Summarize Data

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-01 at 19:02
```

After the data are prepared as a tidied data table and read into the analysis system such as Python, the first analysis step summarizes the data with descriptive statistics and visualizations. As always in data analysis, distinguish between *categorical variables* and *continuous variables* in the analysis.

If reading data from your computer's file system or a networked computer, identify the current working directory. This folder is the reference point in your file system for which file references begin. Then import the needed libraries and read the data for analysis.

As an option, list the current working directory, which, by default, is where this Jupyter notebook file is located. Your current working direction is the default location for reading and writing files. The working directory of `/content` is from running on Google Colab.

```
import os
os.getcwd()
```

```
'/content'
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Display matplotlib visualizations in the notebook. Not needed for Colab. If using, remove the `#` comment, the first character in the following code.

```
#%matplotlib inline
```

Can also display standard HTML links. For example, here view a list of all available color names to use when creating a visualization. [named colors](#) with a sample of each color

Can read the data from the Excel data file *employee.xlsx* on the web. Or, download to your computer or Google Drive and read from there as described the previous week, especially if going to be working on your computer without internet.

```
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
#d = pd.read_excel('data/employee.xlsx')
```

Always verify that the data were read as you intended.

```
d.shape
```

```
(37, 9)
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62

4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Categorical Variables

Counts

One Categorical Variable

Compute the counts of categorical data values with `pandas` function `value_counts()`, here categorical variable *Dept* within data frame *d*. Multiple ways to specify a variable as part of a data frame. First specify with the `.` notation.

```
d.Dept.value_counts()
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

Alternatively, use the `[]` notation.

```
d['Dept'].value_counts()
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

Yet a third notation to get the counts. Illustrated here to highlight that virtually any operation can be done multiple ways. If you are used to one way of coding an analysis, do not be surprised if someone else does it differently.

```
d.value_counts('Dept')
```

```
Dept
SALE    15
MKTG     6
ADMN     6
ACCT     5
FINC     4
dtype: int64
```

And a fourth way to get the counts, just to emphasize that there is typically no standard one way to do an analysis. Also, note that `value_counts()` is a `pandas` function, but when preceded by a data frame name and a dot in the function call, the leading `pd.` is not needed. A data frame is a `pandas` object, so the reference to `pandas` is understood. But when by itself, the `pd.` must precede the function call.

```
pd.value_counts(d.Dept)
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

To obtain the relative frequencies or proportions instead of the frequencies or counts, set the `normalize` paramaeter to `True`.

```
d.Dept.value_counts(normalize=True)
```

```
SALE    0.416667
ADMN    0.166667
MKTG    0.166667
ACCT    0.138889
FINC    0.111111
Name: Dept, dtype: float64
```

The categorical variable *JobSat* has categories 'low', 'med', and 'high'. Python does not understand the English language, and so has no way of knowing how these categories should be ordered. Python does not know that 'low' is less than 'high', for example. Instead, use the

`Categorical()` function to explicitly define a variable as categorical with the option to specify the ordering of the categories when accessed in subsequent analyses.

In this example, *JobSat* is converted from type `object` as read to type `category`.

```
d['JobSat'] = pd.Categorical(d['JobSat'], categories=['low', 'med', 'high'], ordered=True)
d.dtypes
```

```
Name        object
Years      float64
Gender      object
Dept        object
Salary     float64
JobSat    category
Plan         int64
Pre          int64
Post         int64
dtype: object
```

Two Categorical Variables

Use the `pandas` function `crosstab()` for cross-tabulation, to compute two-way frequency distributions for categorical variables. Can refer to the variables with the `.` notation.

```
pd.crosstab(d.Dept,d.Gender)
```

Gender	F	M
Dept		
ACCT	3	2
ADMN	4	2
FINC	1	3
MKTG	5	1
SALE	5	10

Alternatively, use the [ ] notation in the call to `crosstab()`.

```
ct = pd.crosstab(d['Dept'], d['Gender'])
ct
```

Gender	F	M
Dept		
ACCT	3	2
ADMN	4	2
FINC	1	3
MKTG	5	1
SALE	5	10

Bar Charts

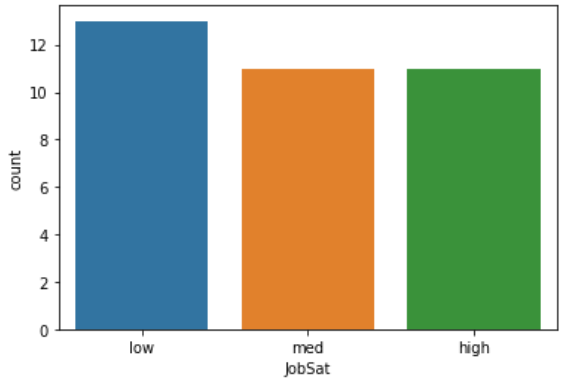
The primary data visualization package for Python data analysis is `matplotlib`. A more recent development is the `seaborn` package which builds upon `matplotlib`, designed to provide more elegant graphics with less work. Most of the examples presented here are from `seaborn`, with the abbreviation `sns`. They could also be replicated with `matplotlib`, continuing the theme that almost always, there is more than one way to proceed.

`seaborn` offers the function `countplot()` to count the number of occurrences for each category of a categorical variable. The function requires to include the parameter name with the value of each parameter, not relying upon the position of the value in the parameter list. For example, a positional system would not require the parameter name for the first parameter listed in the function definition.

One Categorical Variable

```
sns.countplot(x='JobSat', data=d)
```

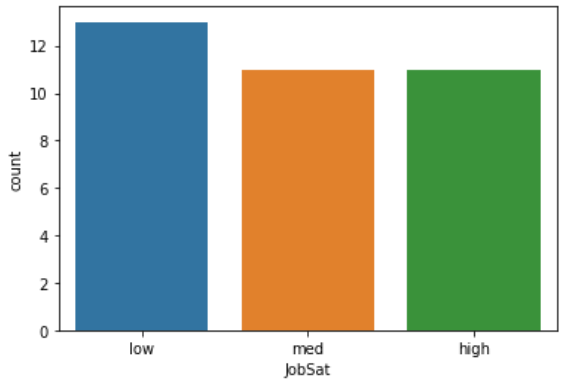
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff22a56e390>
```



If the categorical variable was not defined as an ordered categorical variable with the function `Categorical()`, you can specify the ordering of the categories in the call to `countplot()`.

```
sns.countplot(x=d['JobSat'], order=['low', 'med', 'high'])
```

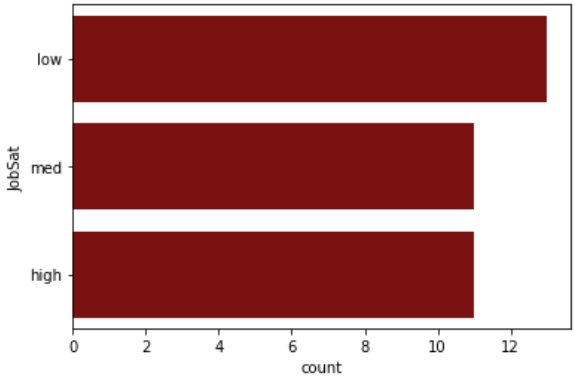
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff229479090>
```



With parameter `y`, request that the categories be placed on the y-axis, that is, a horizontal bar chart. With parameter `color`, specify a custom color.

```
sns.countplot(y='JobSat', data=d, color='darkred')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228ff8410>
```

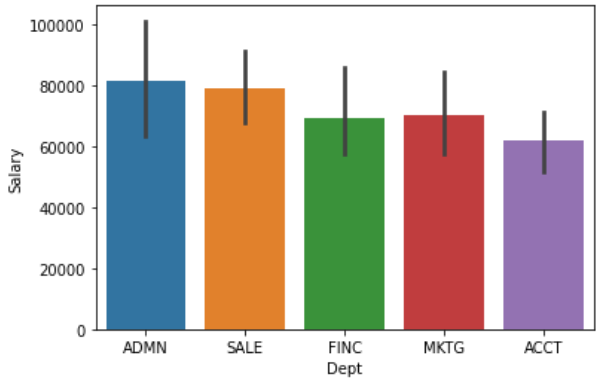


The `seaborn` package has multiple functions that yield bar charts. The previously illustrated `countplot()` function plots the height of the bars according to the count (frequency) or proportion (relative frequency) of each category. The function named `barplot()` applies to one categorical variable and then a numerical variable that defines the height of the bars.

By default, the mean of the numerical variable for each category is plotted, along with a corresponding error bar that illustrates the 95% confidence interval.

```
sns.barplot(x='Dept', y='Salary', data=d)
```

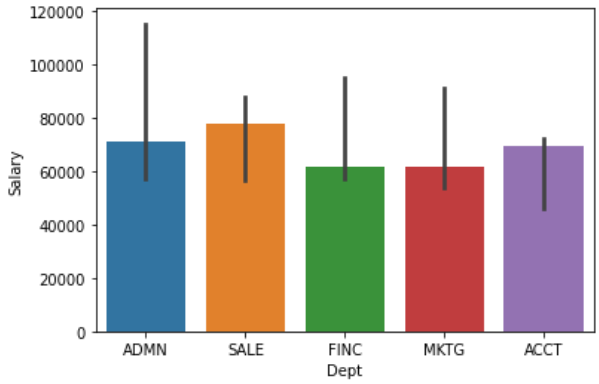
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228f122d0>
```



The default statistic plotted for the height of the bars is the mean of the numerical variable. Use the parameter `estimator` to specify another statistic. The `numpy` package defines these statistics, so precede the statistic's name with `np.` when calling the `barplot()` function.

```
sns.barplot(x='Dept', y='Salary', data=d, estimator=np.median)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff22a05b8d0>
```

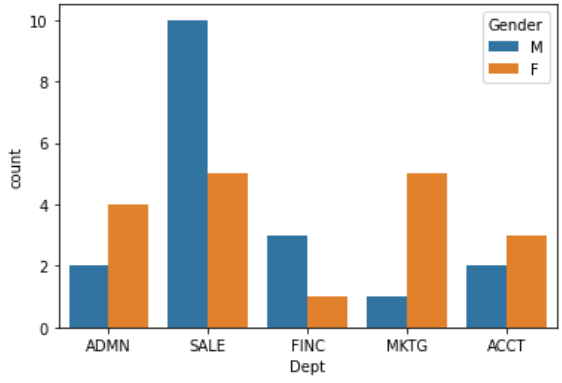


Two Categorical Variables

The bar chart of two categorical variables is presented in one of two basic forms, stacked or unstacked. This first example is unstacked, with the bars separated for the levels of the second variable, *Gender*, at each level of the first variable, *Dept*.

```
sns.countplot(x='Dept', hue='Gender', data=d)
```

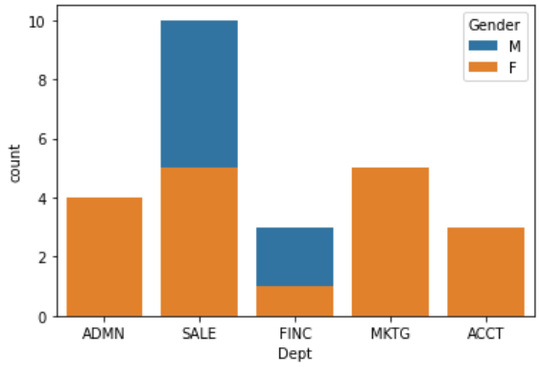
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228e0ff50>
```



The `seaborn` version of the stacked form of the bar chart, indicated by the `dodge` parameter, does not appear to work. The bars for categories SALE and FINC work, but the remaining bars do not show the Male counts.

```
sns.countplot(x='Dept', hue='Gender', dodge=False, data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228e8a790>
```

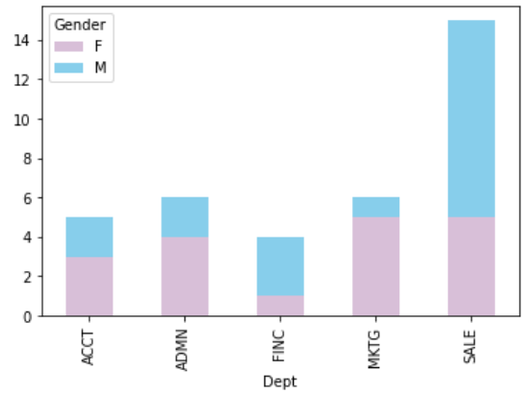


Go to the foundational `matplotlib` with its `plot()` function to get the stacked bar chart to work. Indicate a bar chart with the `kind` parameter and set `stacked` to `True`. For `matplotlib` we need to feed the output of `crosstabs()`, which we previously called *ct*, into the `plot()` function to visualize the cross-tabulation table as a bar chart of two categorical variables. Add some optional colors with the `color` parameter. Find a full list of named colors.

One way to use `plot()` is to enter the cross-tabulation matrix into the analysis. Do so by beginning the function call with the name of the matrix, here *ct*.

```
ct.plot(kind='bar', color=['thistle','skyblue'], stacked=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228cd3490>
```

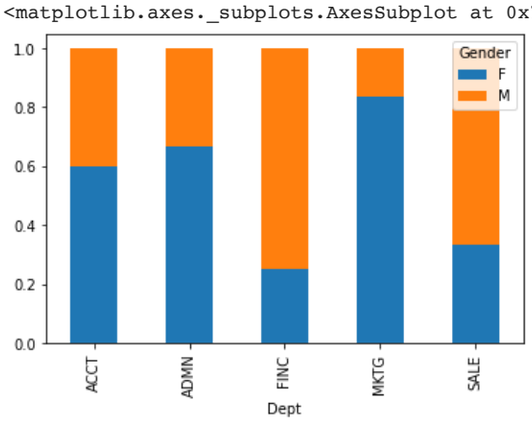


The 100% bar chart for these two categorical variables shows the *percentage* of *Gender* distributed across *each* department. This visualization compares the second categorical variable, here Gender, across the levels of the first variable, particularly useful when there are unequal group sizes.

Specify the grouping of the categorical variables with the `pandas` function `groupby()`. The key to getting the 100% stacked bar chart is to set the `normalize` parameter to `True`, which, as shown previously, converts the counts to frequencies. With `matplotlib`, plot the 100% stacked bar chart, where each bar goes the complete 100%. This example uses the chain method explained in the previous week, where each function call appears on a separate line.

```
(d
 .groupby('Dept')['Gender']
 .value_counts(normalize=True)
 .unstack('Gender')
 .plot.bar(stacked=True)
)
```

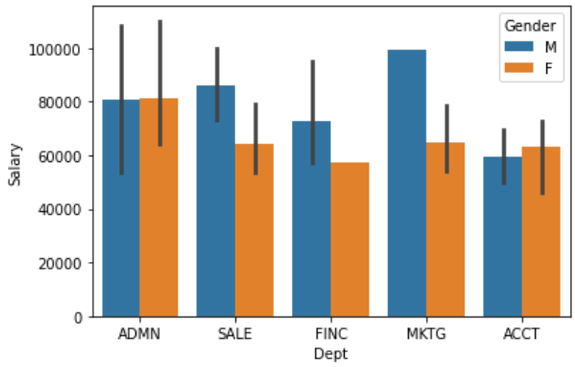
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228c6d6d0>
```



Here apply `barplot()` to two categorical variables, specified with parameters of `x` and `hue`. Specify the numeric variable for analysis with parameter `y`.

```
sns.barplot(x='Dept', y='Salary', hue='Gender', data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228bfc190>
```



Continuous Variables

Statistics

Methods are available for individual statistics.

Calculate the mean of a variable with the `pandas` function `mean()`.

```
d["Salary"].mean()
```

```
73795.55675675675
```

Usually better to also invoke the `round()` function to not have so many decimal digits. In this example, save the mean in its own variable, *m*, then round to two decimal digits.

```
m = d["Salary"].mean()
round(m, 2)
```

```
73795.56
```

The `pandas` function `describe()` computes summary statistics of continuous variables over the entire data frame. Unmodified it applies to all the continuous variables. The basic summary statistics that describe the sample values of a continuous variable: number of non-missing values (count), mean (mean), standard deviation (std), minimum (min), first quartile (25%), median (50%), third quartile (75%), and maximum (max).

```
d.describe()
```

	Years	Salary	Plan	Pre	Post
count	36.000000	37.000000	37.000000	37.000000	37.000000
mean	9.388889	73795.556757	1.783784	78.783784	81.000000
std	5.723524	21799.533464	0.712396	12.037292	11.592622
min	1.000000	46124.970000	1.000000	59.000000	59.000000
25%	5.000000	56772.950000	1.000000	70.000000	72.000000
50%	9.000000	69547.600000	2.000000	80.000000	84.000000
75%	13.000000	87785.510000	2.000000	90.000000	91.000000
max	24.000000	134419.230000	3.000000	100.000000	100.000000

Or, apply to one or more selected variables, here the variable Salary.

```
d.Salary.describe()
```

```
count        37.000000
mean      73795.556757
std       21799.533464
min       46124.970000
25%       56772.950000
50%       69547.600000
75%       87785.510000
max      134419.230000
```

```
Name: Salary, dtype: float64
```

Pivot Tables

Summarizing numerical data over different sub-groups of a data set is a standard analysis technique. Calculate descriptive statistics across groups of data.

> *Aggregation*: Summarize a numerical variable with descriptive statistics computed over combinations of groups defined by one or more categorical variables.

The analysis aggregates statistics across the numerical variables grouped by levels of one or more categorical variables. Excel refers to a table of aggregated statistics as a *pivot table*.

One possibility for aggregation is the `pandas` function `groupby()`, applied to data frames. Here compute the mean of the different groups for all numerical variables in the data frame across the levels of *Dept*.

Note: Available functions include `count()`, `sum()`, `mean()`, `median()`, `min()`, `max()`, `mode()`, `std()`, and `var()`.

Note: This is easier than all that clicking and mousing to get the same result with the Excel pivot table function. (Can also easily write the results back to Excel with function `to_excel()`.)

```
d.groupby('Dept').mean()
```

Dept	Years	Salary	Plan	Pre	Post
ACCT	5.600000	61792.776000	2.00	76.600000	78.600000
ADMN	10.166667	81277.116667	2.00	80.833333	82.000000
FINC	10.250000	69010.675000	1.75	79.750000	82.250000
MKTG	9.833333	70257.128333	2.00	79.666667	84.000000
SALE	10.285714	78830.064667	1.60	79.000000	81.133333

To define a two-way grouping, provide a vector of variable names to `groupby()`.

```
d.groupby(['Dept', 'Gender']).mean()
```

Dept	Gender	Years	Salary	Plan	Pre	Post
ACCT	**F**	4.666667	63237.163333	2.000000	73.333333	77.333333
	M	7.000000	59626.195000	2.000000	81.500000	80.500000
ADMN	**F**	7.500000	81434.002500	2.250000	80.000000	79.750000
	M	15.500000	80963.345000	1.500000	82.500000	86.500000
FINC	**F**	7.000000	57139.900000	2.000000	90.000000	86.000000
	M	11.333333	72967.600000	1.666667	76.333333	81.000000
MKTG	**F**	8.200000	64496.022000	1.800000	79.400000	84.000000
	M	18.000000	99062.660000	3.000000	81.000000	84.000000
SALE	**F**	6.600000	64188.254000	1.600000	76.200000	78.600000
	M	12.333333	86150.970000	1.600000	80.400000	82.400000

Use the `agg()` function to aggregate the values of multiple numerical variables. With this function, need to precede the function names with `np.` as they are functions defined in the `numpy` package.

Here write the code for the aggregation in chain function notation.

```
(d
 .groupby(['Dept', 'Gender'])
 .agg([np.mean, np.median])
)
```

		Years		Salary		Plan		Pre		Post	
		mean	median	mean	median	mean	median	mean	median	mean	median
Dept	**Gender**										
ACCT	**F**	4.666667	3.0	63237.163333	71084.020	2.000000	2.0	73.333333	74.0	77.333333	84.0

	M	7.000000	7.0	59626.195000	59626.195	2.000000	2.0	81.500000	81.5	80.500000	80.5
ADMN	F	7.500000	5.0	81434.002500	71058.595	2.250000	2.0	80.000000	80.5	79.750000	80.0
	M	15.500000	15.5	80963.345000	80963.345	1.500000	1.5	82.500000	82.5	86.500000	86.5
FINC	F	7.000000	7.0	57139.900000	57139.900	2.000000	2.0	90.000000	90.0	86.000000	86.0
	M	11.333333	10.0	72967.600000	66312.890	1.666667	1.0	76.333333	80.0	81.000000	83.0
MKTG	F	8.200000	8.0	64496.022000	61356.690	1.800000	2.0	79.400000	80.0	84.000000	90.0
	M	18.000000	18.0	99062.660000	99062.660	3.000000	3.0	81.000000	81.0	84.000000	84.0
SALE	F	6.600000	8.0	64188.254000	56508.320	1.600000	2.0	76.200000	74.0	78.600000	72.0
	M	12.333333	13.0	86150.970000	82442.740	1.600000	1.0	80.400000	75.0	82.400000	79.0

Of course, as seen before, there are multiple ways to proceed. Another possibility uses the `pandas` function `pivot_table()` for aggregating (pivoting) the values of one or more continuous variables over different groups defined by one or more categorical variables.

Specify the continuous variables over which to aggregate with the `values` parameter. The `index` parameter specifies the categorical variables that define the groups. The `aggfunc` parameter specifies the statistic for the aggregation, here the mean. Note that the calculation of the mean here is from the `numpy` package, abbreviated `np` upon which `pandas` depends.

```
pd.pivot_table(d, values=['Years', 'Salary', 'Plan', 'Pre', 'Post'],
               index=['Dept', 'Gender'], aggfunc=np.mean)
```

		Plan	Post	Pre	Salary	Years
Dept	**Gender**					
ACCT	F	2.000000	77.333333	73.333333	63237.163333	4.666667
	M	2.000000	80.500000	81.500000	59626.195000	7.000000
ADMN	F	2.250000	79.750000	80.000000	81434.002500	7.500000
	M	1.500000	86.500000	82.500000	80963.345000	15.500000
FINC	F	2.000000	86.000000	90.000000	57139.900000	7.000000
	M	1.666667	81.000000	76.333333	72967.600000	11.333333
MKTG	F	1.800000	84.000000	79.400000	64496.022000	8.200000
	M	3.000000	84.000000	81.000000	99062.660000	18.000000
SALE	F	1.600000	78.600000	76.200000	64188.254000	6.600000
	M	1.600000	82.400000	80.400000	86150.970000	12.333333

If you do not specify the `values` parameter, then all numeric variables are analyzed.

```
pd.pivot_table(d, index=['Dept', 'Gender'], aggfunc=np.mean)
```

		Plan	Post	Pre	Salary	Years
Dept	**Gender**					
ACCT	F	2.000000	77.333333	73.333333	63237.163333	4.666667
	M	2.000000	80.500000	81.500000	59626.195000	7.000000
ADMN	F	2.250000	79.750000	80.000000	81434.002500	7.500000
	M	1.500000	86.500000	82.500000	80963.345000	15.500000
FINC	F	2.000000	86.000000	90.000000	57139.900000	7.000000
	M	1.666667	81.000000	76.333333	72967.600000	11.333333
MKTG	F	1.800000	84.000000	79.400000	64496.022000	8.200000
	M	3.000000	84.000000	81.000000	99062.660000	18.000000
SALE	F	1.600000	78.600000	76.200000	64188.254000	6.600000
	M	1.600000	82.400000	80.400000	86150.970000	12.333333

Visualizations

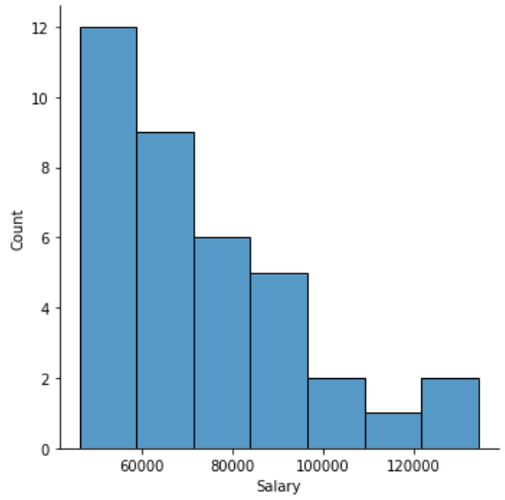
The distribution of a continuous variable can be presented several different ways. Perhaps the two most encountered visualizations are histograms and box plots.

Histogram

`seaborn` provides two functions for generating histograms, `histplot()` and `displot()`. Here we focus on the later.

```
sns.displot(d, x='Salary')
```

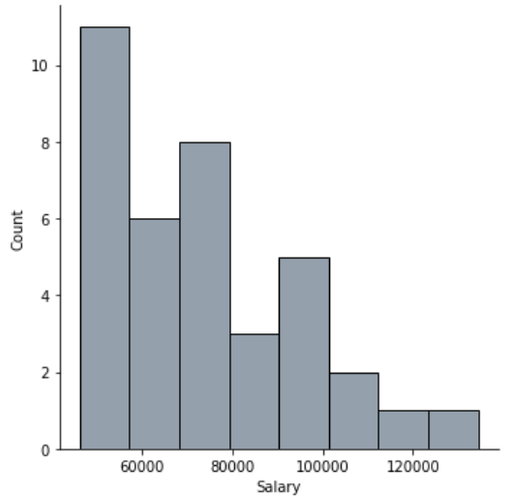
```
<seaborn.axisgrid.FacetGrid at 0x7ff228bf4d10>
```



Reference the parameter *color* for a custom color. Specify the number of bins with the `bins` parameter. Could also use the `binwidth` parameter.

```
sns.displot(d, x='Salary', color='slategray', bins=8)
```

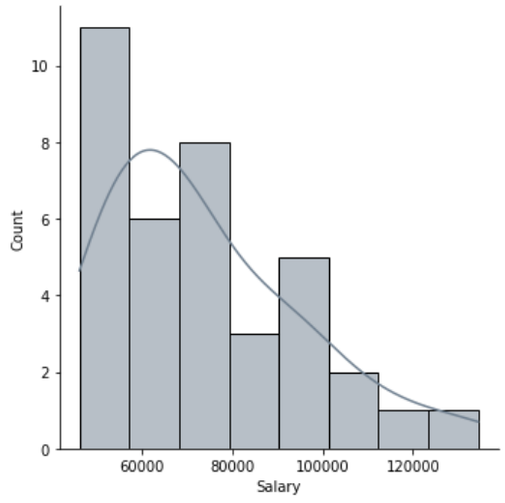
```
<seaborn.axisgrid.FacetGrid at 0x7ff228c03e10>
```



The `seaborn` package provides more advanced plots. For example, easily overlay a smoothed frequency-type curve, called a *density plot*, on the histogram with the function `displot()`.

```
sns.displot(d, x='Salary', color='slategray', bins=8, kde=True)
```

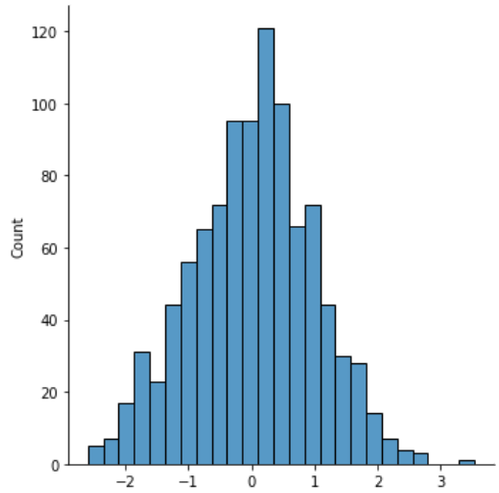
```
<seaborn.axisgrid.FacetGrid at 0x7ff228a9f250>
```




```
from numpy.random import normal
```

```
sim_data = normal(size=1000)
sns.displot(x=sim_data)
```

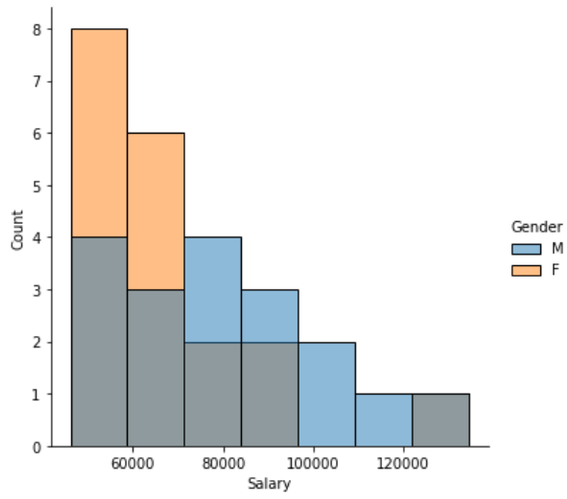
```
<seaborn.axisgrid.FacetGrid at 0x7ff21e16ac90>
```



Throughout the various `seaborn` functions, generate multiple visualizations on the same panel according to different groups with the `hue` parameter.

```
sns.displot(d, x='Salary', hue='Gender')
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21e15f210>
```

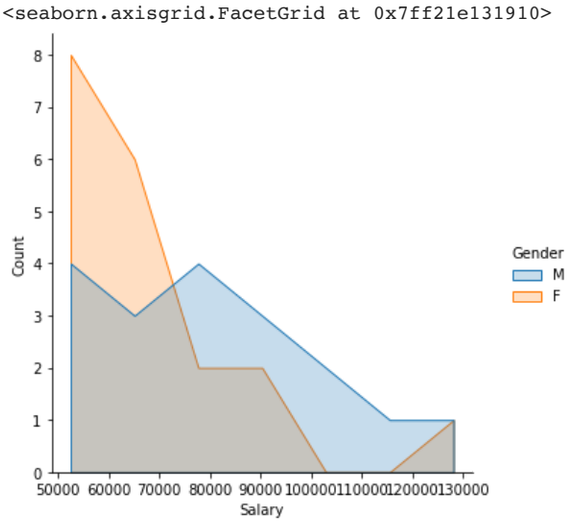


A closely related visualization to a histogram is a frequency polygon, which may provide a helpful portrayal of the distributions for overlapping distributions. For example, in the above histogram, overlapping values are shown in gray. In the frequency polygon version, the polygons overlap as a blending of their respective colors.

Specify with the `element` parameter.

```
sns.displot(d, x='Salary', hue='Gender', element='poly')
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21e131910>
```

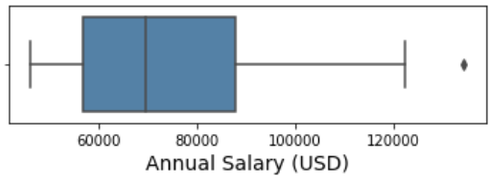


Box Plot

The box plot has already been demonstrated in an earlier notebook, but included here for completeness as a standard visualization of a distribution, emphasizing detecting outliers. Use the `seaborn` function `boxplot()`. The height of the box plot is not relevant, and by default is quite large, so set to a small number, 1.5, with the `matplotlib figure()` function.

```
plt.figure(figsize=(6,1.5))
sns.boxplot(x=d['Salary'], color='steelblue')
plt.xlabel('Annual Salary (USD)', fontsize=14)
```

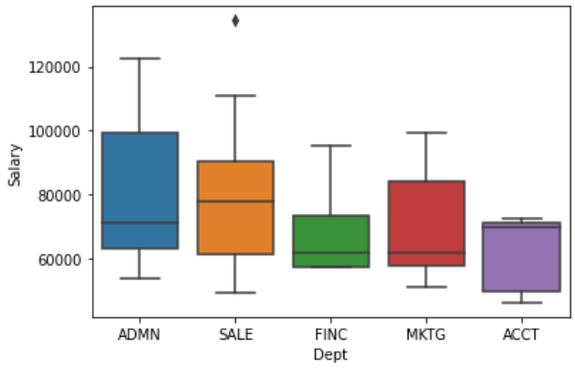
```
Text(0.5, 0, 'Annual Salary (USD)')
```



The box plot is an excellent way to visualize the distribution of a continuous variable across levels of a categorical variable. Here compare *Salary* across the different *Dept* or departments of a company.

```
sns.boxplot(x='Dept', y='Salary', data=d)
```

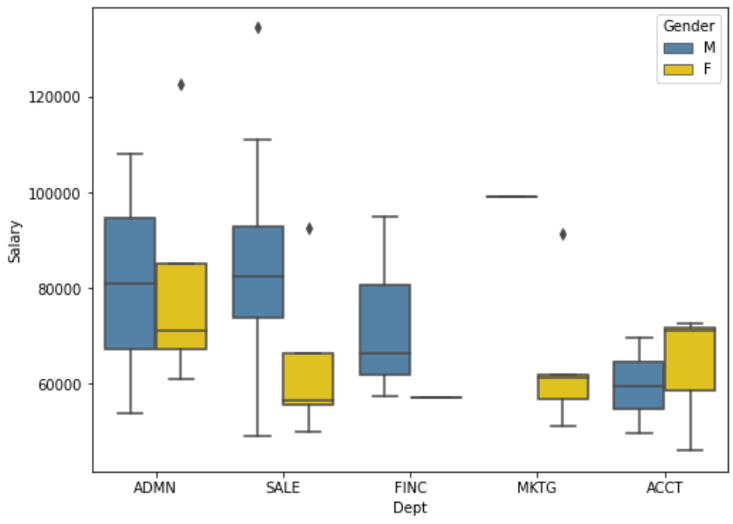
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228c73e50>
```



Can go one step further and specify a second categorical variable at each level of the first categorical variable. Here, with the `hue` parameter, show box plots for *Gender* at each *Dept*. Change the colors from the default blue and orange with the `palette` parameter.

```
plt.figure(figsize=(8,6))
sns.boxplot(x='Dept', y='Salary', hue='Gender', palette=['steelblue', 'gold'], data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff21deb3850>
```



Correlational Structure

The *correlational structure* refers to the (linear) relations among the variables, pairwise, that is, two at a time. The sample or population correlation varies between -1 and 1, which respectively indicate perfect − or + linear relationship. A correlation of 0 indicates no linear relationship.

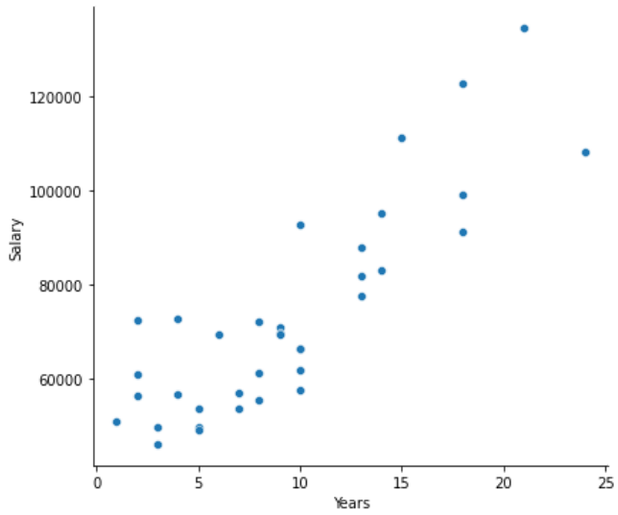
Scatterplot

The classic visualization of the relation between two numerical variables is the *scatterplot*, which plots each pair of values for the two variables for a row of data as a point. The coordinates of the plotted point are the values of the two variables for that row of data.

A `seaborn` scatterplot function is `relplot()`. The `aspect` parameter controls the ratio of height and width.

```
sns.relplot(x='Years', y='Salary', data=d, aspect=1.2)
```

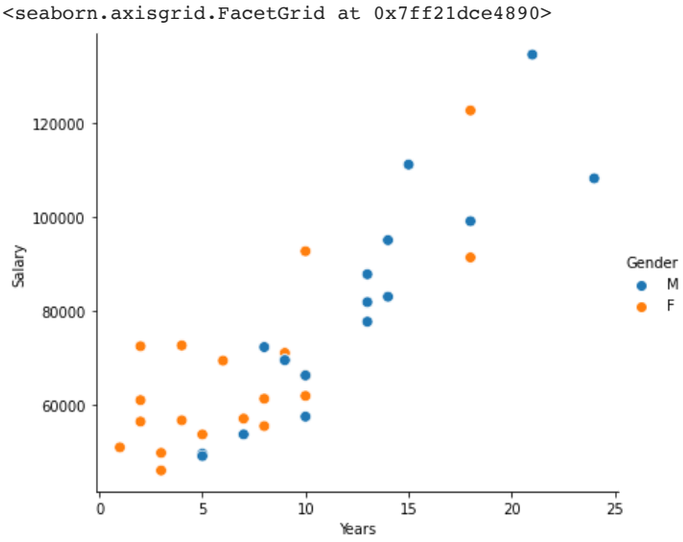
```
<seaborn.axisgrid.FacetGrid at 0x7ff22022e8d0>
```



Consistent with other `seaborn` functions, introduce a grouping variable with the `hue` parameter. Control the size of the points with the `s` parameter.

```
sns.relplot(x='Years', y='Salary', hue='Gender', s=60, data=d, aspect=1.2)
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21dce4890>
```



Correlation matrix

The correlation coefficient varies from -1 for a perfectly negative or inverse relationship, to 0 for no relationship, to +1 for a perfectly positive relationship.

Get correlations with the `pandas` function `corr()`. Here calculate the correlation between number of *Years* worked for the company and annual *Salary*. Clearly an employee's salary is related to the length of time working for the company.

```
d['Years'].corr(d['Salary']).round(2)
```

```
0.85
```

The correlation matrix in traditional form, a square matrix that, given a list of variables, contains the correlation for each variable in the list, twice, because the correlation coefficient is the same regardless of the order of the two variables: $r_{12} = r_{21}$. 1's run down what is called the *principle diagonal*, indicating that each variable correlates with itself a perfect 1.0. It is distracting and generally useless to display more than two decimal digits for each of the displayed correlations.

```
d.corr().round(2)
```

Heat map

The heat map visualizes the correlation matrix, color coded according to the intensity of each correlation. Get the heat map with the seaborn function `heatmap()`. Specify the parameter `annot` for annotation to be `True` to also provide the numerical value of the correlations. Here use the `loc` method to select the variables for the heat map.

```
keep_vars = ['Years', 'Salary', 'Pre', 'Post']
d2 = d.loc[:, keep_vars]
sns.heatmap(d2.corr().round(2), linewidths=2.0, annot=True)
```

Scatterplot matrix

The *scatterplot matrix* has the form of a correlation matrix, but replaces each correlation with the corresponding scatterplot. The diagonal elements of the correlation matrix are replaced with a plot of the distribution of the variable.

Get the scatterplot matrix with the seaborn function `pairplot()`.

Can also use the `kind` parameter to specify a regression line for each scatterplot, and the `diag_kind` parameter to specify kernel density plots (kde) in the diagonal. If there is no `hue` parameter, the default is a histogram for each diagonal element.

```
sns.pairplot(data=d, vars=["Years", "Salary"], kind="reg",
             diag_kind="kde")
```

The pairplot can be obtained for multiple levels of a categorical variable. The parameter `hue` specifies a categorical variable from which to map its levels to different colors in scatterplot matrix.

```
sns.pairplot(d, vars=["Years", "Salary", "Pre", "Post"], hue='Gender')
```

✓ 0s completed at 4:08 PM