gen_order = freq.index

# Plot the bar chart in the decreasing order of the frequency of the `generation_id`
sb.countplot(data=pokemon, x='generation_id', color=base_color, order=gen_order)
```

Out[25]:

```
<AxesSubplot:xlabel='generation_id', ylabel='count'>
```



While we could sort the levels by frequency like above, we usually care about whether the most frequent values are at high levels, low levels, etc. For ordinal-type data, we probably want to sort the bars in order of the variables. The best thing for us to do in this case is to convert the column into an ordered categorical data type.

> > Additional Variation - Refer to the `CategoricalDtype` to convert the column into an ordered categorical data type. By default, pandas reads in string data as object types, and will plot the bars in the order in which the unique values were seen. By converting the data into an ordered type, the order of categories becomes innate to the feature, and we won't need to specify an "order" parameter each time it's required in a plot.

Should you find that you need to sort an ordered categorical type in a different order, you can always temporarily override the data type by setting the "order" parameter as above.

The category labels in the examples above are very small. In case, the category labels have large names, you can make use of the `matplotlib.pyplot.xticks(rotation=90)` function, which will rotate the category labels (not axes) counter-clockwise 90 degrees.

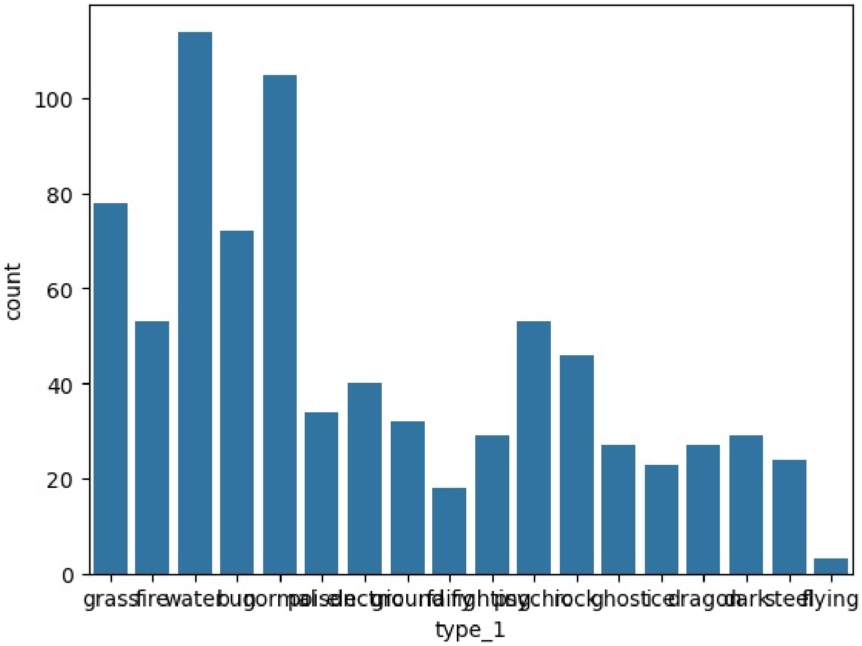
Example 5. Rotate the category labels (not axes)

In [26]:

```
# Plot the Pokemon type on a Vertical bar chart
sb.countplot(data=pokemon, x='type_1', color = base_color)
```

Out[26]:

```
<AxesSubplot:xlabel='type_1', ylabel='count'>
```

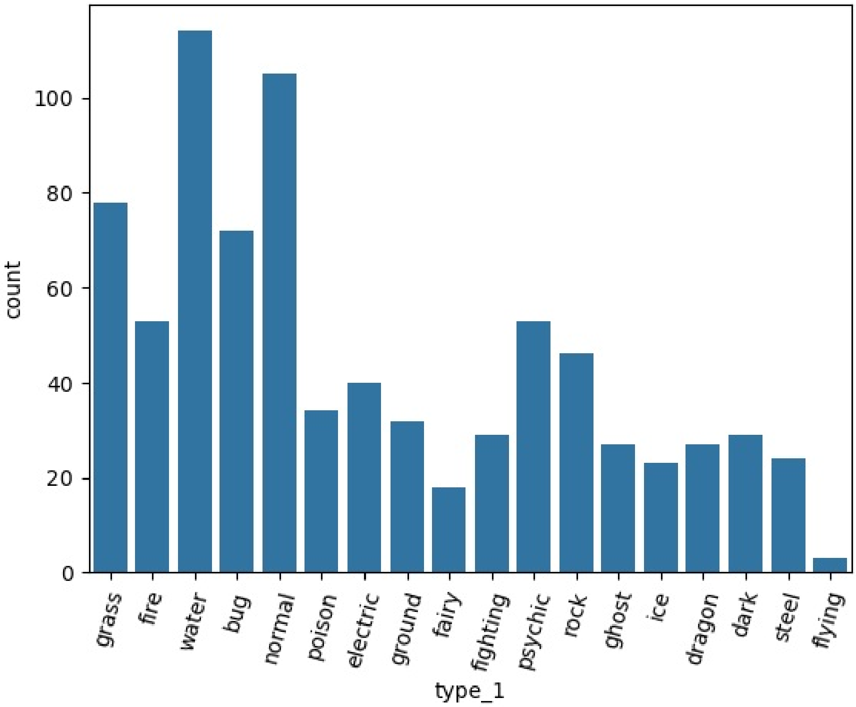


In [31]:

```
# Plot the Pokemon type on a Vertical bar chart
sb.countplot(data=pokemon, x='type_1', color = base_color)

# Use xticks to rotate the category labels (not axes) counter-clockwise
plt.xticks(rotation=75)
```

Out[31]:

```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17]),
 [Text(0, 0, 'grass'),
  Text(1, 0, 'fire'),
  Text(2, 0, 'water'),
  Text(3, 0, 'bug'),
  Text(4, 0, 'normal'),
  Text(5, 0, 'poison'),
  Text(6, 0, 'electric'),
  Text(7, 0, 'ground'),
  Text(8, 0, 'fairy'),
  Text(9, 0, 'fighting'),
  Text(10, 0, 'psychic'),
  Text(11, 0, 'rock'),
  Text(12, 0, 'ghost'),
  Text(13, 0, 'ice'),
  Text(14, 0, 'dragon'),
  Text(15, 0, 'dark'),
  Text(16, 0, 'steel'),
  Text(17, 0, 'flying')])
```

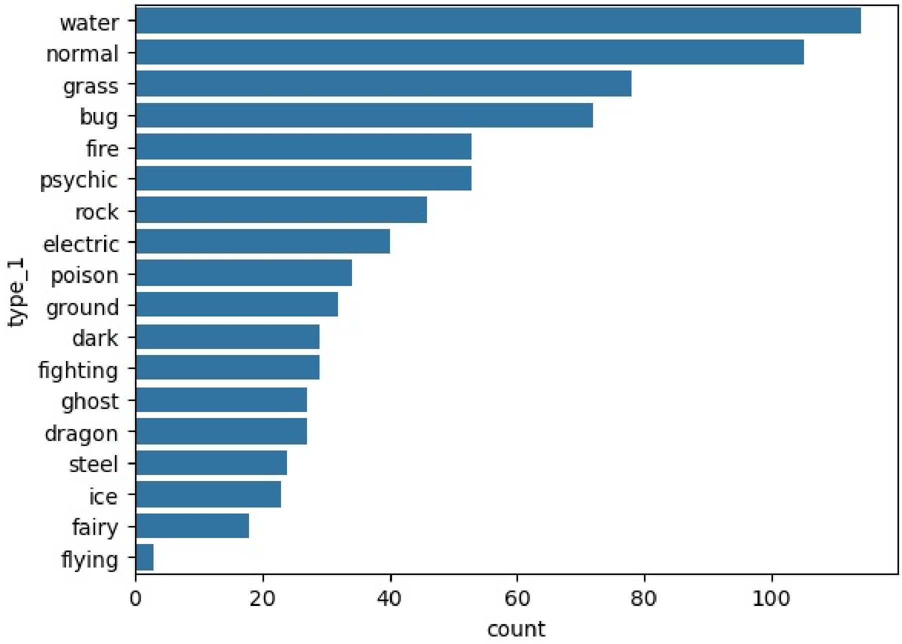


Even after using the `matplotlib.pyplot.xticks(rotation=90)` function, if the category labels do not fit well, you can rotate the axes.

Example 6. Rotate the axes clockwise

In [36]:

```
# Plot the Pokemon type on a Horizontal bar chart
order = pokemon['type_1'].value_counts().index
sb.countplot(data=pokemon, y='type_1', color=base_color, order=order);
```



Absolute vs. Relative Frequency

By default, seaborn's `countplot` function will summarize and plot the data in terms of `absolute frequency`, or pure counts. In certain cases, you might want to understand the distribution of data or want to compare levels in terms of the proportions of the whole. In this case, you will want to plot the data in terms of `relative frequency`, where the height indicates the proportion of data taking each level, rather than the absolute count.

One method of plotting the data in terms of relative frequency on a bar chart is to just relabel the count's axis in terms of proportions. The underlying data will be the same, it will simply be the scale of the axis ticks that will be changed.

Example 1. Demonstrate data wrangling, and plot a horizontal bar chart.

Example 1 - Step 1. Make the necessary import

In [2]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline

# Read the data from a CSV file
pokemon = pd.read_csv('pokemon.csv')
print(pokemon.shape)
pokemon.head(10)
```

```
(807, 14)
```

Out[2]:

	id	species	generation_id	height	weight	base_experience	type_1	type_2	hp	attack	defense	speed	special-attack	special-defense
0	1	bulbasaur	1	0.7	6.9	64	grass	poison	45	49	49	45	65	65
1	2	ivysaur	1	1.0	13.0	142	grass	poison	60	62	63	60	80	80
2	3	venusaur	1	2.0	100.0	236	grass	poison	80	82	83	80	100	100
3	4	charmander	1	0.6	8.5	62	fire	NaN	39	52	43	65	60	50
4	5	charmeleon	1	1.1	19.0	142	fire	NaN	58	64	58	80	80	65
5	6	charizard	1	1.7	90.5	240	fire	flying	78	84	78	100	109	85
6	7	squirtle	1	0.5	9.0	63	water	NaN	44	48	65	43	50	64
7	8	wartortle	1	1.0	22.5	142	water	NaN	59	63	80	58	65	80
8	9	blastoise	1	1.6	85.5	239	water	NaN	79	83	100	78	85	105
9	10	caterpie	1	0.3	2.9	39	bug	NaN	45	30	35	45	20	20

Last time we created the bar chart of pokemon by their type_1. Let's club the rows of both `type_1` and `type_2`, so that the resulting dataframe has new column, type_level.

This operation will double the number of rows in pokemon from 807 to 1614. Data Wrangling Step

We will use the `pandas.DataFrame.melt()` method to unpivot a DataFrame from wide to long format, optionally leaving identifiers set. The syntax is:

```
DataFrame.melt(id_vars, value_vars, var_name, value_name, col_level, ignore_index)
```

It is essential to understand the parameters involved:

1. `id_vars` - It is a tuple representing the column(s) to use as identifier variables.
2. `value_vars` - It is tuple representing the column(s) to unpivot (remove, out of place).
3. `var_name` - It is a name of the new column.
4. `value_name` - It is a name to use for the ‘value’ of the columns that are unpivoted.

The function below will do the following in the pokemon dataframe out of place:

1. Select the 'id', and 'species' columns from pokemon.
2. Remove the 'type_1', 'type_2' columns from pokemon
3. Add a new column 'type_level' that can have a value either 'type_1' or 'type_2'
4. Add another column 'type' that will contain the actual value contained in the 'type_1', 'type_2' columns. For example, the first row in the pokemon dataframe having `id=1` and species=bulbasaur will now occur twice in the resulting dataframe after the `melt()` operation. The first occurrence will have `type=grass`, whereas, the second occurrence will have `type=poison`.

Example 1 - Step 2. Data wrangling to reshape the pokemon dataframe

In [43]:

```
pkmn_types = pokemon.melt(id_vars=['id', 'species'],
                          value_vars=['type_1', 'type_2'],
                          var_name='type_level',
                          value_name='type')
pkmn_types.head(10)
#pkmn_types.shape
```

Out[43]:

	id	species	type_level	type
0	1	bulbasaur	type_1	grass
1	2	ivysaur	type_1	grass
2	3	venusaur	type_1	grass
3	4	charmander	type_1	fire
4	5	charmeleon	type_1	fire
5	6	charizard	type_1	fire
6	7	squirtle	type_1	water
7	8	wartortle	type_1	water
8	9	blastoise	type_1	water
9	10	caterpie	type_1	bug

Example 1 - Step 3. Find the frequency of unique values in the type column

In [44]:

```
# Count the frequency of unique values in the `type` column of pkmn_types dataframe.
# By default, returns the decreasing order of the frequency.
type_counts = pkmn_types['type'].value_counts()
type_counts
```

Out[44]:

```
water       131
normal      109
flying       98
grass        97
psychic      82
bug          77
poison       66
ground       64
fire         64
rock         60
fighting     54
electric     48
fairy        47
steel        47
dark         46
dragon       45
ghost        43
ice          34
Name: type, dtype: int64
```

In [45]:

```
# Get the unique values of the `type` column, in the decreasing order of the frequency.
type_order = type_counts.index
type_order
```

Out[45]:

```
Index(['water', 'normal', 'flying', 'grass', 'psychic', 'bug', 'poison',
       'ground', 'fire', 'rock', 'fighting', 'electric', 'fairy', 'steel',
       'dark', 'dragon', 'ghost', 'ice'],
      dtype='object')
```

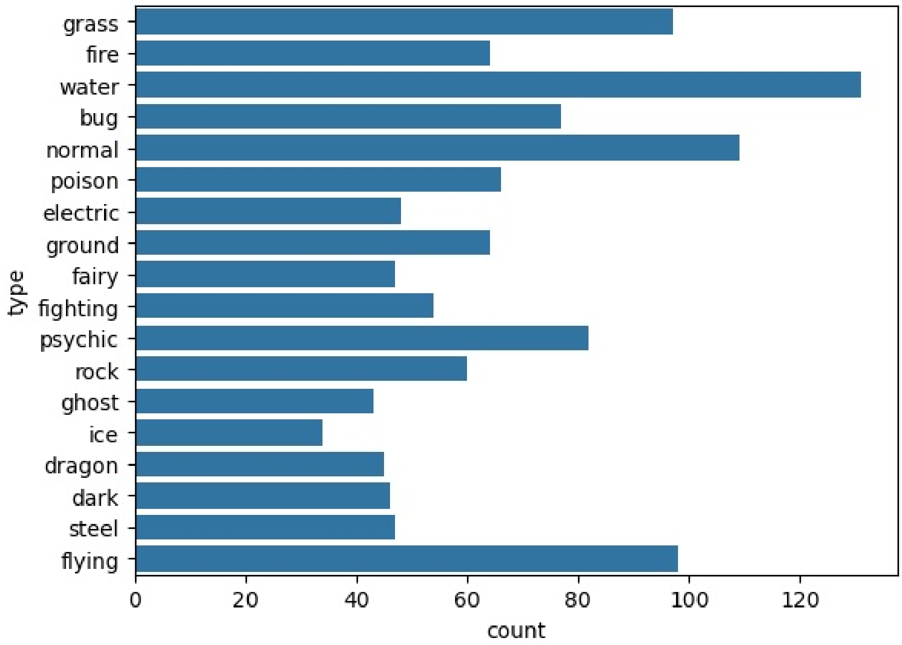
Example 1 - Step 4. Plot the horizontal bar charts

In [47]:

```
sb.countplot(data=pkmn_types, color=base_color, y='type')
```

Out[47]:

```
<AxesSubplot:xlabel='count', ylabel='type'>
```



Example 2. Plot a bar chart having the proportions, instead of the actual count, on one of the axes.

Example 2 - Step 1. Find the maximum proportion of bar

In [49]:

```
# Returns the sum of all not-null values in `type` column
n_pokemon = pkmn_types['type'].value_counts().sum()

# Return the highest frequency in the `type` column
max_type_count = type_counts[0]

# Return the maximum proportion, or in other words,
# compute the length of the longest bar in terms of the proportion
max_prop = max_type_count / n_pokemon
print(max_prop)
```

```
0.10808580858085809
```

Example 2 - Step 2. Create an array of evenly spaced proportioned values

In [50]:

```
# Use numpy.arange() function to produce a set of evenly spaced proportioned values
# between 0 and max_prop, with a step size 2\%
tick_props = np.arange(0, max_prop, 0.02)
tick_props
```

Out[50]:

```
array([0.  , 0.02, 0.04, 0.06, 0.08, 0.1 ])
```

We need x-tick labels that must be evenly spaced on the x-axis. For this purpose, we must have a list of labels ready with us, before using it with `plt.xticks()` function.

Example 2 - Step 3. Create a list of String values that can be used as tick labels.

In [52]:

```
# Use a list comprehension to create tick_names that we will apply to the tick labels.
# Pick each element `v` from the `tick_props`, and convert it into a formatted string.
# `{:0.2f}` denotes that before formatting, we 2 digits of precision and `f` is used to represent floating poin
# Refer [here](https://docs.python.org/2/library/string.html#format-string-syntax) for more details
tick_names = ['{:0.2f}'.format(v) for v in tick_props]
tick_names
```

Out[52]:

```
['0.00', '0.02', '0.04', '0.06', '0.08', '0.10']
```

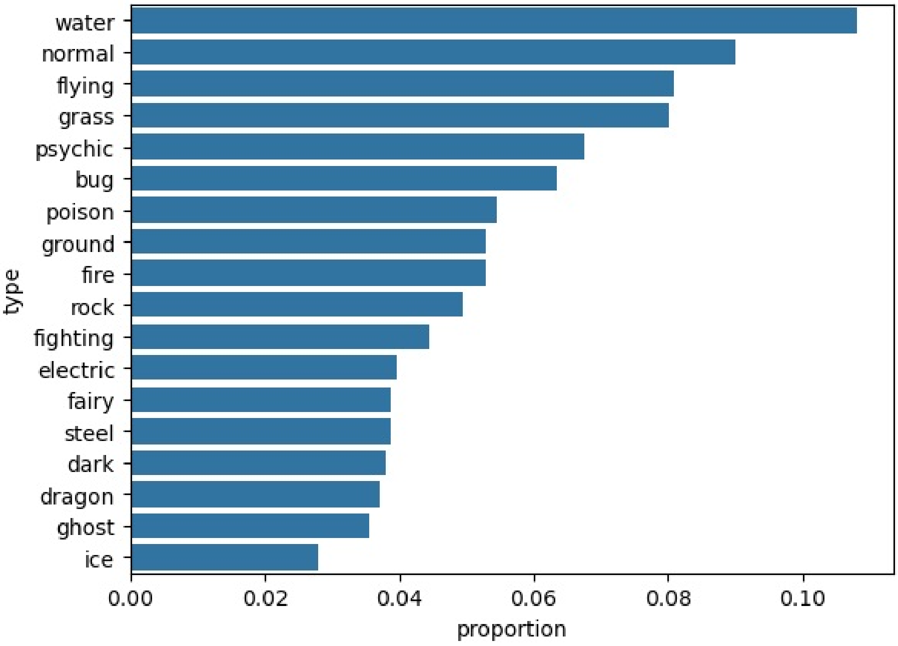
The `xticks` and `yticks` functions aren't only about rotating the tick labels. You can also get and set their locations and labels as well. The first argument takes the tick locations: in this case, the tick proportions multiplied back to be on the scale of counts. The second argument takes the tick names: in this case, the tick proportions formatted as strings to two decimal places.

I've also added a `ylabel` call to make it clear that we're no longer working with straight counts.

Example 2 - Step 4. Plot the bar chart, with new `x-tick` labels

In [53]:

```
sb.countplot(data=pkmn_types, y='type', color=base_color, order=type_order);
# Change the tick locations and labels
plt.xticks(tick_props * n_pokemon, tick_names)
plt.xlabel('proportion');
```



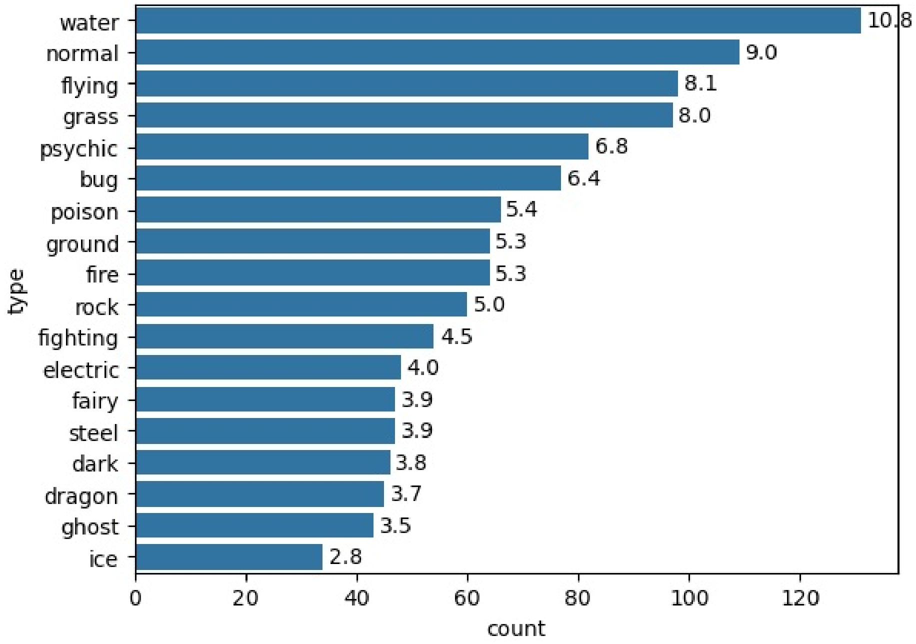
Additional Variation

Rather than plotting the data on a relative frequency scale, you might use text annotations to label the frequencies on bars instead. This requires writing a loop over the tick locations and labels and adding one text element for each bar.

Example 3. Print the text (proportion) on the bars of a horizontal plot.

In [56]:

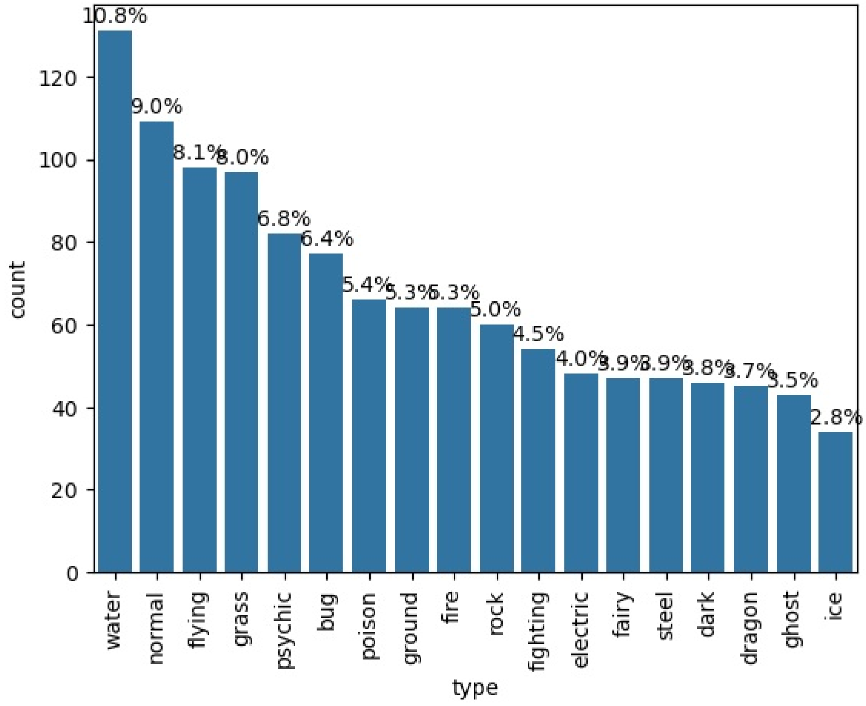
```
# Considering the same chart from the Example 1 above, print the text (proportion) on the bars
base_color = sb.color_palette()[0]
sb.countplot(data=pkmn_types, y='type', color=base_color, order=type_order);

# Logic to print the proportion text on the bars
for i in range (type_counts.shape[0]):
    # Remember, type_counts contains the frequency of unique values in the `type` column in decreasing order.
    count = type_counts[i]
    # Convert count into a percentage, and then into string
    pct_string = '{:0.1f}'.format(100*count/n_pokemon)
    # Print the string value on the bar.
    # Read more about the arguments of text() function [here](https://matplotlib.org/3.1.1/api/_as_gen/matplotl
    plt.text(count+1, i, pct_string, va='center')
```



Example 4. Print the text (proportion) below the bars of a Vertical plot.

In [57]:

```
# Considering the same chart from the Example 1 above, print the text (proportion) BELOW the bars
base_color = sb.color_palette()[0]
sb.countplot(data=pkmn_types, x='type', color=base_color, order=type_order);


# Recalculating the type_counts just to have clarity.
type_counts = pkmn_types['type'].value_counts()

# get the current tick locations and labels
locs, labels = plt.xticks(rotation=90)

# loop through each pair of locations and labels
for loc, label in zip(locs, labels):

    # get the text property for the label to get the correct count
    count = type_counts[label.get_text()]
    pct_string = '{:0.1f}%'.format(100*count/n_pokemon)

    # print the annotation just below the top of the bar
    plt.text(loc, count+2, pct_string, ha = 'center', color = 'black')
```



I use the `.get_text()` method to obtain the category name, so I can get the count of each category level. At the end, I use the `text` function to print each percentage, with the x-position, y-position, and string as the three main parameters to the function.

Tip: Is the text on the bars not readable clearly? Consider changing the size of the plot by using the following:

In [70]:

```python
from matplotlib import rcParams
# Specify the figure size in inches, for both X, and Y axes
rcParams['figure.figsize'] = 12,4
```

Counting Missing Data

If you have a large dataframe, and it contains a few missing values (`None` or a `numpy.NaN`), then you can find the count of such missing value across the given label. For this purpose, you can use either of the following two analogous functions :

1. pandas.DataFrame.isna()
2. pandas.DataFrame.isnull()

The functions above are alias of each other and detect missing values by returning the same sized object as that of the calling dataframe, made up of boolean True/False.

Step 1. Load the dataset

In [75]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline

# Read the data from a CSV file
# Original source of data: https://www.kaggle.com/manjeetsingh/retaildataset available under C0 1.0 Universal (
sales_data = pd.read_csv('sales_data.csv')
sales_data.head(10)
```

Out[75]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	Is
0	1	05/02/2010	42.31	2.572	NaN	NaN	NaN	NaN	NaN	211.096358	8.106	
1	1	12/02/2010	38.51	2.548	NaN	NaN	NaN	NaN	NaN	211.242170	8.106	
2	1	19/02/2010	39.93	2.514	NaN	NaN	NaN	NaN	NaN	211.289143	8.106	
3	1	26/02/2010	46.63	2.561	NaN	NaN	NaN	NaN	NaN	211.319643	8.106	
4	1	05/03/2010	46.50	2.625	NaN	NaN	NaN	NaN	NaN	211.350143	8.106	
5	1	12/03/2010	57.79	2.667	NaN	NaN	NaN	NaN	NaN	211.380643	8.106	
6	1	19/03/2010	54.58	2.720	NaN	NaN	NaN	NaN	NaN	211.215635	8.106	
7	1	26/03/2010	51.45	2.732	NaN	NaN	NaN	NaN	NaN	211.018042	8.106	
8	1	02/04/2010	62.27	2.719	NaN	NaN	NaN	NaN	NaN	210.820450	7.808	
9	1	09/04/2010	65.86	2.770	NaN	NaN	NaN	NaN	NaN	210.622857	7.808	

In [77]:

```python
# Use either of the functions below
# sales_data.isna()
sales_data.isnull()
```

Out[77]:

	Store	Date	Temperature	Fuel_Price	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5	CPI	Unemployment	IsHoliday
0	False	False	False	False	True	True	True	True	True	False	False	False
1	False	False	False	False	True	True	True	True	True	False	False	False
2	False	False	False	False	True	True	True	True	True	False	False	False
3	False	False	False	False	True	True	True	True	True	False	False	False
4	False	False	False	False	True	True	True	True	True	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...
8185	False	False	False	False	False	False	False	False	False	True	True	False
8186	False	False	False	False	False	False	False	False	False	True	True	False
8187	False	False	False	False	False	False	False	False	False	True	True	False
8188	False	False	False	False	False	False	False	False	False	True	True	False
8189	False	False	False	False	False	False	False	False	False	True	True	False

8190 rows × 12 columns

In [78]:

```python
sales_data.shape
```

Out[78]:

```
(8190, 12)
```

We can use pandas functions to create a table with the number of missing values in each column. Once, you have the label-wise count of missing values, you try plotting the tabular data in the form of a bar chart.

In [80]:

```
sales_data.isnull().sum()
```

Out[80]:

```
Store              0
Date               0
Temperature        0
Fuel_Price         0
MarkDown1       4158
MarkDown2       5269
MarkDown3       4577
MarkDown4       4726
MarkDown5       4140
CPI              585
Unemployment     585
IsHoliday          0
dtype: int64
```

In [82]:

```
sales_data.isna().sum()
```

Out[82]:

```
Store              0
Date               0
Temperature        0
Fuel_Price         0
MarkDown1       4158
MarkDown2       5269
MarkDown3       4577
MarkDown4       4726
MarkDown5       4140
CPI              585
Unemployment     585
IsHoliday          0
dtype: int64
```

> What if we want to visualize these missing value counts?

One interesting way we can apply bar charts is through the visualization of missing data. We could treat the variable names as levels of a categorical variable, and create a resulting bar plot. However, since the data is not in its tidy, unsummarized form, we need to make use of a different plotting function. Seaborn's `barplot` function is built to depict a summary of one quantitative variable against levels of a second, qualitative variable, but can be used here. Step 2 - Prepare a NaN tabular data

In [86]:

```
# Let's drop the column that do not have any NaN/None values
na_counts = sales_data.drop(['Store','Date','Temperature','Fuel_Price','IsHoliday'], axis = 1).isna().sum()
print(na_counts)
```

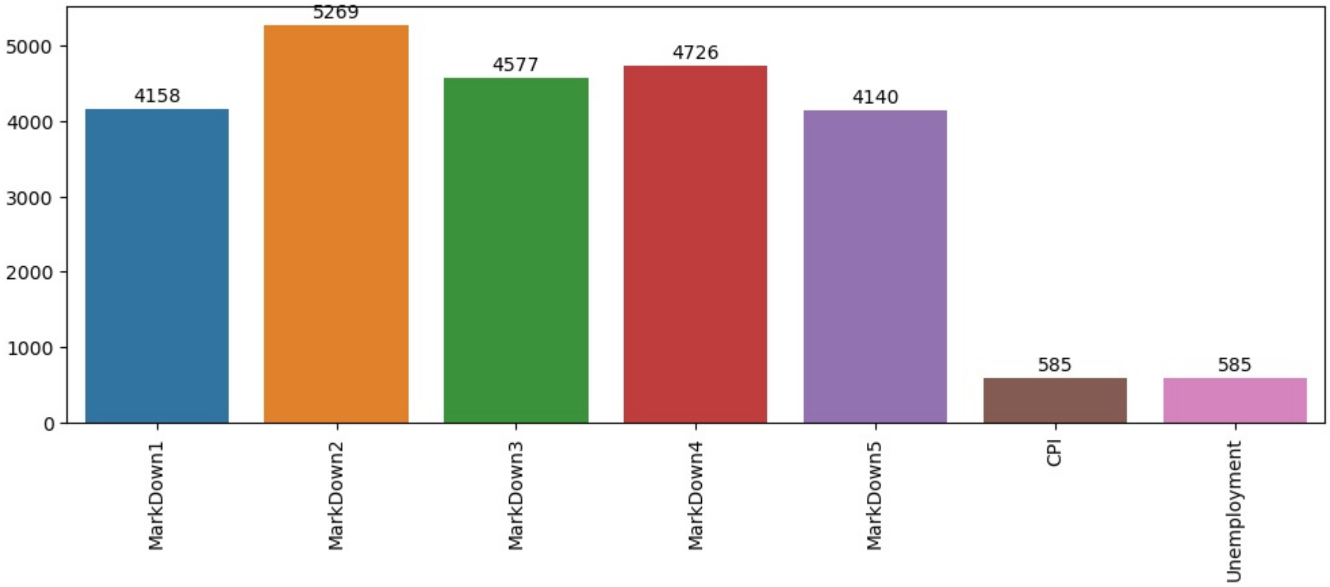
```
MarkDown1       4158
MarkDown2       5269
MarkDown3       4577
MarkDown4       4726
MarkDown5       4140
CPI              585
Unemployment     585
dtype: int64
```

Use `seaborn.barplot()`

Step 3 - Plot the bar chart from the NaN tabular data, and also print values on each bar

In [89]:

```
# The first argument to the function below contains the x-values (column names), the second argument the y-valu
# Refer to the syntax and more example here - https://seaborn.pydata.org/generated/seaborn.barplot.html
sb.barplot(na_counts.index.values, na_counts)

# get the current tick locations and labels
plt.xticks(rotation=90)

# Logic to print value on each bar
for i in range (na_counts.shape[0]):
    count = na_counts[i]

    # Refer here for details of the text() - https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.text.ht
    plt.text(i, count+300, count, ha = 'center', va='top')
```



`Note` - The `seaborn.barplot()` is a useful function to keep in mind if your data is summarized and you still want to build a bar chart. If your data is not yet summarized, however, just use the `countplot` function so that you don't need to do extra summarization work. In addition, you'll see what barplot's main purpose is in the next lesson when we discuss adaptations of univariate plots for plotting bivariate data.

Pie Charts

A `__pie chart__` is a common univariate plot type that is used to depict relative frequencies for levels of a categorical variable. Frequencies in a pie chart are depicted as wedges drawn on a circle: the larger the angle or area, the more common the categorical value taken. Use a Pie chart only when the number of categories is less, and you'd like to see the proportion of each category on a chart.

Guidelines to Use a Pie Chart

If you want to use a pie chart, try to follow certain guidelines:

1. Make sure that your interest is in relative frequencies. Areas should represent parts of a whole, rather than measurements on a second variable (unless that second variable can logically be summed up into some whole).
2. Limit the number of slices plotted. A pie chart works best with two or three slices, though it's also possible to plot with four or five slices as long as the wedge sizes can be distinguished. If you have a lot of categories, or categories that have small proportional representation, consider grouping them together so that fewer wedges are plotted, or use an 'Other' category to handle them.
3. Plot the data systematically. One typical method of plotting a pie chart is to start from the top of the circle, then plot each categorical level clockwise from most frequent to least frequent. If you have three categories and are interested in the comparison of two of them, a common plotting method is to place the two categories of interest on either side of the 12 o'clock direction, with the third category filling in the remaining space at the bottom.

If these guidelines cannot be met, then you should probably make use of a bar chart instead. A bar chart is a safer choice in general. The bar heights are more precisely interpreted than areas or angles, and a bar chart can be displayed more compactly than a pie chart. There's also more flexibility with a bar chart for plotting variables with a lot of levels, like plotting the bars horizontally.

Plot a Pie Chart

`matplotlib.pyplot.pie()`

You can create a pie chart with matplotlib's `matplotlib.pyplot.pie()` function. A basic syntax is:

```
matplotlib.pyplot.pie(x_data, labels, colors, startangle, counterclock, wedgeprops)
```

This function requires that the data be in a summarized form: the primary argument to the function will be the wedge sizes. Refer to the function syntax for details about all other arguments.

`matplotlib.pyplot.axis()`

We also need to know about the `matplotlib.pyplot.axis()` function to set some axis properties. It optionally accepts the axis limits in the form of `xmin, xmax, ymin, ymax floats`, and returns the updated values.

`matplotlib.pyplot.axis(*args, emit=True, **kwargs)`

In the function above, the `*args` represents any number of arguments that you can pass to the function, whereas `**kwargs` stands for keyword arguments, generally passed in the form of a dictionary.

Refer to the function syntax for in-depth details on the all possible values of the arguments.

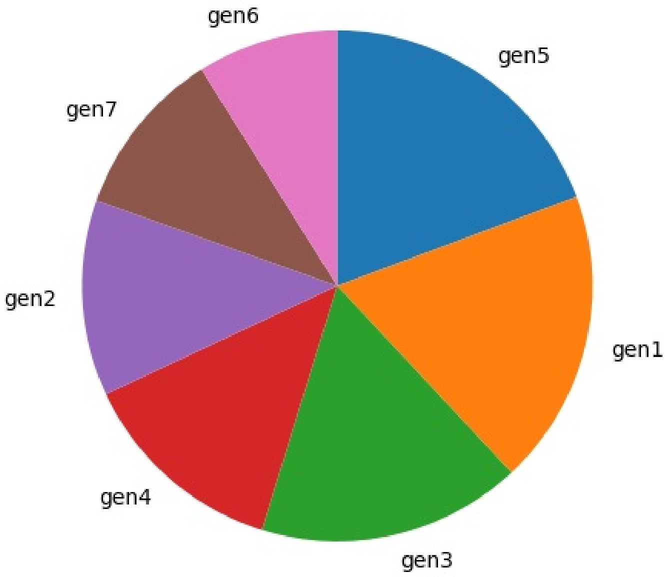
Example 1. Plot a simple Pie chart

In [11]:

```
# Use the same pokemon dataset
sorted_counts = pokemon['generation_id'].value_counts()

plt.pie(sorted_counts, data=sorted_counts.index, startangle=90, counterclock=False, labels=['gen5', 'gen1', 'ge

# We have the used option `Square`.
# Though, you can use either one specified here - https://matplotlib.org/api/_as_gen/matplotlib.pyplot.axis.htm
plt.axis('square')
```

Out[11]:

```
(-1.1063354030102694,
 1.1197837798494124,
 -1.114763917050899,
 1.1113552658087829)
```



To follow the guidelines in the bullet points above, I include the " `startangle = 90` " and " `counterclock = False` " arguments to start the first slice at vertically upwards, and will plot the sorted counts in a clockwise fashion. The axis function call and 'square' argument makes it so that the scaling of the plot is equal on both the x- and y-axes. Without this call, the pie could end up looking oval-shaped, rather than a circle.

TO DO

Did you notice the various arguments in the `plt.pie()` function? Particularly, the `labels = sorted_counts.index` argument represents a list of strings serving as labels for each wedge. In the example above, the labels have used the following list:

In [5]:

```
sorted_counts.index
```

Out[5]:

```
Int64Index([5, 1, 3, 4, 2, 7, 6], dtype='int64')
```

Donut Plot

A sister plot to the pie chart is the donut plot. It's just like a pie chart, except that there's a hole in the center of the plot. Perceptually, there's not much difference between a donut plot and a pie chart, and donut plots should be used with the same guidelines as a pie chart. Aesthetics might be one of the reasons why you would choose one or the other. For instance, you might see statistics reported in the hole of a donut plot to better make use of available space.

To create a donut plot, you can add a `wedgeprops` argument to the `pie` function call. By default, the radius of the pie (circle) is 1; setting the wedges' width property to less than 1 removes coloring from the center of the circle.

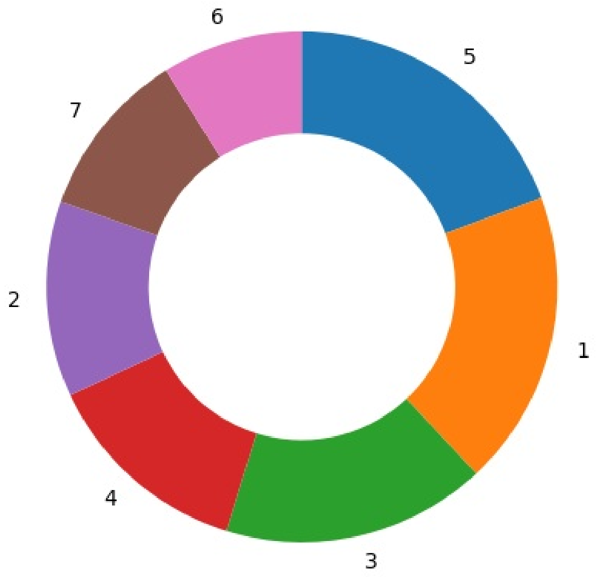
Example 2. Plot a simple Donut plot

In [18]:

```
sorted_counts = pokemon['generation_id'].value_counts()

plt.pie(sorted_counts, labels = sorted_counts.index, startangle = 90,
        counterclock = False, wedgeprops = {'width' : 0.4});
plt.axis('square')
```

Out[18]:

```
(-1.1063354030102694,
 1.1197837798494124,
 -1.114763917050899,
 1.1113552658087829)
```



Histograms

A histogram is used to plot the distribution of a numeric variable. It's the quantitative version of the bar chart. However, rather than plot one bar for each unique numeric value, values are grouped into continuous bins, and one bar for each bin is plotted to depict the number. You can use either Matplotlib or Seaborn to plot the histograms. There is a mild variation in the specifics, such as plotting gaussian-estimation line along with bars in Seabron's distplot(), and the arguments that you can use in either case.

Matplotlib.pyplot.hist()

You can use the default settings for matplotlib's `hist()` function to plot a histogram with 10 bins:

Example 1. Plot a default histogram

In [19]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline

pokemon = pd.read_csv('pokemon.csv')
print(pokemon.shape)
pokemon.head(10)
```

```
(807, 14)
```

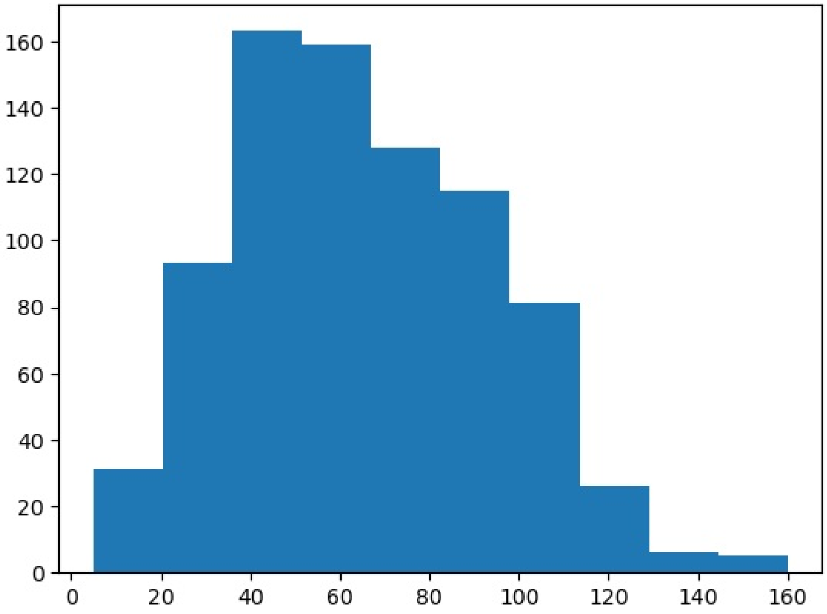
Out[19]:

	id	species	generation_id	height	weight	base_experience	type_1	type_2	hp	attack	defense	speed	special-attack	special-defense
0	1	bulbasaur	1	0.7	6.9	64	grass	poison	45	49	49	45	65	65
1	2	ivysaur	1	1.0	13.0	142	grass	poison	60	62	63	60	80	80
2	3	venusaur	1	2.0	100.0	236	grass	poison	80	82	83	80	100	100
3	4	charmander	1	0.6	8.5	62	fire	NaN	39	52	43	65	60	50
4	5	charmeleon	1	1.1	19.0	142	fire	NaN	58	64	58	80	80	65
5	6	charizard	1	1.7	90.5	240	fire	flying	78	84	78	100	109	85
6	7	squirtle	1	0.5	9.0	63	water	NaN	44	48	65	43	50	64
7	8	wartortle	1	1.0	22.5	142	water	NaN	59	63	80	58	65	80
8	9	blastoise	1	1.6	85.5	239	water	NaN	79	83	100	78	85	105
9	10	caterpie	1	0.3	2.9	39	bug	NaN	45	30	35	45	20	20

Plot a default histogram as shown below:

In [21]:

```python
# We have intentionally not put a semicolon at the end of the statement below to see the bar-width
plt.hist(data=pokemon, x='speed');
```



You can see a non-uniform distribution of data points in different bins.

Overall, a generally bimodal distribution is observed (one with two peaks or humps). The direct adjacency of the bars in the histogram, in contrast to the separated bars in a bar chart, emphasizes the fact that the data takes on a continuous range of values. When a data value is on a bin edge, it is counted in the bin to its right. The exception is the rightmost bin edge, which places data values equal to the uppermost limit into the right-most bin (to the upper limit's left).

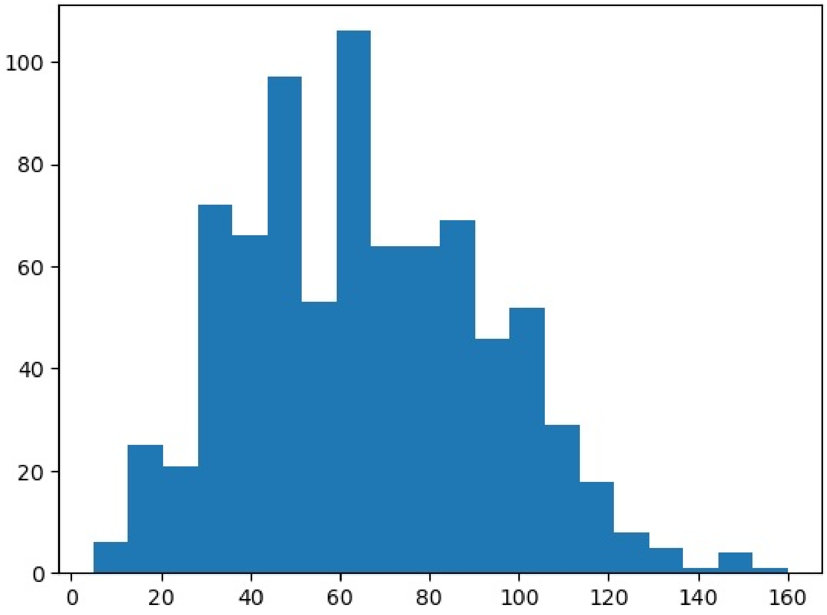
By default, the `hist` function divides the data into 10 bins, based on the range of values taken. In almost every case, we will want to change these settings. Usually, having only ten bins is too few to really understand the distribution of the data. And the default tick marks are often not on nice, 'round' values that make the ranges taken by each bin easy to interpret.

Wouldn't it be better if I said "between 0 and 2.5" instead of "between about 0 and 2.5", and "from 2.5 to 5" instead of "from about 2.5 to 5" above?

You can use descriptive statistics (e.g. via `dataframe['column'].describe()`) to gauge what minimum and maximum bin limits might be appropriate for the plot. These bin edges can be set using numpy's arange function:

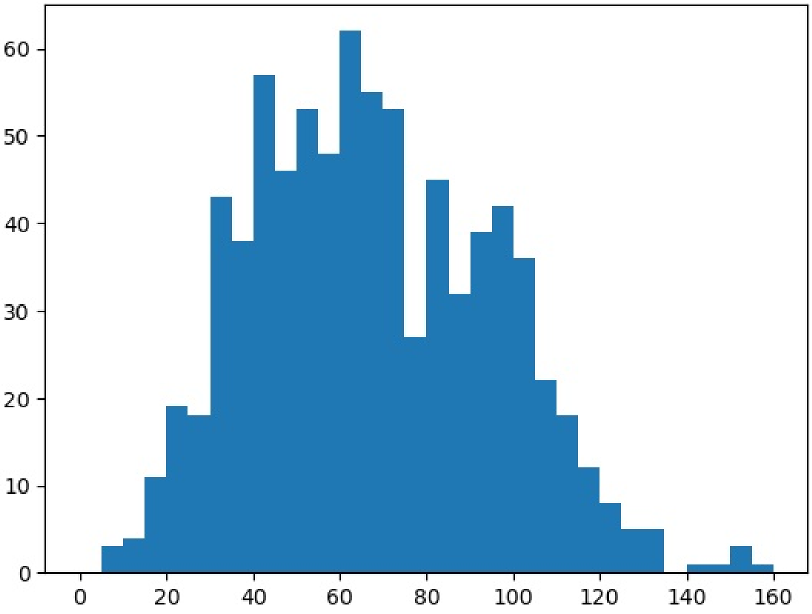
Example 2. Histogram with fixed number of bins

In [24]:
```
plt.hist(data=pokemon, x='speed', bins=20);
```



Example 3. Histogram with dynamic number of bins

In [34]:
```
# Create bins with step-size 5
bins = np.arange(0, pokemon['speed'].max()+5, 5)

plt.hist(data=pokemon, x='speed', bins=bins);
```



The first argument to `arange` is the leftmost bin edge, the second argument the upper limit, and the third argument the bin width. Note that even though I've specified the "max" value in the second argument, I've added a "+5" (the bin width). That is because `arange` will only return values that are strictly less than the upper limit. Adding in "+5" is a safety measure to ensure that the rightmost bin edge is at least the maximum data value, so that all of the data points are plotted. The leftmost bin is set as a hardcoded value to get a nice, interpretable value, though you could use functions like numpy's `around` if you wanted to approach that end programmatically.

Alternative Approach - Seaborn's `displot()`

This function can also plot histograms, as similar to the `pyploy.hist()` function, and is integrated with other univariate plotting functions. This is in contrast to our ability to specify a data source and column as separate arguments, like we've seen with and `countplot` and `hist`.

The basic syntax is:

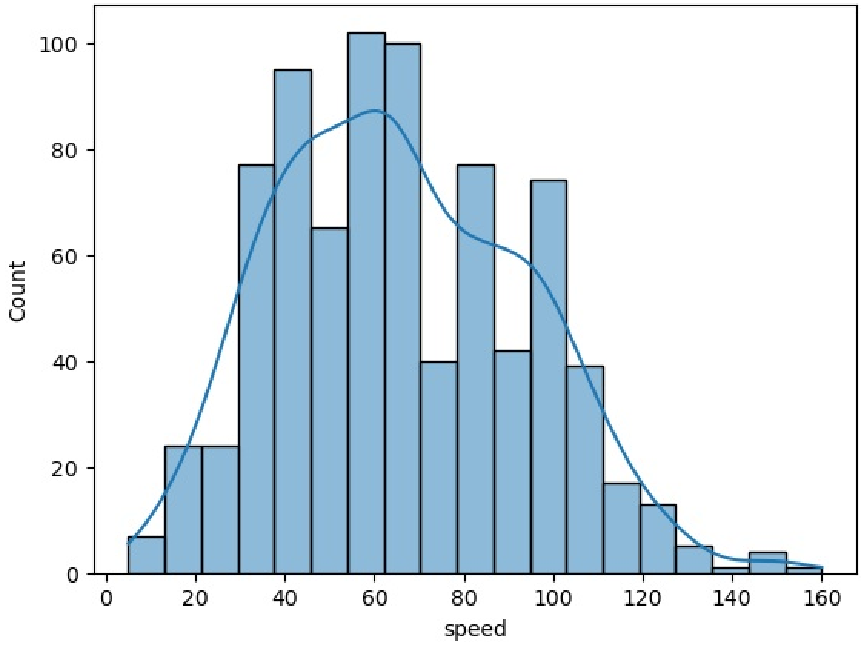
```
seaborn.distplot(Series, bins, kde, hist_kws)
```

Let's see the sample usage of the arguments mentioned above. However, there are many other arguments that you can explore in the syntax definition.

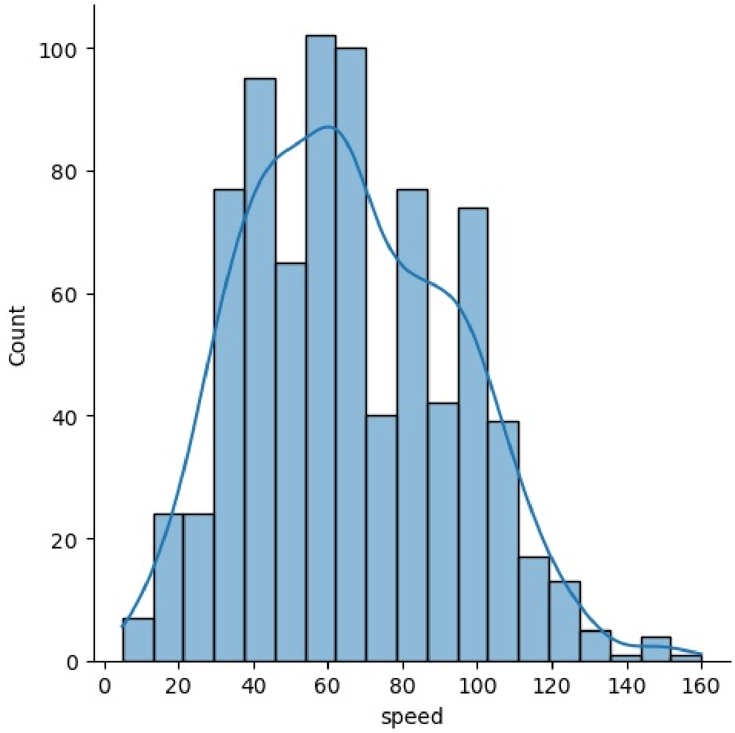
> Note: From the Seaborn v0.11.0 onwards, this function is deprecated and will be removed in a future version. You can use either of the following two functions: `displot()` or `histplot()` to plot histograms using Seaborn.

Example 4. Plot the similar histogram with Seaborn's `displot()`

```
In [47]: sb.histplot(pokemon['speed'], kde=True);
```



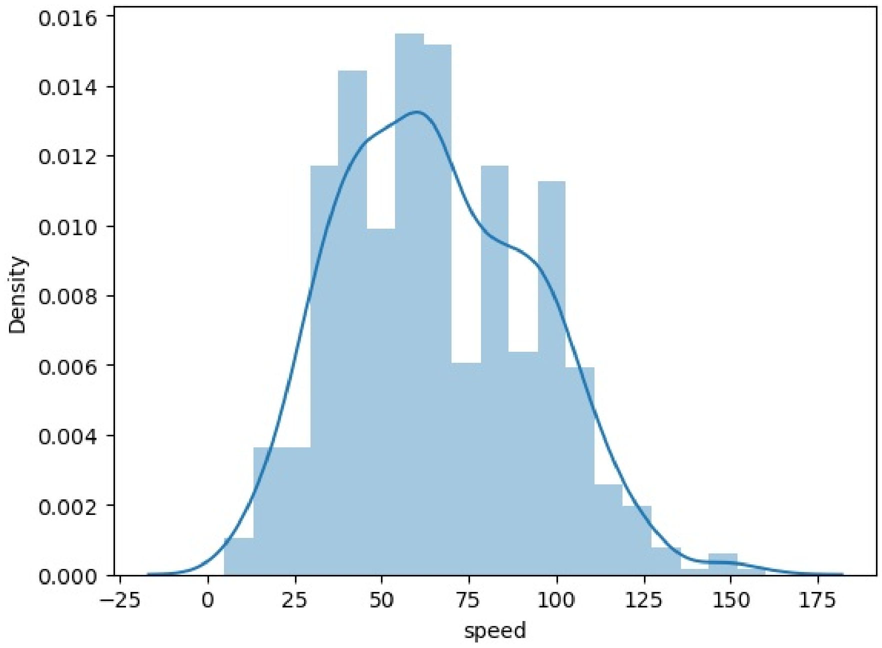
```
In [46]: sb.displot(pokemon['speed'], kde=True);
```



```
In [50]: sb.distplot(pokemon['speed'])
```

```
C:\Users\Isaac\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprec
ated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure
-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

```
Out[50]: <AxesSubplot:xlabel='speed', ylabel='Density'>
```



The `distplot` function has built-in rules for specifying histogram bins, and by default plots a curve depicting the kernel density estimate (KDE) on top of the data. The vertical axis is based on the KDE, rather than the histogram: you shouldn't expect the total heights of the bars to equal 1, but the area under the curve should equal 1. If you want to learn more about KDEs, check out the extra page at the end of the lesson.

Despite the fact that the default bin-selection formula used by `distplot` might be better than the choice of ten bins that `.hist` uses, you'll still want to do some tweaking to align the bins to 'round' values. You can use other parameter settings to plot just the histogram and specify the bins like before:

```
In [55]: bin_edges = np.arange(0, df['num_var'].max()+1, 1)
         sb.distplot(df['num_var'], bins = bin_edges, kde = False,
                     hist_kws = {'alpha' : 1})
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9716\2070304994.py in <module>
----> 1 bin_edges = np.arange(0, df['num_var'].max()+1, 1)
      2 sb.distplot(df['num_var'], bins = bin_edges, kde = False,
      3             hist_kws = {'alpha' : 1})

NameError: name 'df' is not defined
```

Plot two histograms side-by-side

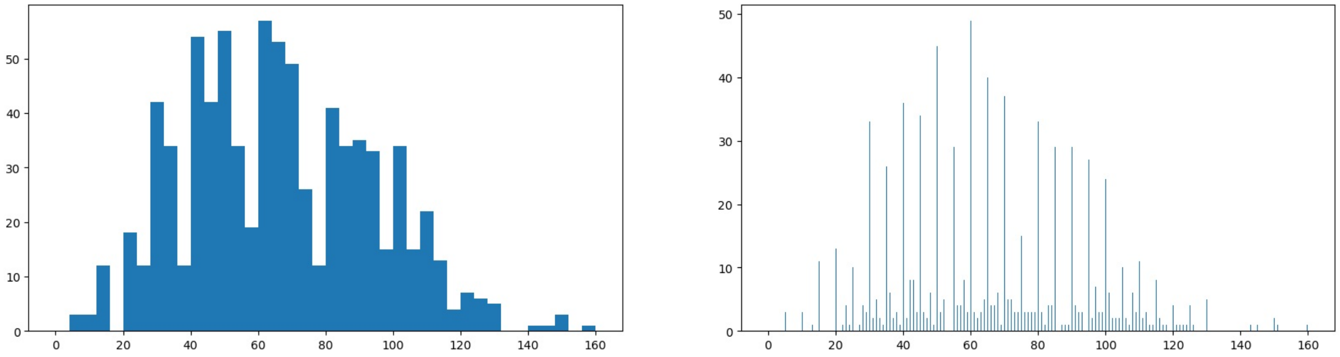
When creating histograms, it's useful to play around with different bin widths to see what represents the data best. Too many bins, and you may see too much noise that interferes with the identification of the underlying signal. Too few bins, and you may not be able to see the true signal in the first place.

Let's see a new example demonstrating a few new functions, `pyplot.subplot()` and `pyplot.figure()`. We will learn more in the upcoming concepts.

Example 5. Plot two histograms side-by-side

In [56]:

```
# Resize the chart, and have two plots side-by-side
# Set a larger figure size for subplots
plt.figure(figsize = [20, 5])

# histogram on left, example of too-large bin size
# 1 row, 2 cols, subplot 1
plt.subplot(1, 2, 1)
bins = np.arange(0, pokemon['speed'].max()+4, 4)
plt.hist(data = pokemon, x = 'speed', bins = bins);

# histogram on right, example of too-small bin size
plt.subplot(1, 2, 2) # 1 row, 2 cols, subplot 2
bins = np.arange(0, pokemon['speed'].max()+1/4, 1/4)
plt.hist(data = pokemon, x = 'speed', bins = bins);
```



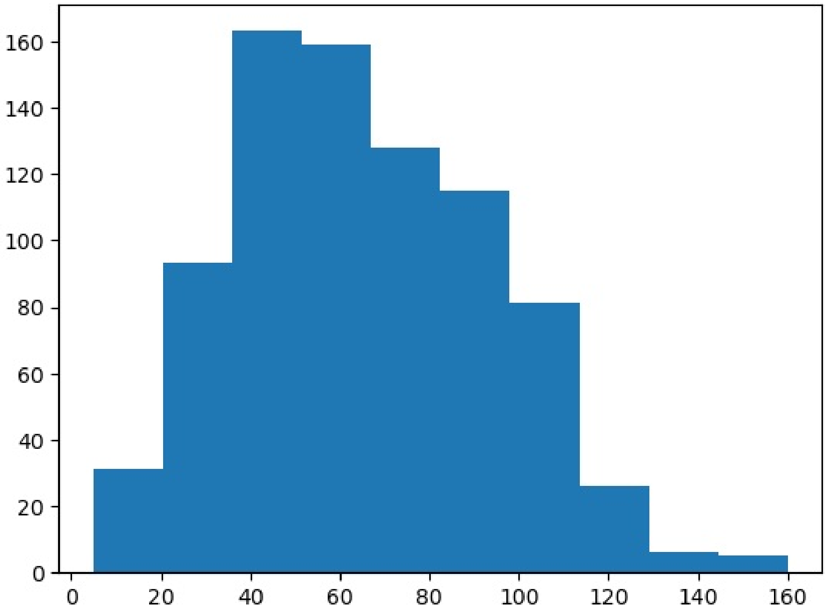
Figures, Axes, and Subplots

At this point, you've seen and had some practice with some basic plotting functions using matplotlib and seaborn. The previous page introduced something a little bit new: creating two side-by-side plots through the use of matplotlib's subplot() function. If you have any questions about how that or the figure() function worked, then read on. This page will discuss the basic structure of visualizations using matplotlib and how subplots work in that structure.

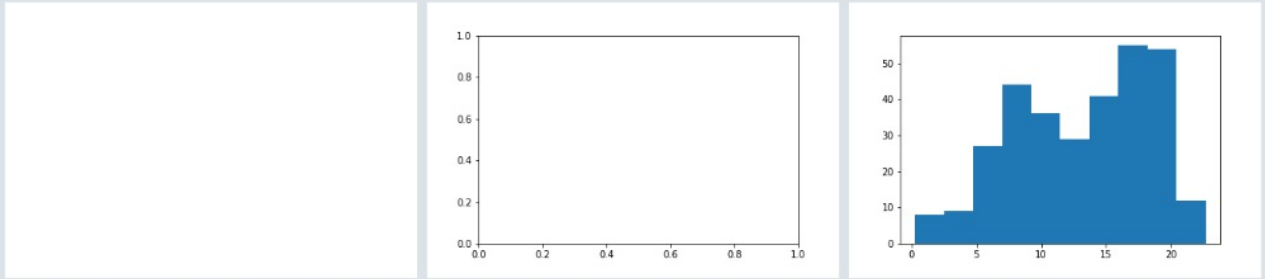
The base of visualization in matplotlib is a Figure object. Contained within each Figure will be one or more Axes objects, each Axes object containing a number of other elements that represent each plot. In the earliest examples, these objects have been created implicitly. Let's say that the following expression is run inside a Jupyter notebook to create a histogram:

In [57]:

```
plt.hist(data=pokemon, x='speed');
```



Since we don't have a Figure area to plot inside, Python first creates a Figure object. And since the Figure doesn't start with any Axes to draw the histogram onto, an Axes object is created inside the Figure. Finally, the histogram is drawn within that Axes.



Create a Figure,

add an Axes to the Figure...

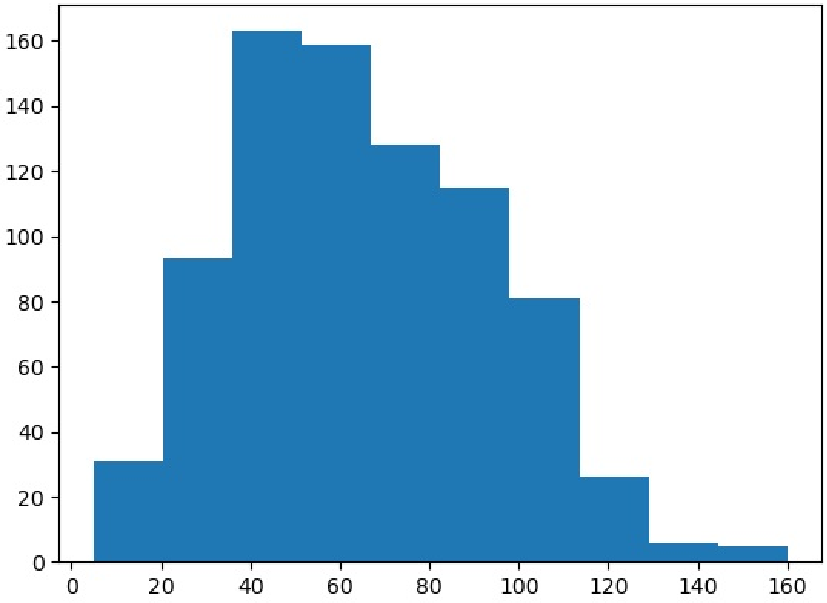
then plot the data inside that Axes!

This hierarchy of objects is useful to know about so that we can take more control over the layout and aesthetics of our plots. One alternative way we could have created the histogram is to explicitly set up the Figure and Axes like this:

Example 2. Demonstrate figure.add_axes() and axes.hist()

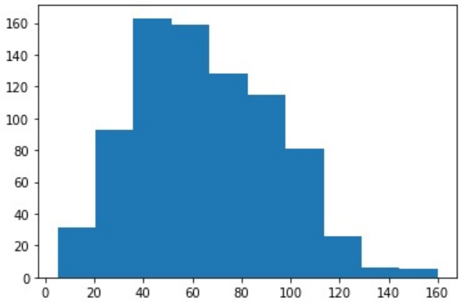
In [58]:

```
# Create a new figure
fig = plt.figure()

# The argument of add_axes represents the dimensions [left, bottom, width, height] of the new axes.
# All quantities are in fractions of figure width and height.
ax = fig.add_axes([.125, .125, .775, .755])
ax.hist(data=pokemon, x='speed');
```



`figure()` creates a new Figure object, a reference to which has been stored in the variable `fig`. One of the Figure methods is

`.add_axes()`, which creates a new Axes object in the Figure. The method requires one list as argument specifying the dimensions of the Axes: the first two elements of the list indicate the position of the lower-left hand corner of the Axes (in this case one quarter of the way from the lower-left corner of the Figure) and the last two elements specifying the Axes width and height, respectively. We refer to the Axes in the variable `ax`. Finally, we use the Axes method `.hist()` just like we did before with `plt.hist()`.



This plot is just like the first histogram on the Histograms page.

To use Axes objects with seaborn, seaborn functions usually have an "ax" parameter to specify upon which Axes a plot will be drawn.

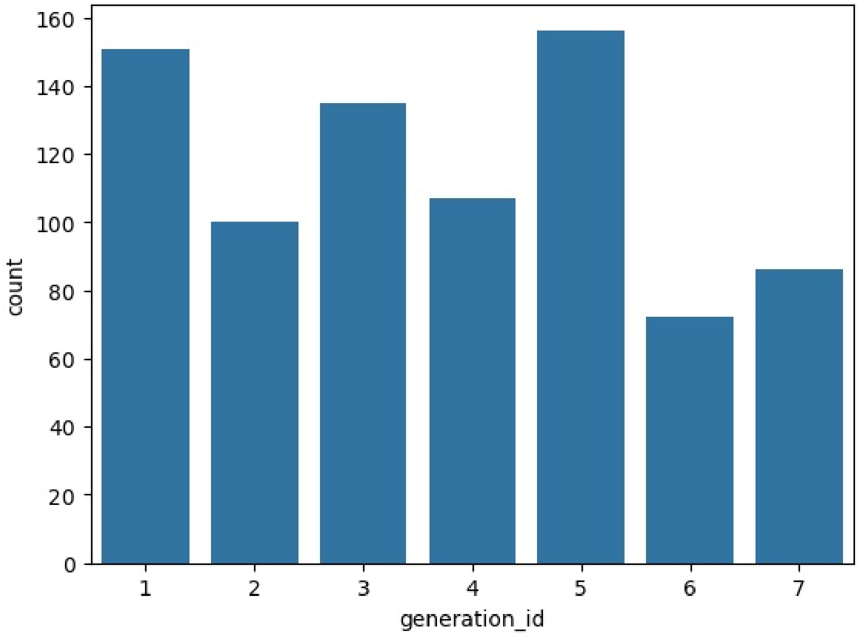
Example 2. Use axes with seaborn.countplot()

In [59]:

```
fig = plt.figure()
ax = fig.add_axes([.125, .125, .775, .755])
base_color = sb.color_palette()[0]
sb.countplot(data = pokemon, x = 'generation_id', color = base_color, ax = ax)
```

Out[59]:

```
<Axes:xlabel='generation_id', ylabel='count'>
```

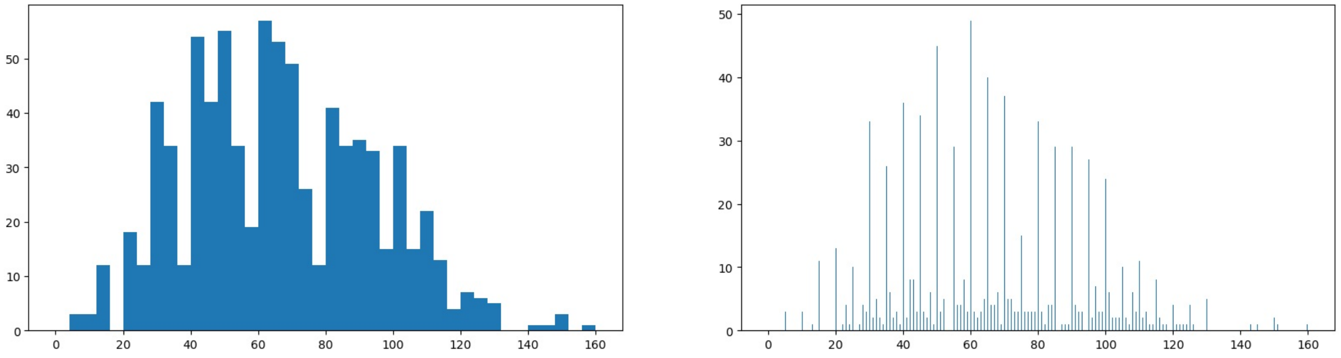


In the above two cases, there was no purpose to explicitly go through the Figure and Axes creation steps. And indeed, in most cases, you can just use the basic matplotlib and seaborn functions as is. Each function targets a Figure or Axes, and they'll automatically target the most recent Figure or Axes worked with. As an example of this, let's review in detail how `subplot()` was used on the Histograms page:

Example 3. Sub-plots

In [60]:

```
# Resize the chart, and have two plots side-by-side
# set a larger figure size for subplots
plt.figure(figsize = [20, 5])

# histogram on left, example of too-large bin size
# 1 row, 2 cols, subplot 1
plt.subplot(1, 2, 1)
bins = np.arange(0, pokemon['speed'].max()+4, 4)
plt.hist(data = pokemon, x = 'speed', bins = bins);

# histogram on right, example of too-small bin size
plt.subplot(1, 2, 2) # 1 row, 2 cols, subplot 2
bins = np.arange(0, pokemon['speed'].max()+1/4, 1/4)
plt.hist(data = pokemon, x = 'speed', bins = bins);
```



First of all, plt.figure(figsize = [20, 5])creates a new Figure, with the "figsize" argument setting the width and height of the overall figure to 20 inches by 5 inches, respectively. Even if we don't assign any variable to return the function's output, Python will still implicitly know that further plotting calls that need a Figure will refer to that Figure as the active one.

Then, `plt.subplot(1, 2, 1)` creates a new Axes in our Figure, its size determined by the `subplot()` function arguments. The first two arguments says to divide the figure into one row and two columns, and the third argument says to create a new Axes in the first slot. Slots are numbered from left to right in rows from top to bottom. Note in particular that the index numbers start at 1 (rather than the usual Python indexing starting from 0). (You'll see the indexing a little better in the example at the end of the page.) Again, Python will implicitly set that Axes as the current Axes, so when the `plt.hist()` call comes, the histogram is plotted in the left-side subplot.

Finally, `plt.subplot(1, 2, 2)` creates a new Axes in the second subplot slot, and sets that one as the current Axes. Thus, when the next plt.hist() call comes, the histogram gets drawn in the right-side subplot.

Additional Techniques

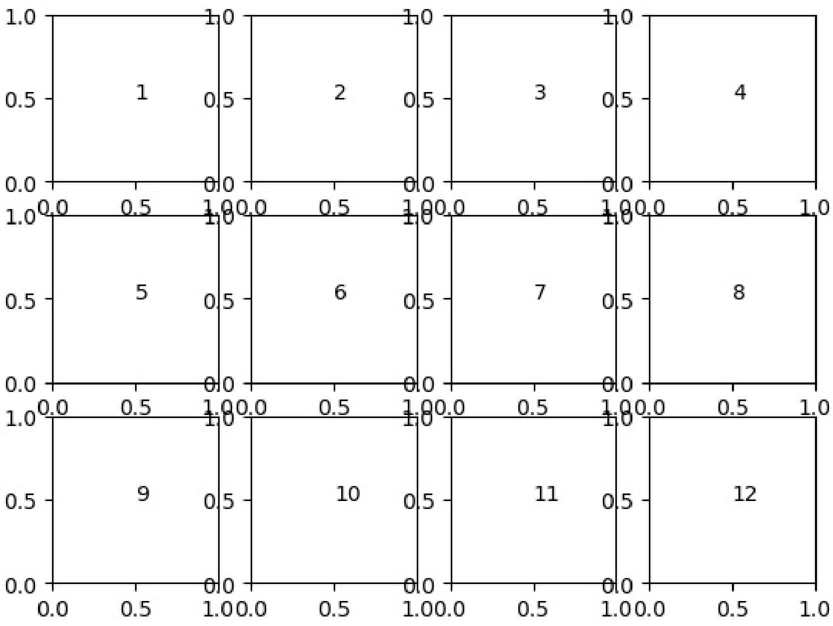
To close this page, we'll quickly run through a few other ways of dealing with Axes and subplots. The techniques above should suffice for basic plot creation, but you might want to keep the following in the back of your mind as additional tools to break out as needed.

If you don't assign Axes objects as they're created, you can retrieve the current Axes using **`ax = plt.gca()`**, or you can get a list of all Axes in a Figure `fig` by using **`axes = fig.get_axes()`**. As for creating subplots, you can use **`fig.add_subplot()`** in the same way as `plt.subplot()` above. If you already know that you're going to be creating a bunch of subplots, you can use the **`plt.subplots()`** function:

Example 4. Demonstrate pyplot.sca() and pyplot.text() to generate a grid of subplots

In [61]:

```
fig, axes = plt.subplots(3, 4) # grid of 3x4 subplots
axes = axes.flatten() # reshape from 3x4 array into 12-element vector
for i in range(12):
    plt.sca(axes[i]) # set the current Axes
    plt.text(0.5, 0.5, i+1) # print conventional subplot index number to middle of Axes
```



As a special note for the text, the Axes limits are [0,1] on each Axes by default, and we increment the iterator counter i by 1 to get the subplot index, if we were creating the subplots through subplot(). (Reference: plt.sca(), plt.text())

Choosing a Plot for Discrete Data

If you want to plot a **discrete quantitative variable**, it is possible to select either a histogram or a bar chart to depict the data.

- Here, the **discrete** means non-continuous values. In general, a discrete variable can be assigned to any of the limited (countable) set of values from a given set/range, for example, the number of family members, number of football matches in a tournament, number of departments in a university.
- The **quantitative** term shows that it is the outcome of the measurement of a quantity.

The histogram is the most immediate choice since the data is numeric, but there's one particular consideration to make regarding the bin edges. Since data points fall on set values (bar-width), it can help to reduce ambiguity by putting bin edges between the actual values taken by the data.

An example describing the ambiguity

For example, assume a given bar falls in a range [10-20], and there is an observation with value 20. This observation will lie on the *next* bar because the given range [10-20] does not include the upper limit 20. Therefore, your readers may not know that values on bin edges end up in the bin to their right, so this can bring potential confusion when they interpret the plot.

Compare the two visualizations of 100 random die rolls below (in `die_rolls`), with bin edges *falling on* the observation values in the left subplot, and bin edges *in between* the observation values in the right subplot.

Preparatory Step

In [64]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline

die_rolls = pd.read_csv('die_rolls.csv')

# A fair dice has six-faces having numbers [1-6].
# There are 100 dices, and two trials were conducted.
# In each trial, all 100 dices were rolled down, and the outcome [1-6] was recorded.
# The `Sum` column represents the sum of the outcomes in the two trials, for each given dice.
die_rolls.head(10)
```

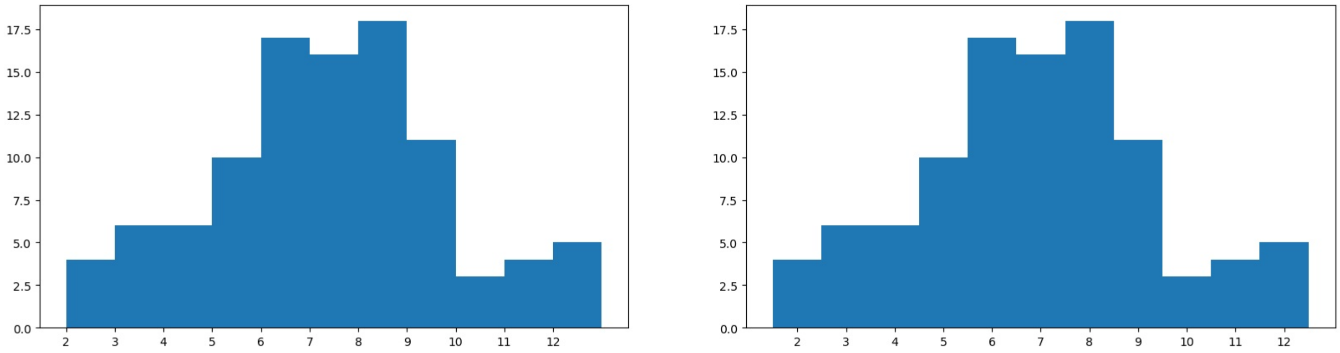
Out[64]:

	Dice	Trial 1	Trial 2	Sum
0	1	4	1	5
1	2	4	5	9
2	3	2	6	8
3	4	6	3	9
4	5	3	6	9
5	6	6	6	12
6	7	3	3	6
7	8	3	2	5
8	9	2	6	8
9	10	6	6	12

Example 1. Shifting the edges of the bars can remove ambiguity in the case of Discrete data

In [65]:

```
plt.figure(figsize = [20, 5])

# Histogram on the left, bin edges on integers
plt.subplot(1, 2, 1)
bin_edges = np.arange(2, 12+1.1, 1) # note `+1.1`, see below
plt.hist(data=die_rolls, x='Sum', bins = bin_edges);
plt.xticks(np.arange(2, 12+1, 1));


# Histogram on the right, bin edges between integers
plt.subplot(1, 2, 2)
bin_edges = np.arange(1.5, 12.5+1, 1)
plt.hist(data=die_rolls, x='Sum', bins = bin_edges);
plt.xticks(np.arange(2, 12+1, 1));
```



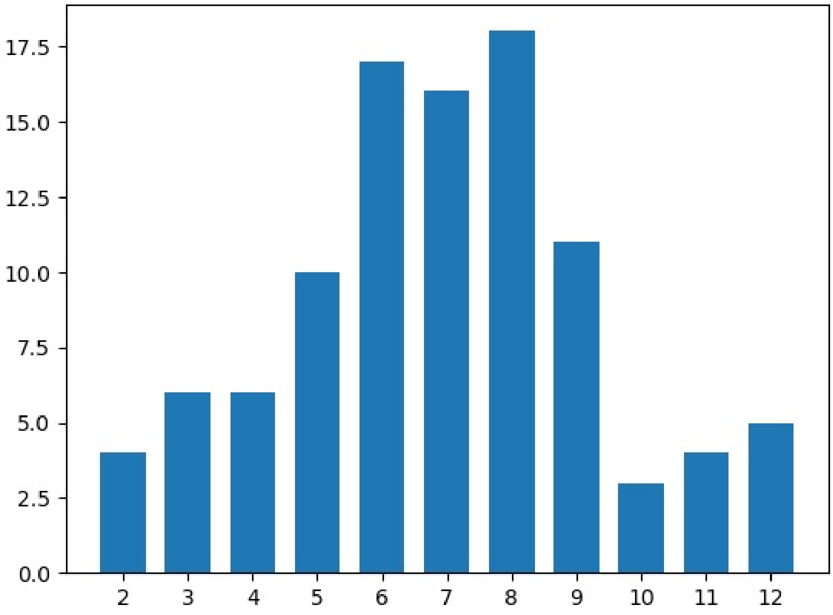
You'll notice for the left histogram, in a deviation from the examples that have come before, I've added 1.1 to the max value (12) for setting the bin edges, rather than just the desired bin width of 1. Recall that data that is equal to the rightmost bin edge gets lumped in to the last bin. This presents a potential problem in perception when a lot of data points take the maximum value, and so is especially relevant when the data takes on discrete values. The 1.1 adds an additional bin to the end to store the die rolls of value 12 alone, to avoid having the last bar catch both 11 and 12.

Alternatively to the histogram, consider if a bar chart with non-connected bins might serve your purposes better. The plot below takes the code from before, but adds the "rwidth" parameter to set the proportion of the bin widths that will be filled by each histogram bar.

Example 2. Making gaps between individual bars

In [66]:

```
bin_edges = np.arange(1.5, 12.5+1, 1)
plt.hist(data=die_rolls, x='Sum', bins = bin_edges, rwidth = 0.7)
plt.xticks(np.arange(2, 12+1, 1));
```

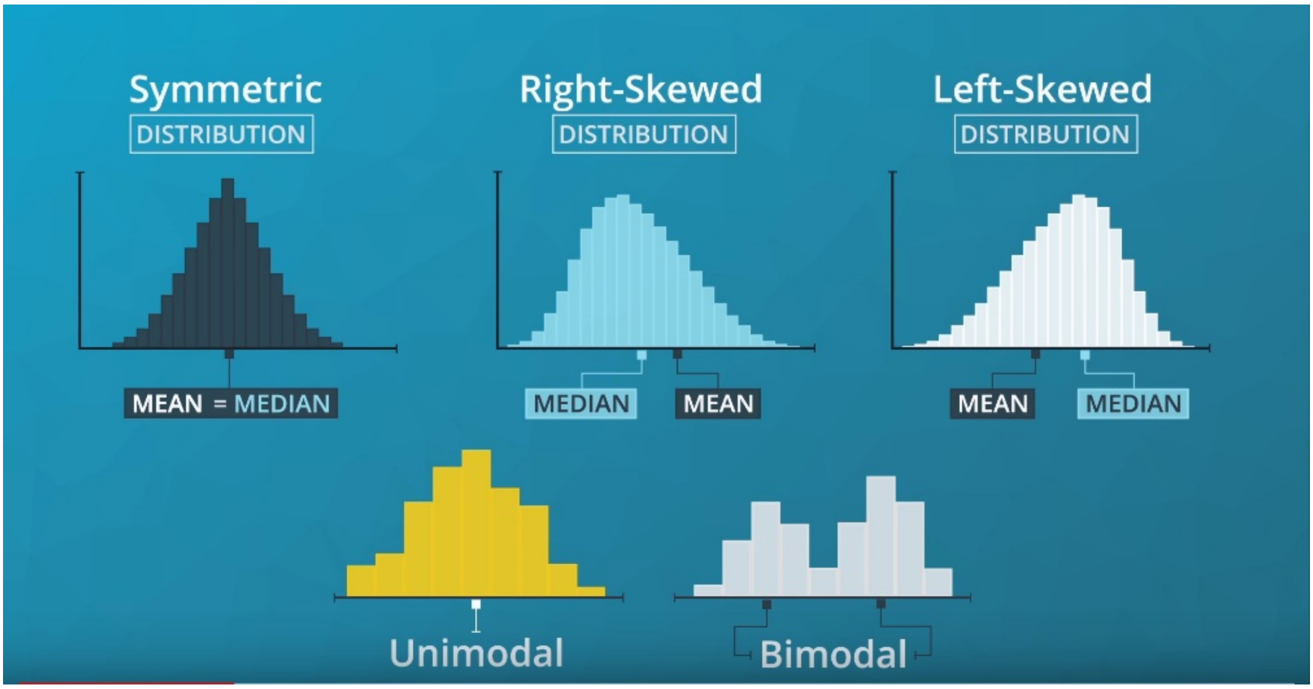


By adding gaps between bars, you emphasize the fact that the data is discrete in value. On the other hand, plotting your quantitative data in this manner might cause it to be interpreted as ordinal-type data, which can have an effect on overall perception.

For continuous numeric data, you should not make use of the "rwidth" parameter, since the gaps imply discreteness of value. As another caution, it might be tempting to use seaborn's `countplot` function to plot the distribution of a discrete numeric variable as bars. Be careful about doing this, since each unique numeric value will get a bar, regardless of the spacing in values between bars. (For example, if the unique values were {1, 2, 4, 5}, missing 3, `countplot` would only plot four bars, with the bars for 2 and 4 right next to one another.) Also, even if your data is technically discrete numeric, you should probably not consider either of the variants depicted on this page unless the number of unique values is small enough to allow for the half-unit shift or discrete bars to be interpretable. If you have a large number of unique values over a large enough range, it's better to stick with the standard histogram than risk interpretability issues.

While you might justify plotting discrete numeric data using a bar chart, you'll be less apt to justify the opposite: plotting ordinal data as a histogram. The space between bars in a bar chart helps to remind the reader that values are not contiguous in an 'interval'-type fashion: only that there is an order in levels. With that space removed as in a histogram, it's harder to remember this important bit of interpretation.

Descriptive Statistics, Outliers, and Axis Limits



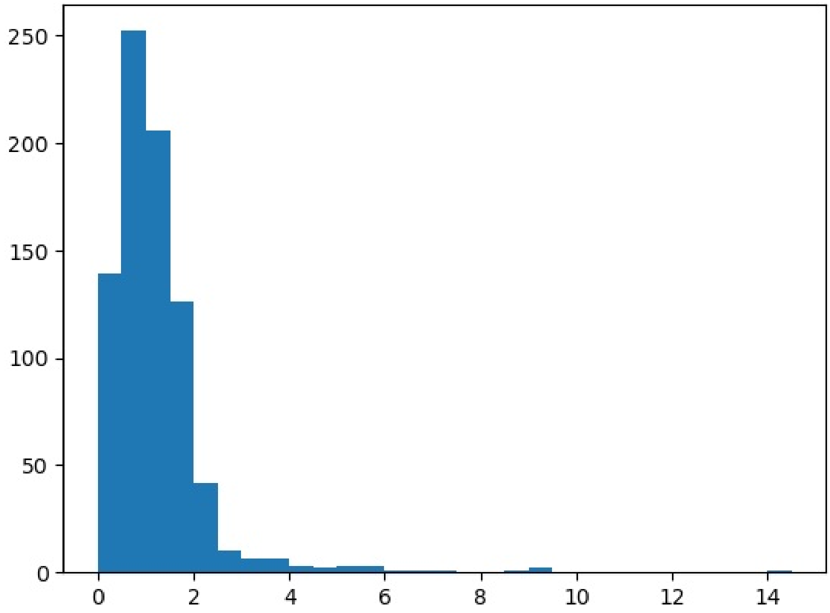
As you create your plots and perform your exploration, make sure that you pay attention to what the plots tell you that go beyond just the basic descriptive statistics. Note any aspects of the data like the number of modes and skew, and note the presence of outliers in the data for further investigation.

Related to the latter point, you might need to change the limits or scale of what is plotted to take a closer look at the underlying patterns in the data. Let's see a few examples.

Example 1. Plot the histogram from the data having a skewed distribution of values

In [67]:

```
# TO DO: Necessary import

# Load the data, and see the height column
pokemon = pd.read_csv('pokemon.csv')
pokemon.head(10)

# Get the ticks for bins between [0-15], at an interval of 0.5
bins = np.arange(0, pokemon['height'].max()+0.5, 0.5)

# Plot the histogram for the height column
plt.hist(data=pokemon, x='height', bins=bins);
```



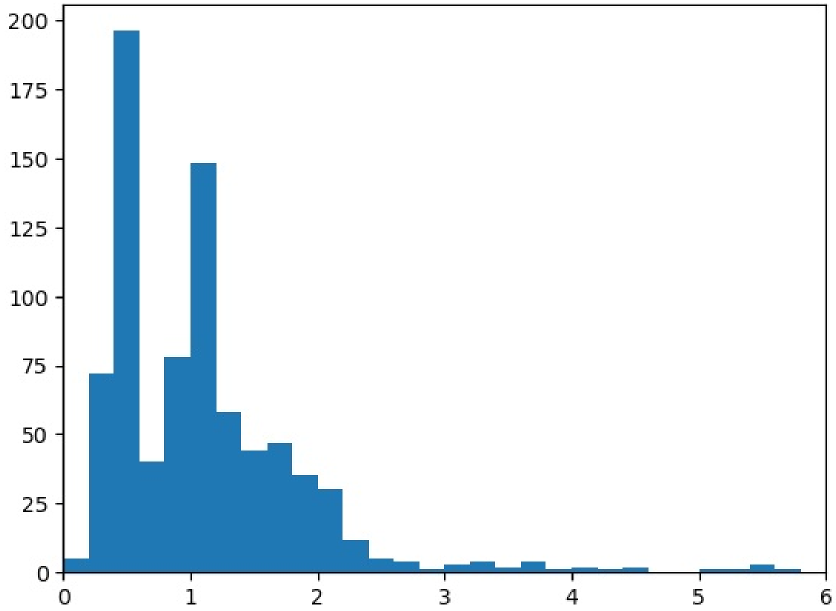
This page covers the topic of axis limits; and the next topic is scales and transformations. In order to change a histogram's axis limits, you can add a Matplotlib `xlim()` call to your code. The function takes a tuple of two numbers specifying the upper and lower bounds of the x-axis range. See the example below.

Example 2. Plot the histogram with a changed axis limit.

In [68]:

```
# Get the ticks for bins between [0-15], at an interval of 0.5
bins = np.arange(0, pokemon['height'].max()+0.2, 0.2)
```

```
plt.hist(data=pokemon, x='height', bins=bins);

# Set the upper and lower bounds of the bins that are displayed in the plot
# Refer here for more information - https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.xlim.html
# The argument represent a tuple of the new x-axis limits.
plt.xlim((0,6));
```



TO DO: Plot the above two graphs in a single figure of size 20 x 5 inches, side-by-side.

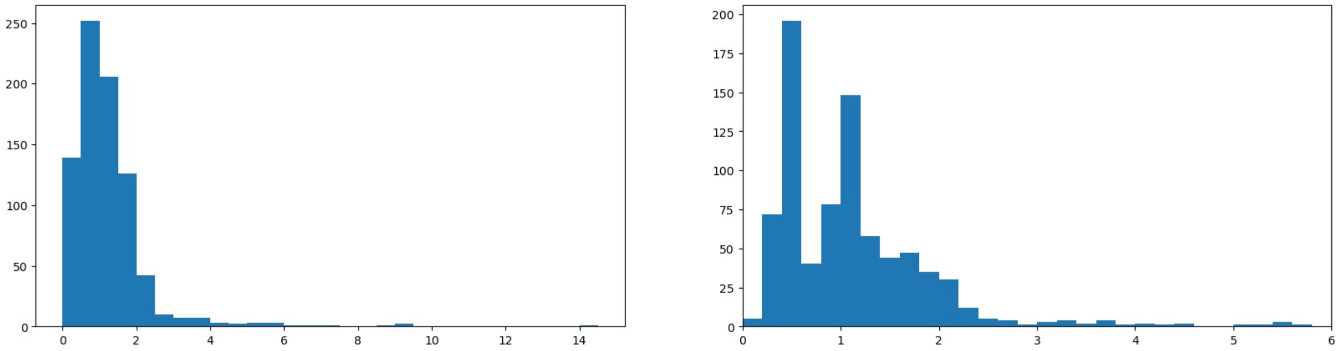
Hint - Use the steps below:

1. Define the figure size, using `matplotlib.pyplot.figure(figsize = [float, float])`.
2. Add a subplot using `matplotlib.pyplot.subplot(int, int, index)` for the left-graph to the current figure. Then, define the left-graph.
3. Similarly, add a subplot for the right-graph to the current figure. Then, define the right-graph.

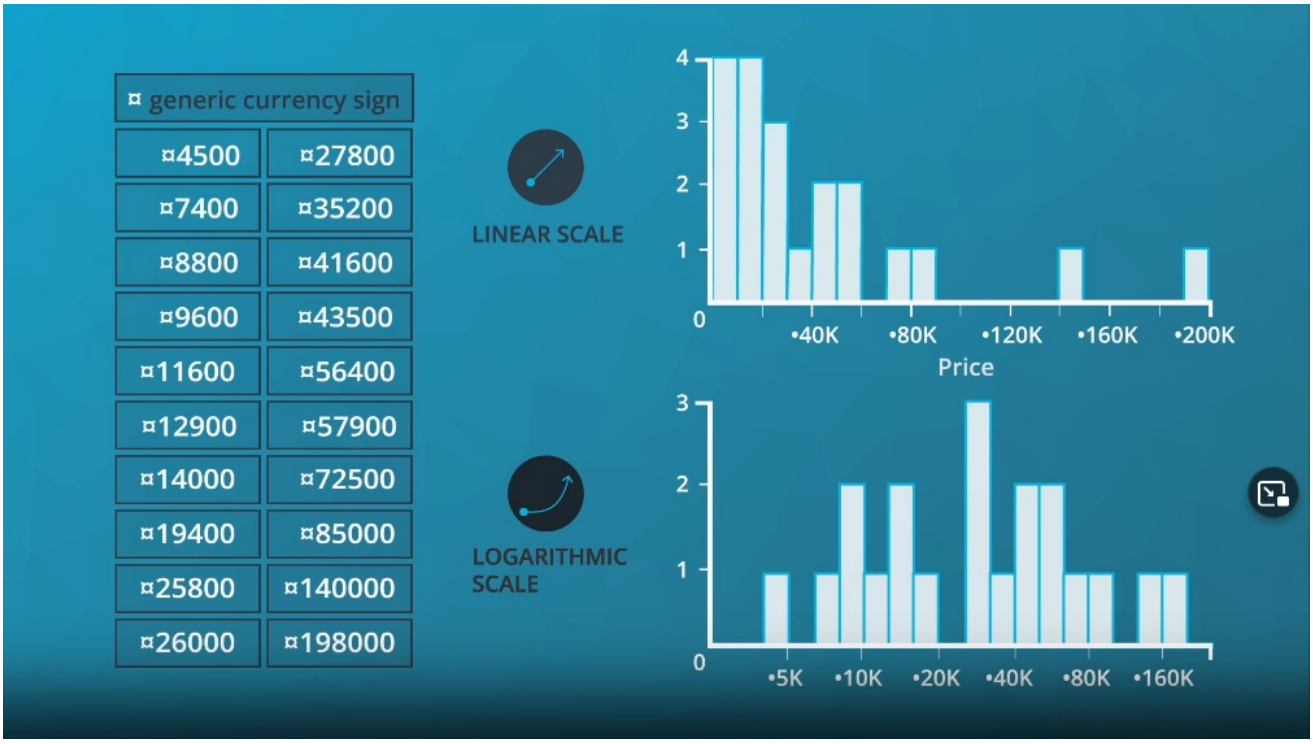
The expected output is shown below:

```
In [69]:
# Define the figure size
plt.figure(figsize = [20, 5])

# histogram on left: full data
plt.subplot(1, 2, 1)
bin_edges = np.arange(0, pokemon['height'].max()+0.5, 0.5)
plt.hist(data=pokemon, x='height', bins = bin_edges)

# histogram on right: focus in on bulk of data < 6
plt.subplot(1, 2, 2)
bin_edges = np.arange(0, pokemon['height'].max()+0.2, 0.2)
plt.hist(data=pokemon, x='height', bins = bin_edges)
plt.xlim(0, 6) # could also be called as plt.xlim((0, 6))
```

```
Out[69]: (0.0, 6.0)
```



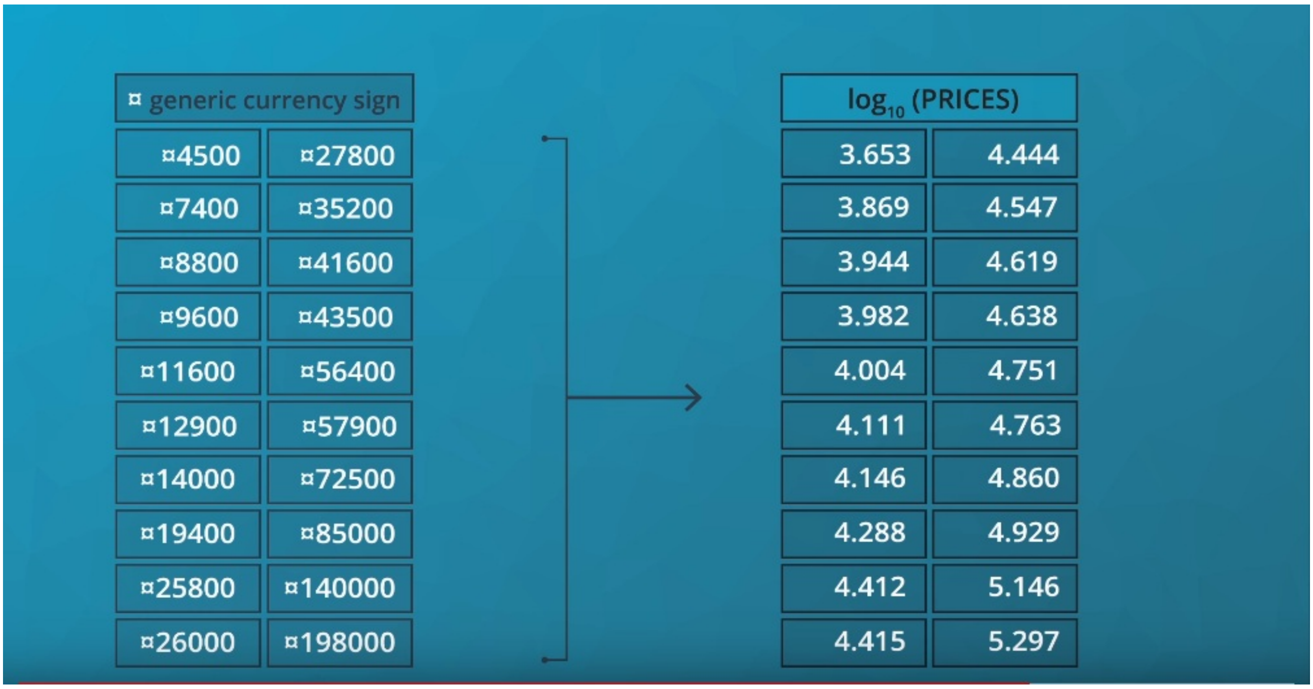
Scales and Transformations



¤ generic currency sign	
¤4500	¤27800
¤7400	¤35200
¤8800	¤41600
¤9600	¤43500
¤11600	¤56400
¤12900	¤57900
¤14000	¤72500
¤19400	¤85000
¤25800	¤140000
¤26000	¤198000

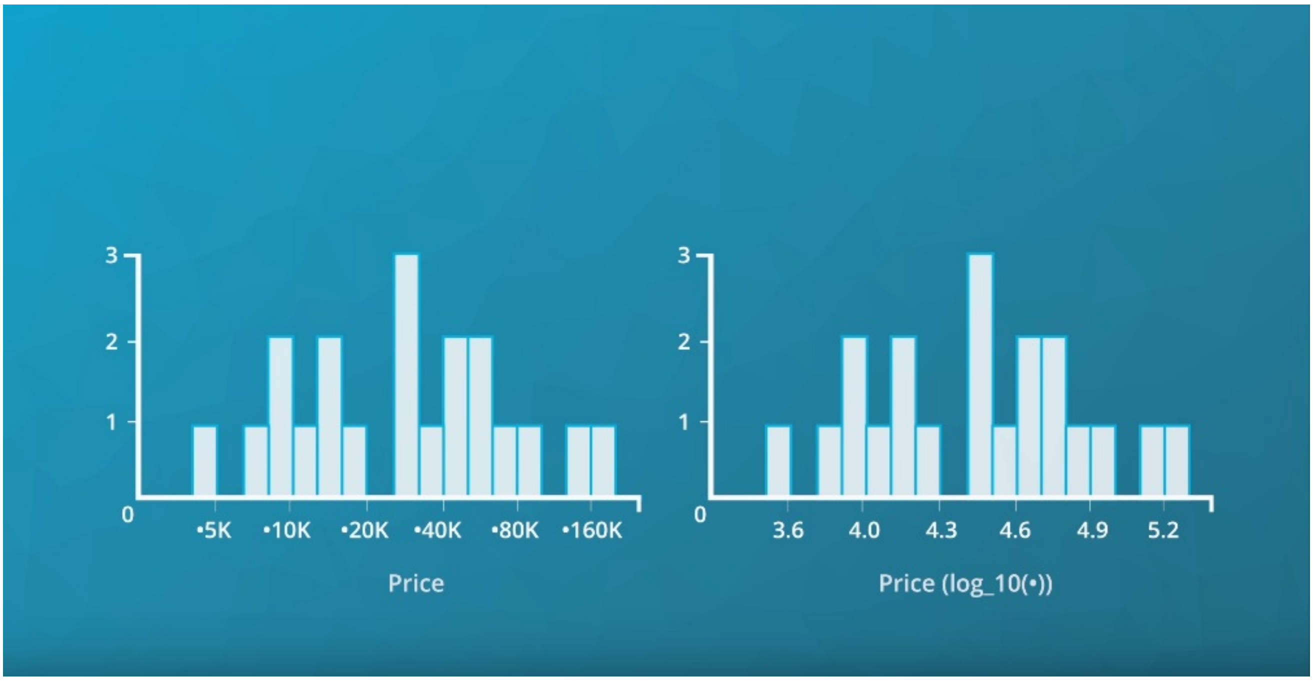
LINEAR SCALE

LOGARITHMIC SCALE



¤ generic currency sign	
¤4500	¤27800
¤7400	¤35200
¤8800	¤41600
¤9600	¤43500
¤11600	¤56400
¤12900	¤57900
¤14000	¤72500
¤19400	¤85000
¤25800	¤140000
¤26000	¤198000

$\log_{10}$ (PRICES)	
3.653	4.444
3.869	4.547
3.944	4.619
3.982	4.638
4.004	4.751
4.111	4.763
4.146	4.860
4.288	4.929
4.412	5.146
4.415	5.297

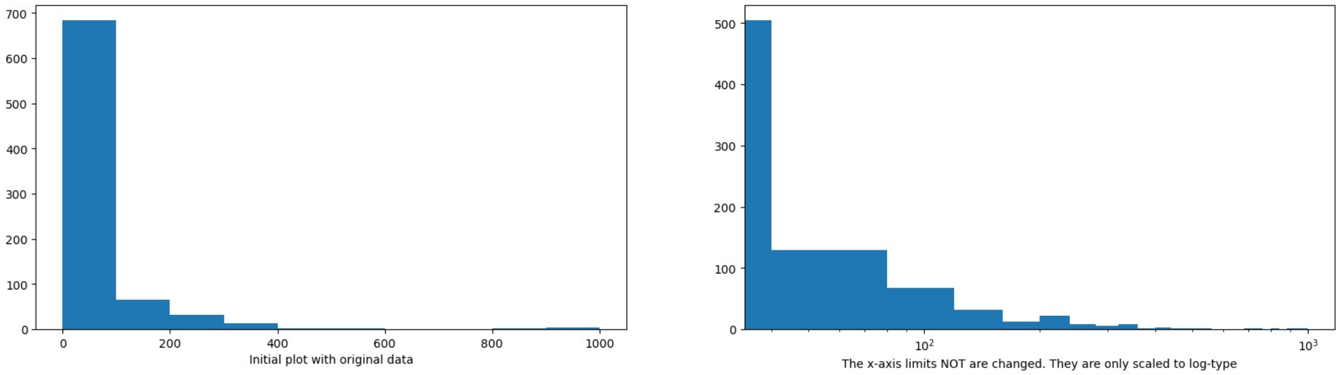


In [77]:

```
# Necessary import

pokemon = pd.read_csv('pokemon.csv')
pokemon.head(10)

plt.figure(figsize = [20, 5])

# HISTOGRAM ON LEFT: full data without scaling
plt.subplot(1, 2, 1)
plt.hist(data=pokemon, x='weight');
# Display a label on the x-axis
plt.xlabel('Initial plot with original data')

# HISTOGRAM ON RIGHT
plt.subplot(1, 2, 2)

# Get the ticks for bins between [0 - maximum weight]
bins = np.arange(0, pokemon['weight'].max()+40, 40)
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to apply.
# The possible values are: {"linear", "log", "symlog", "logit", ...}
# Refer - https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.xscale.html
plt.xscale('log')
plt.xlabel('The x-axis limits NOT are changed. They are only scaled to log-type')
```

Out[77]:

```
Text(0.5, 0, 'The x-axis limits NOT are changed. They are only scaled to log-type')
```



In [74]:

```
# Describe the data
pokemon['weight'].describe()
```

Out[74]:

```
count    807.000000
mean      61.771128
std      111.519355
min        0.100000
25%        9.000000
50%       27.000000
75%       63.000000
max      999.900000
Name: weight, dtype: float64
```

Notice two things about the right histogram of example 1 above, now.

1. Even though the data is on a log scale, the bins are still linearly spaced. This means that they change size from wide on the left to thin on the right, as the values increase multiplicative. Matplotlib's `xscale` function includes a few built-in transformations: we have used the '`log`' scale here.
2. Secondly, the default label (x-axis ticks) settings are still somewhat tricky to interpret and are sparse as well.

To address the bin size issue, we just need to change them so that they are evenly-spaced powers of 10. Depending on what you are plotting, a different base power like 2 might be useful instead.

To address the second issue of interpretation of x-axis ticks, the scale transformation is the solution. In a scale transformation, the gaps between values are based on the transformed scale, but you can interpret data in the variable's natural units.

Let's see another example below.

Example 2 - Scale the x-axis to log-type, and change the axis limit.

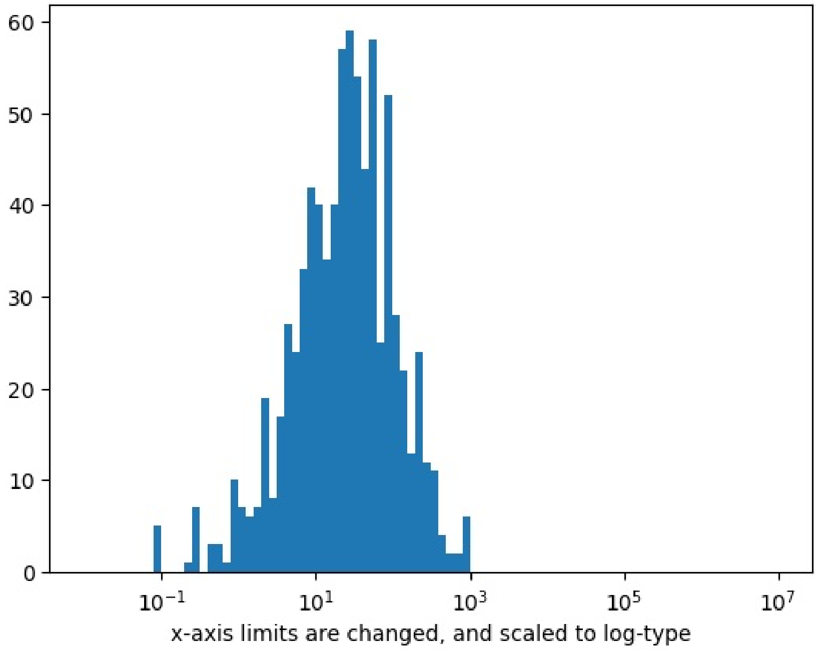
In [80]:

```
# Transform the describe() to a scale of log10
# Documentation: [numpy `log10`](https://docs.scipy.org/doc/numpy/reference/generated/numpy.log10.html)
np.log(pokemon['weight']).describe()
```

Out[80]:
```
count    807.000000
mean       3.141569
std        1.544962
min       -2.302585
25%        2.197225
50%        3.295837
75%        4.143135
max        6.907655
Name: weight, dtype: float64
```

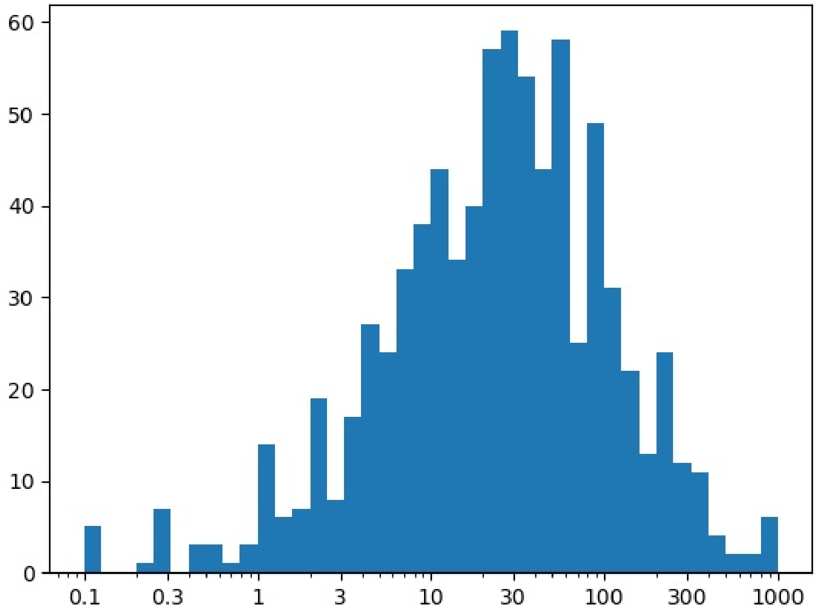
In [101…
```
# Axis transformation
# Bin size
bins = 10 ** np.arange(-2, 7+0.1, 0.1)
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to apply.
# The possible values are: {"linear", "log", "symlog", "logit", ...}
plt.xscale('log')


# Apply x-axis label
# Documentatin: [matplotlib `xlabel`](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xlabel.html))
plt.xlabel('x-axis limits are changed, and scaled to log-type')
```

Out[101]:
```
Text(0.5, 0, 'x-axis limits are changed, and scaled to log-type')
```



Example 3 - Scale the x-axis to log-type, change the axis limits, and increase the x-ticks

In [104…
```
# Get the ticks for bins between [0 - maximum weight]
bins = 10 ** np.arange(-1, 3+0.1, 0.1)

# Generate the x-ticks you want to apply
ticks = [0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000]
# Convert ticks into string values, to be displaye dlong the x-axis
labels = ['{}'.format(v) for v in ticks]

# Plot the histogram
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to apply.
# The possible values are: {"linear", "log", "symlog", "logit", ...}
plt.xscale('log')

# Apply x-ticks
plt.xticks(ticks, labels);
```



Observation - We've ended up with the same plot as when we performed the direct log transform, but now with a much nicer set of tick marks and labels.

For the ticks, we have used xticks() to specify locations and labels in their natural units. Remember: we aren't changing the values taken by the data, only how they're displayed. Between integer powers of 10, we don't have clean values for even markings, but we can still get close. Setting ticks in cycles of 1-3-10 or 1-2-5-10 are very useful for base-10 log transforms.

```
    It is important that the xticks are specified after xscale since that function has its own
built-in tick settings.
```

Alternative Approach

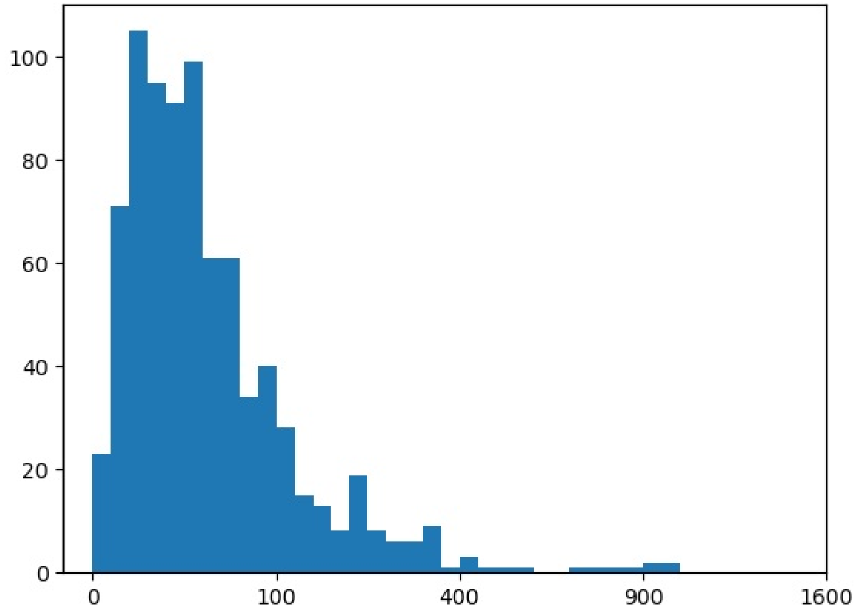
Be aware that a logarithmic transformation is not the only one possible. When we perform a logarithmic transformation, our data values have to all be positive; it's impossible to take a log of zero or a negative number. In addition, the transformation implies that additive steps on the log scale will result in multiplicative changes in the natural scale, an important implication when it comes to data modeling. The type of transformation that you choose may be informed by the context for the data.

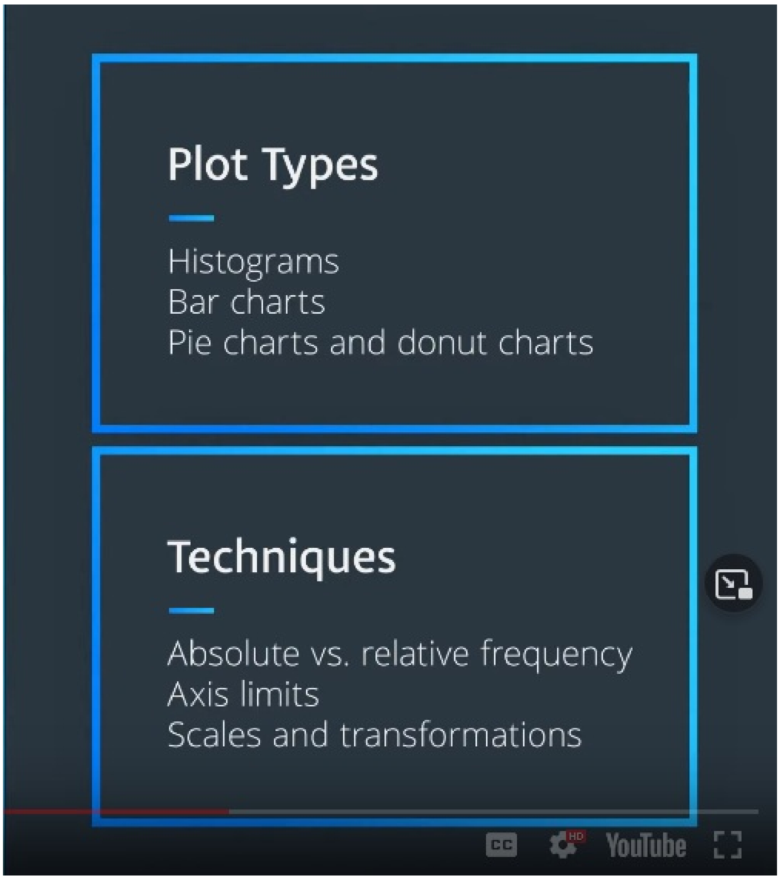
If you want to use a different transformation that's not available in xscale, then you'll have to perform some feature engineering. In cases like this, we want to be systematic by writing a function that applies both the transformation and its inverse. The inverse will be useful in cases where we specify values in their transformed units and need to get the natural units back. For the purposes of demonstration, let's say that we want to try plotting the above data on a square-root transformation. (Perhaps the numbers represent areas, and we think it makes sense to model the data on a rough estimation of radius, length, or some other 1-d dimension.) We can create a visualization on this transformed scale like this:

Example 4. Custom scaling the given data Series, instead of using the built-in log scale

In [105…

```
def sqrt_trans(x, inverse = False):
    """ transformation helper function """
    if not inverse:
        return np.sqrt(x)
    else:
        return x ** 2

# Bin resizing, to transform the x-axis
bin_edges = np.arange(0, sqrt_trans(pokemon['weight'].max())+1, 1)

# Plot the scaled data
plt.hist(pokemon['weight'].apply(sqrt_trans), bins = bin_edges)

# Identify the tick-locations
tick_locs = np.arange(0, sqrt_trans(pokemon['weight'].max())+10, 10)

# Apply x-ticks
plt.xticks(tick_locs, sqrt_trans(tick_locs, inverse = True).astype(int));
```





Plot Types

Histograms
Bar charts
Pie charts and donut charts

Techniques

Absolute vs. relative frequency
Axis limits
Scales and transformations

Glossary

1. Univariate visualizations: Visualize single-variables, such as bar charts, histograms, and line charts.
2. Bivariate visualizations: Plots representing the relationship between two variables measured on the given sample data. These plots help to identify the relationship pattern between the two variables.
3. Ordinal data: It is a categorical data type where the variables have natural and ordered categories. The distances between the categories are unknown, such as the survey options presented on a five-point scale.

Kernel Density Estimation

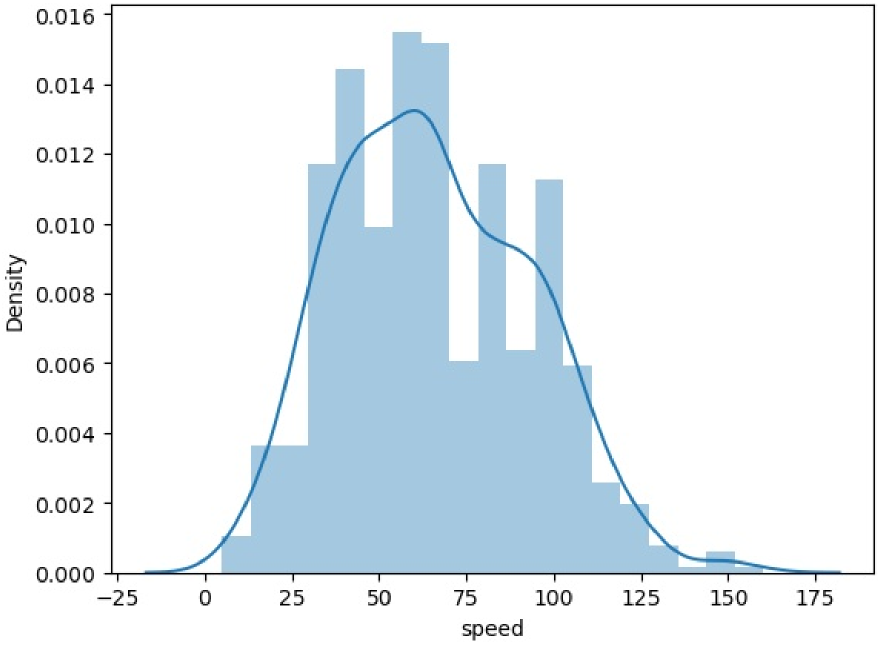
Earlier in this lesson, you saw an example of kernel density estimation (KDE) through the use of seaborn's distplot function, which plots a KDE on top of a histogram.

Example 1. Plot the Kernel Density Estimation (KDE)

In [106…

```
# The pokemon dataset is available to download at the bottom of this page.
sb.distplot(pokemon['speed']);
```

```
C:\Users\Isaac\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprec
ated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure
-level function with similar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



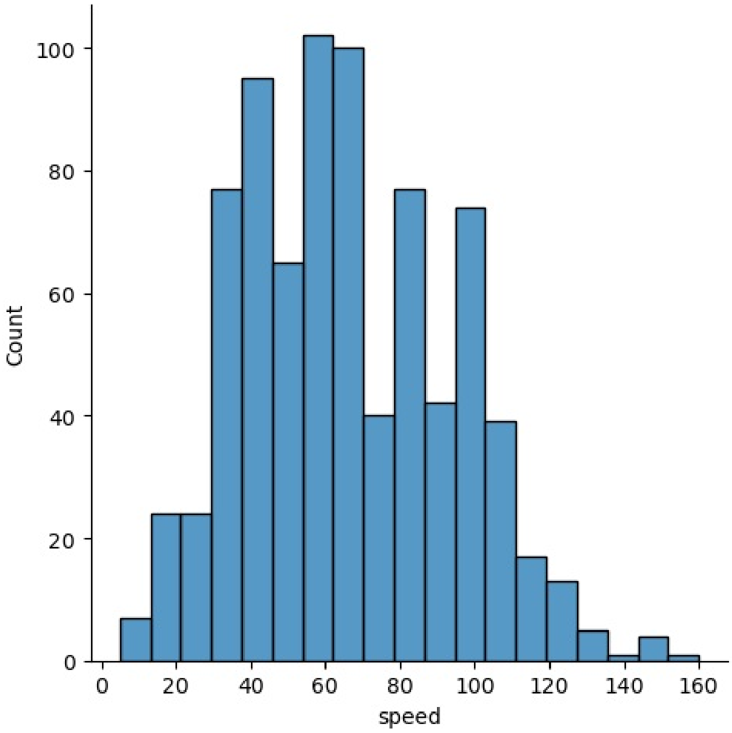
> Note - The distplot() function is deprecated in Seaborn v 0.11.0, and will be removed in a future version. The alternative is either of the following:

1. displot() - A figure-level function with similar flexibility.
2. histplot() - An axes-level function for histograms.

See the same example with newer `displot()` function:

In [107…

```
# Use this new function only with Seaborn 0.11.0 and above.
# The kind argument can take any one value from {“hist”, “kde”, “ecdf”}.
sb.displot(pokemon['speed'], kind='hist');
# Use the 'kde' kind for kernel density estimation
# sb.displot(pokemon['speed'], kind='kde');
```



Kernel density estimation is one way of estimating the probability density function of a variable. In a KDE plot, you can think of each observation as replaced by a small ‘lump’ of area. Stacking these lumps all together produces the final density curve. The default settings use a normal-distribution kernel, but most software that can produce KDE plots also include other kernel function options.

Seaborn's `distplot` function calls another function, `kdeplot`, to generate the KDE. The demonstration code below also uses a third function called by `distplot` for illustration, `rugplot()`. In a rugplot, data points are depicted as dashes on a number line. Example

2. Demonstrating distplot() and rugplot() to plot the KDE

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js