Analyzing Categorical Data

In this project, we'll be working with categorical data and will be using a subset of data from the following data set: (https://www.kaggle.com/datasets/norc/general-social-survey?select=gss.csv (https://www.kaggle.com/datasets/norc/general-social-survey?select=gss.csv)).

After cleaning the data, we will use some visualizations tools. We also had used statsmodels for a special type of categorical plot.

In [3]:
```
# Import packages
import pandas as pd
import numpy as np
import plotly.express as px
import matplotlib.pyplot as plt
from statsmodels.graphics.mosaicplot import mosaic

# Read in csv as a DataFrame and preview it
df = pd.read_csv("/Users/antoniogondim/Downloads/gss_sub.csv")
df
```

Out[3]:

	year	id	labor_status	self_employed	marital_status	n_siblings	age	high_school	
0	1972.0	1.0	WORKING FULLTIME	SOMEONE ELSE	NEVER MARRIED	3.0	23.0	16.0	BA(
1	1972.0	2.0	RETIRED	SOMEONE ELSE	MARRIED	4.0	70.0	10.0	S
2	1972.0	3.0	WORKING PARTTIME	SOMEONE ELSE	MARRIED	5.0	48.0	12.0	S
3	1972.0	4.0	WORKING FULLTIME	SOMEONE ELSE	MARRIED	5.0	27.0	17.0	BA(
4	1972.0	5.0	KEEPING HOUSE	SOMEONE ELSE	MARRIED	2.0	61.0	12.0	S
...	...	...	...	...	...	...	...	...	
59594	2014.0	2539.0	KEEPING HOUSE	SOMEONE ELSE	WIDOWED	6.0	89.0	14.0	C
59595	2014.0	2540.0	WORKING FULLTIME	SOMEONE ELSE	DIVORCED	3.0	56.0	12.0	S
59596	2014.0	2541.0	WORKING FULLTIME	SOMEONE ELSE	NEVER MARRIED	5.0	24.0	14.0	S
59597	2014.0	2542.0	WORKING FULLTIME	SOMEONE ELSE	NEVER MARRIED	2.0	27.0	13.0	S
59598	2014.0	2543.0	WORKING PARTTIME	SOMEONE ELSE	WIDOWED	2.0	71.0	12.0	S

59599 rows × 16 columns

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59599 entries, 0 to 59598
Data columns (total 16 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   year                   59599 non-null  float64
 1   id                     59599 non-null  float64
 2   labor_status           59583 non-null  object 
 3   self_employed          59306 non-null  object 
 4   marital_status         59575 non-null  object 
 5   n_siblings             56682 non-null  float64
 6   age                    59599 non-null  float64
 7   high_school            59440 non-null  float64
 8   degree                 59464 non-null  object 
 9   political_affiliation  59257 non-null  object 
 10  environment            59388 non-null  object 
 11  law_enforcement        59378 non-null  object 
 12  drugs                  59380 non-null  object 
 13  space_exploration      59596 non-null  object 
 14  inequality             1532 non-null   float64
 15  household_size         59599 non-null  float64
dtypes: float64(7), object(9)
memory usage: 7.3+ MB
```

In [5]:
```
df=df.drop('inequality', axis=1)
#Too many null values at this column
df.isnull().sum()
```

Out[5]:
```
year                        0
id                          0
labor_status               16
self_employed             293
marital_status             24
n_siblings               2917
age                         0
high_school               159
degree                    135
political_affiliation     342
environment               211
law_enforcement           221
drugs                     219
space_exploration           3
household_size              0
dtype: int64
```

In [6]:
```
df['n_siblings'].value_counts()
```

Out[6]:
```
 2.0     10717
 1.0      9602
 3.0      9109
 4.0      6705
 5.0      4860
 6.0      3688
 7.0      3168
 8.0      2075
 9.0      1536
-1.0      1518
 10.0     1066
 11.0      826
 12.0      550
 13.0      370
 14.0      213
 15.0      144
 99.0      111
 16.0      105
 98.0       57
 17.0       48
 21.0       46
 18.0       41
 20.0       29
 19.0       25
 22.0       15
 23.0       14
 25.0        6
 27.0        6
 26.0        6
 31.0        6
 24.0        5
 30.0        4
 33.0        2
 32.0        2
 29.0        2
 68.0        1
 35.0        1
 34.0        1
 37.0        1
 55.0        1
Name: n_siblings, dtype: int64
```

In [7]:
```
df.n_siblings.replace(np.nan,int(2.0), inplace=True)
```

In [8]:
```
df.shape
```

Out[8]:
```
(59599, 15)
```

Type *Markdown* and LaTeX: α^2

In [9]:
```
df=df.dropna()
df.shape
```

Out[9]:
```
(58439, 15)
```

Above we see that our DataFrame contains `float64` column (numerical data), as well as a number of `object` columns, i.e object data types contain strings.

df.describe() method with the `include` parameter to select a particular DataType (in this case `"O"`). This returns the count, number of unique values, the mode, and frequency of the mode for each column having object as data type.

In [10]:
```
df.describe(include="O")
```

Out[10]:

	labor_status	self_employed	marital_status	degree	political_affiliation	environment	law_e
count	58439	58439	58439	58439	58439	58439	
unique	8	4	5	6	9	5	
top	WORKING FULLTIME	SOMEONE ELSE	MARRIED	HIGH SCHOOL	NOT STR DEMOCRAT	IAP	
freq	28960	48809	31376	30124	12272	37576	

In [11]:
```
df["environment"].value_counts()
```

Out[11]:
```
IAP            37576
TOO LITTLE     12971
ABOUT RIGHT     5351
TOO MUCH        1660
DK               881
Name: environment, dtype: int64
```

Manipulating categorical data

- The categorical variable type can be useful, especially here:
  - It is possible to specify a precise order to the categories when the default order may be incorrect (e.g., via alphabetical).
  - Can be compatible with other Python libraries.

Let's take our existing categorical variables and convert them from strings to categories. Here, we use [`.select_dtypes()` (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html)](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html) to return only object columns, and with a dictionary set their type to be a category.

In [13]:
```
# Create a dictionary of column and data type mappings
conversion_dict = {k: "category" for k in df.select_dtypes(include="object"

# Convert our DataFrame and check the data types
df = df.astype(conversion_dict)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 58439 entries, 0 to 59598
Data columns (total 15 columns):
 #   Column                Non-Null Count  Dtype   
---  ------                --------------  -----   
 0   year                  58439 non-null  float64 
 1   id                    58439 non-null  float64 
 2   labor_status          58439 non-null  category
 3   self_employed         58439 non-null  category
 4   marital_status        58439 non-null  category
 5   n_siblings            58439 non-null  float64 
 6   age                   58439 non-null  float64 
 7   high_school           58439 non-null  float64 
 8   degree                58439 non-null  category
 9   political_affiliation 58439 non-null  category
 10  environment           58439 non-null  category
 11  law_enforcement       58439 non-null  category
 12  drugs                 58439 non-null  category
 13  space_exploration     58439 non-null  category
 14  household_size        58439 non-null  float64 
dtypes: category(9), float64(6)
memory usage: 3.6 MB
```

Already we can see that the memory usage of the DataFrame has been halved from 7 mb to about 4 mb, optmizing the data.

Cleaning up the `labor_status` column

To analyze the relationship between employment and attitudes over time, we need to clean up the `labor_status` column. We can preview the existing categories using `.categories`.

In [14]:
```
df["labor_status"].values
```

Out[14]:
```
['WORKING FULLTIME', 'RETIRED', 'WORKING PARTTIME', 'WORKING FULLTIME',
'KEEPING HOUSE', ..., 'KEEPING HOUSE', 'WORKING FULLTIME', 'WORKING FULLT
IME', 'WORKING FULLTIME', 'WORKING PARTTIME']
Length: 58439
Categories (8, object): ['KEEPING HOUSE', 'OTHER', 'RETIRED', 'SCHOOL',
'TEMP NOT WORKING', 'UNEMPL, LAID OFF', 'WORKING FULLTIME', 'WORKING PART
TIME']
```

In [15]:
```
df["labor_status"].value_counts()
```

Out[15]:
```
WORKING FULLTIME    28960
KEEPING HOUSE        9478
RETIRED              7861
WORKING PARTTIME     6012
UNEMPL, LAID OFF     1920
SCHOOL               1807
TEMP NOT WORKING     1240
OTHER                1161
Name: labor_status, dtype: int64
```

Let's collapse some of these categories. The easiest way to do this is to replace the values inside the column using a dictionary, and then reset the data type back to a category.

In [16]:
```
# Create a dictionary of categories to collapse
new_labor_status = {"UNEMPL, LAID OFF": "UNEMPLOYED",
                    "TEMP NOT WORKING": "UNEMPLOYED",
                    "WORKING FULLTIME": "EMPLOYED",
                    "WORKING PARTTIME": "EMPLOYED"
                   }

# Replace the values in the column and reset as a category
df["labor_status_clean"] = df["labor_status"].replace(new_labor_status).ast
print(df.dtypes)
# Preview the new column
df["labor_status_clean"].value_counts()
```

```
year                     float64
id                       float64
labor_status            category
self_employed           category
marital_status          category
n_siblings               float64
age                      float64
high_school              float64
degree                  category
political_affiliation   category
environment             category
law_enforcement         category
drugs                   category
space_exploration       category
household_size           float64
labor_status_clean      category
dtype: object
```

Out[16]:
```
EMPLOYED         34972
KEEPING HOUSE     9478
RETIRED           7861
UNEMPLOYED        3160
SCHOOL            1807
OTHER             1161
Name: labor_status_clean, dtype: int64
```

Reordering categories

In [17]:
```
df["environment"].values
```

Out[17]:
```
['IAP', 'IAP', 'IAP', 'IAP', 'IAP', ..., 'TOO LITTLE', 'TOO LITTLE', 'TOO
LITTLE', 'IAP', 'IAP']
Length: 58439
Categories (5, object): ['ABOUT RIGHT', 'DK', 'IAP', 'TOO LITTLE', 'TOO M
UCH']
```

In [21]:
```
# Set the new order
new_order = ["TOO LITTLE", "ABOUT RIGHT", "TOO MUCH", "DK", "IAP"]
categories_to_remove = ["DK", "IAP"]

# Loop through each column
for col in ["environment", "law_enforcement", "drugs"]:
    # Reorder and remove the categories
    df[col + "_clean"] = df[col].cat.reorder_categories(new_order, ordered=
    df[col + "_clean"] = df[col + "_clean"].cat.remove_categories(categorie

# Preview one of the columns' categories
df["environment_clean"].cat.categories
```

Out[21]:
```
Index(['TOO LITTLE', 'ABOUT RIGHT', 'TOO MUCH'], dtype='object')
```

Now let's also apply these steps to education level in one go: collapsing, removing, and reording.

In [22]:
```
df['degree'].values #let's reorder that and remove 'DK'
```

Out[22]:
```
['BACHELOR', 'LT HIGH SCHOOL', 'HIGH SCHOOL', 'BACHELOR', 'HIGH SCHOOL',
..., 'JUNIOR COLLEGE', 'HIGH SCHOOL', 'HIGH SCHOOL', 'HIGH SCHOOL', 'HIGH
SCHOOL']
Length: 58439
Categories (6, object): ['BACHELOR', 'DK', 'GRADUATE', 'HIGH SCHOOL', 'JU
NIOR COLLEGE', 'LT HIGH SCHOOL']
```

In [23]:
```
# Define a dictionary to map old degree categories to new ones
new_degree = {"LT HIGH SCHOOL": "HIGH SCHOOL",
              "BACHELOR": "COLLEGE/UNIVERSITY",
              "GRADUATE": "COLLEGE/UNIVERSITY",
              "JUNIOR COLLEGE": "COLLEGE/UNIVERSITY"}

# Replace old degree categories with new ones and convert to categorical da
df["degree_clean"] = df["degree"].replace(new_degree).astype("category")

# Remove "DK" category from degree_clean column
df["degree_clean"] = df["degree_clean"].cat.remove_categories(["DK"])

# Reorder degree_clean categories and set as ordered
df["degree_clean"] = df["degree_clean"].cat.reorder_categories(["HIGH SCHOO

# Preview the new column
df["degree_clean"].value_counts()
```

Out[23]:
```
HIGH SCHOOL           42756
COLLEGE/UNIVERSITY    15660
Name: degree_clean, dtype: int64
```

By `IntervalIndex` [(https://pandas.pydata.org/docs/reference/api/pandas.IntervalIndex.html)](https://pandas.pydata.org/docs/reference/api/pandas.IntervalIndex.html) we set cutoff ranges for the `year` . We then use `pd.cut()` [(https://pandas.pydata.org/docs/reference/api/pandas.cut.html)](https://pandas.pydata.org/docs/reference/api/pandas.cut.html) to cut our `year` column by these ranges, and set labels for each range.

In [27]:
```
decade_boundaries = [(1970, 1979),(1979,1989) , (1989, 1999), (1999, 2009),
# Set the bins and cut the DataFrame

bins = pd.IntervalIndex.from_tuples(decade_boundaries)#this method returns

decade_labels = {bins[0]:'1970s',
                 bins[1]: '1980s',
                 bins[2]:'1990s',
                 bins[3]:'2000s',
                bins[4]:'2010s'}


print(bins)
df["decade"] = pd.cut(df["year"], bins)#creates a new column based on the y
print(df['decade'].values, df['decade'].dtypes)

# Rename each of the intervals of decade_boundaries as the decades on decad
df['decade']=df["decade"].replace(decade_labels)#.astype('category')

# Preview the new column
df[['year','decade']]
```

```
IntervalIndex([(1970, 1979], (1979, 1989], (1989, 1999], (1999, 2009], (2
009, 2019]], dtype='interval[int64, right]')
[(1970, 1979], (1970, 1979], (1970, 1979], (1970, 1979], (1970, 1979],
..., (2009, 2019], (2009, 2019], (2009, 2019], (2009, 2019], (2009, 201
9]]
Length: 58439
Categories (5, interval[int64, right]): [(1970, 1979] < (1979, 1989] < (1
989, 1999] < (1999, 2009] < (2009, 2019]] category
```

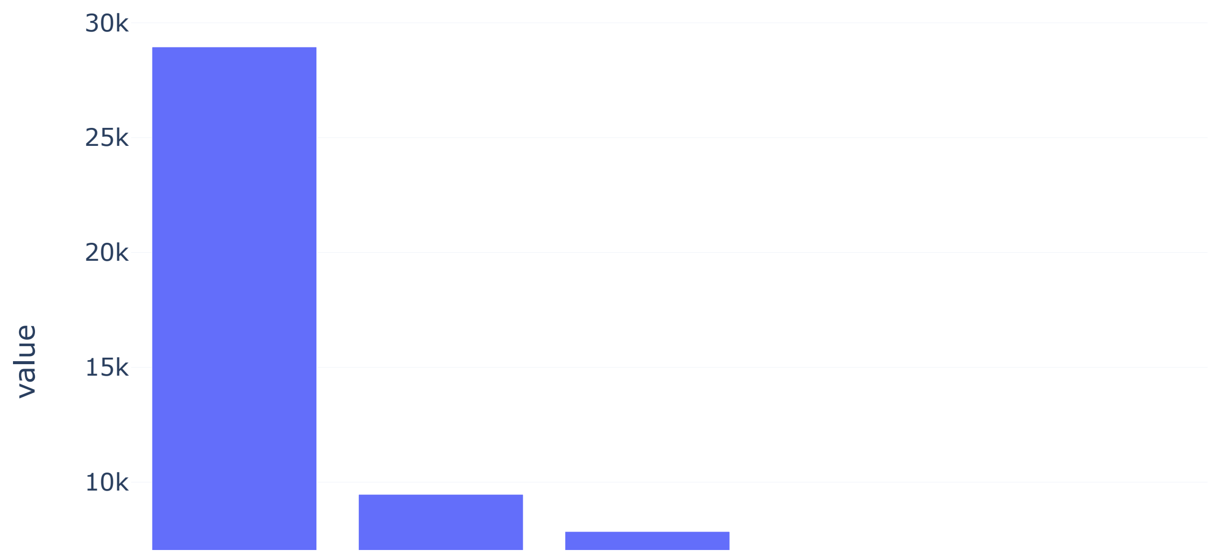
Out[27]:

	year	decade
0	1972.0	1970s
1	1972.0	1970s
2	1972.0	1970s
3	1972.0	1970s
4	1972.0	1970s
...	...	...
59594	2014.0	2010s
59595	2014.0	2010s
59596	2014.0	2010s
59597	2014.0	2010s
59598	2014.0	2010s

58439 rows × 2 columns

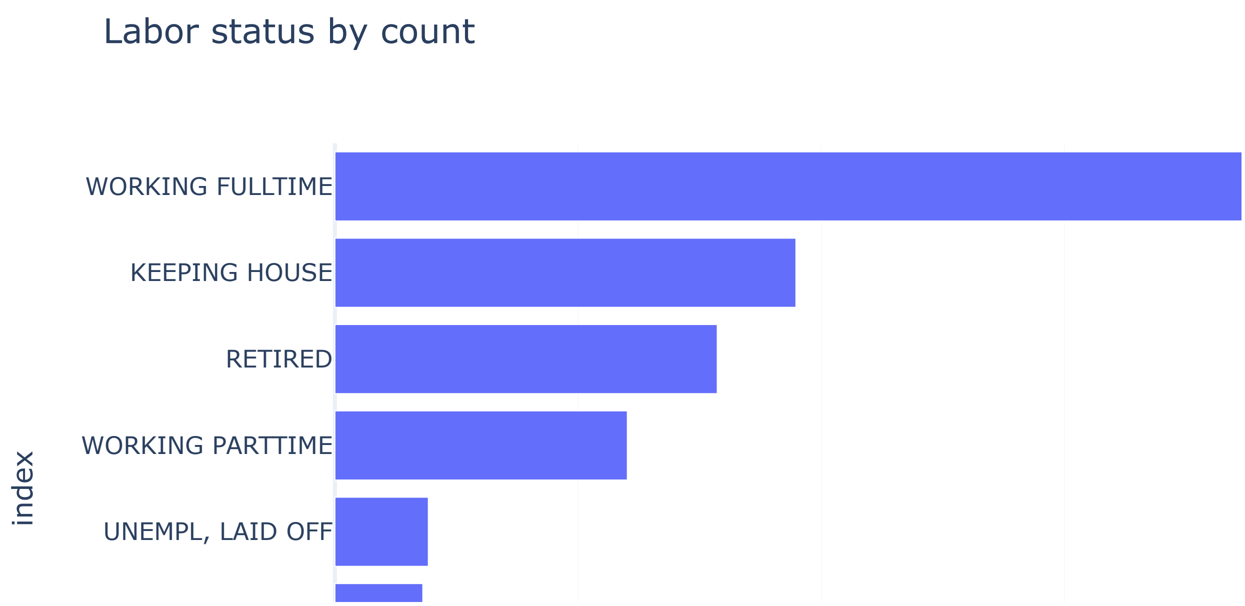
Visualization

In [28]:
```
# Create a new figure object
fig = px.bar(df["labor_status"].value_counts(),
             template="plotly_white"
            )

# Hide the legend and show the plot
fig.update_layout(showlegend=False)
fig.show()
```



Let's change the orientation of the plot and add a title, for a better perspective.

In [96]:
```
# Create a new figure object
fig = px.bar(df["labor_status"].value_counts(ascending=True),
             template="plotly_white",
             orientation="h",
             title="Labor status by count"
            )

# Hide the legend and show the plot
fig.update_layout(showlegend=False)
fig.show()
```



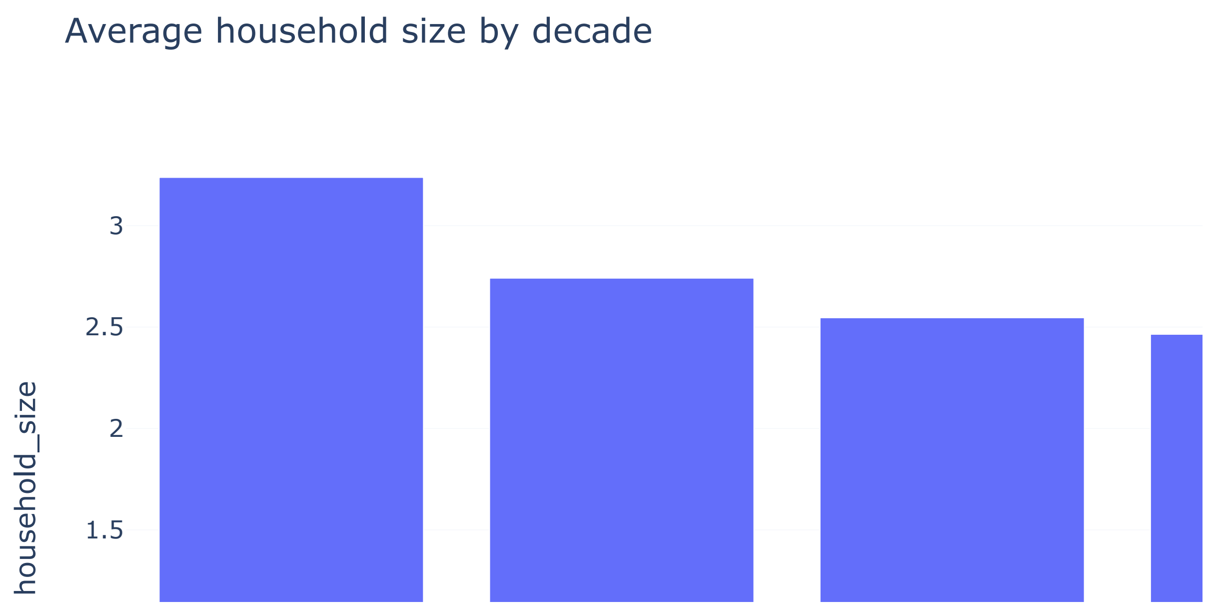
Bar charts

In [97]:
```
## Aggregate household size by year
household_by_decade = df.groupby("decade",as_index=False)["household_size"]
household_by_decade
```

Out[97]:

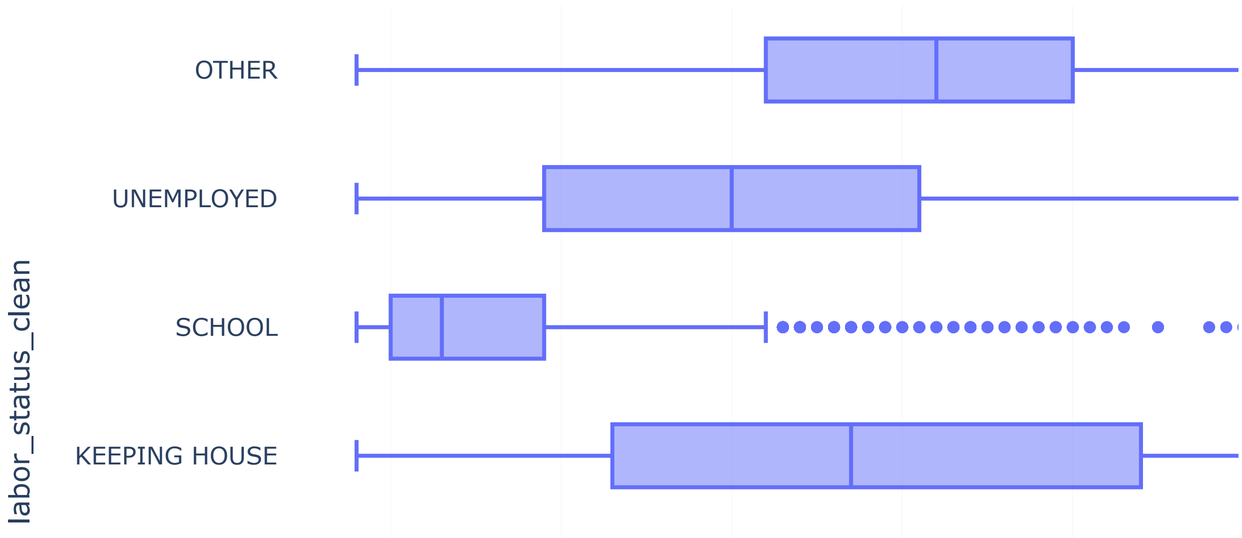
	decade	household_size
0	1970s	3.238023
1	1980s	2.741394
2	1990s	2.546036
3	2000s	2.464682
4	2010s	2.411328

In [98]:
```
# Create a new figure object
fig = px.bar(household_by_decade,
             x="decade",
             y="household_size",
             template="plotly_white",
             title="Average household size by decade"
            )

fig.show()
```



Boxplots

In [99]:
```
# Create a new figure object
fig = px.box(df,
             x="age",
             y="labor_status_clean",
             template="plotly_white"
            )

fig.show()
```



Mosaic plots

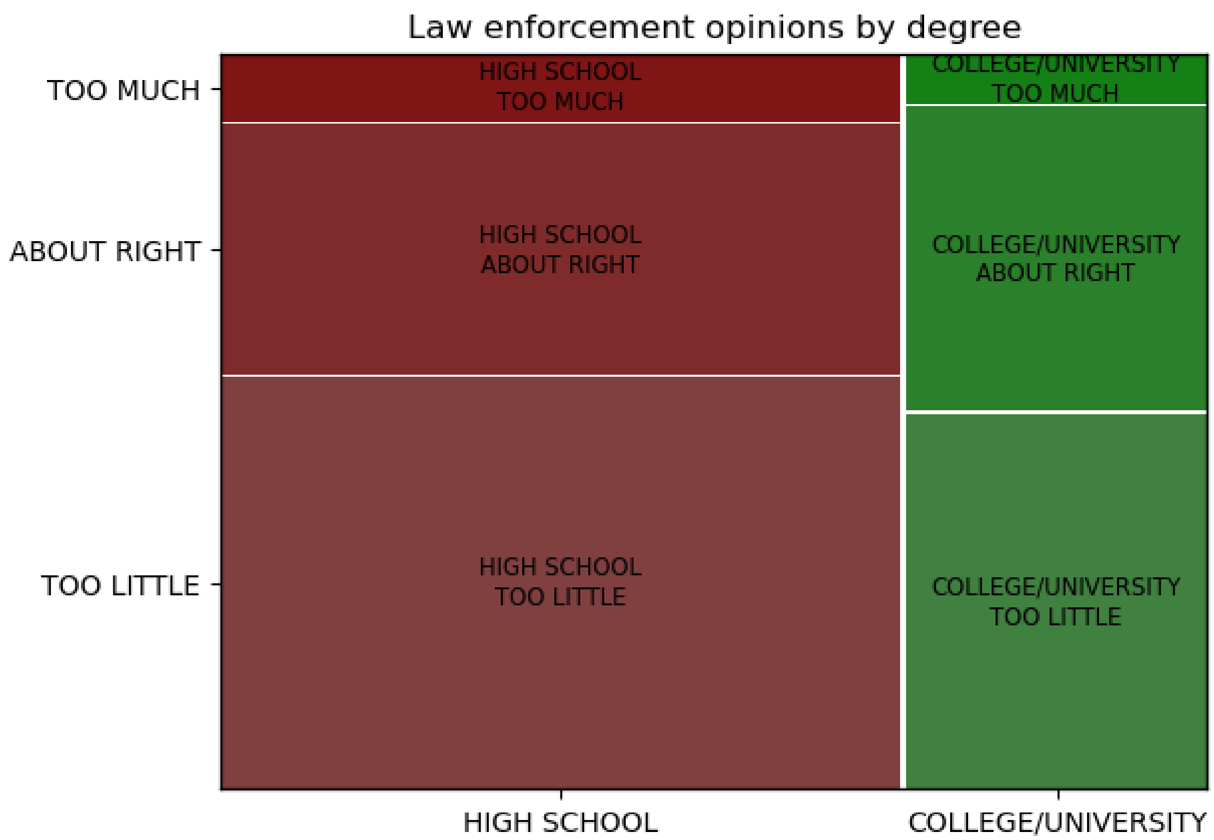
visualize the relationship between two categorical variables. One way to do this is a frequency table, which will give the counts across the different combinations of the two variables. create a frequency table using [`pd.crosstab()` (https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html)](https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html)

In [100]:
```
pd.crosstab(df["degree_clean"], df["law_enforcement_clean"])
```

Out[100]:

law_enforcement_clean	TOO LITTLE	ABOUT RIGHT	TOO MUCH
degree_clean			
HIGH SCHOOL	7937	4799	1309
COLLEGE/UNIVERSITY	3193	2598	422

In [101]:
```
# Create a mosaic plot and show it
mosaic(df,
       ['degree_clean', 'law_enforcement_clean'],
       title='Law enforcement opinions by degree')

plt.show()
```



Line charts

The final plot type we will cover is a line plot. Line plots often (but not always!) show the relationship between time and a numerical variable. Adding in a categorical variable can be a great way to enrich a line plot and provide other information.

Here, we use the `.value_counts()` method as an aggregation function, and use this in combination with a Plotly `line_plot()` (https://plotly.com/python/line-charts/) to visualize the trend in marital statuses over the years.

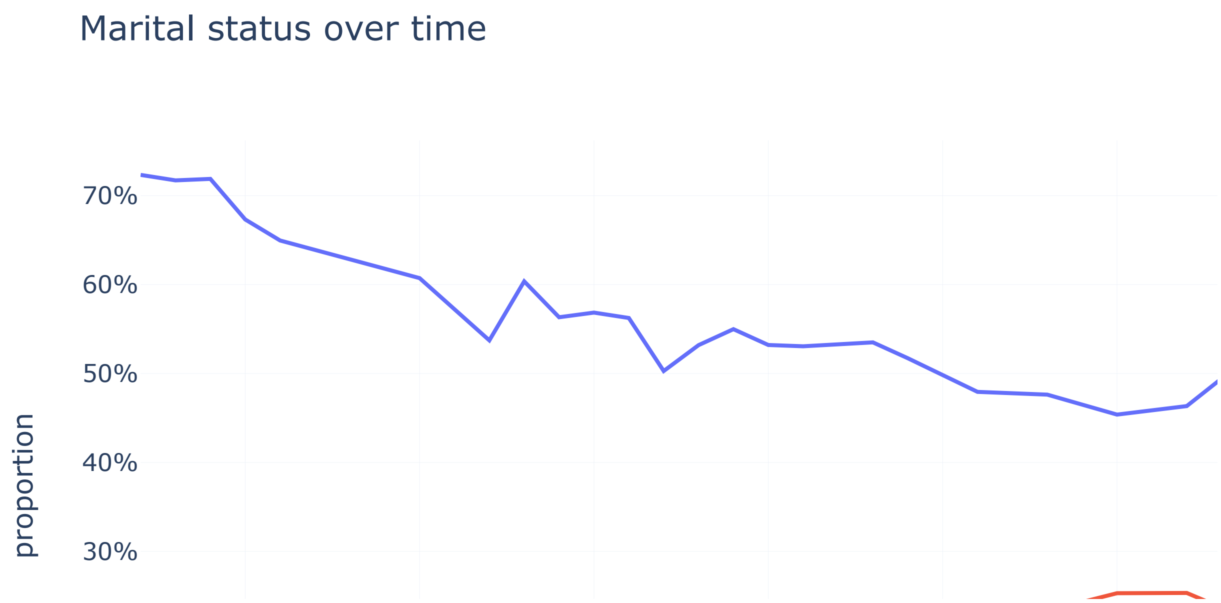
In [102]:
```
# Group the dataframe by year and marital status, and calculate the normali
marital_rates = df.groupby(["year"], as_index=False)["marital_status"].valu

# Display the resulting DataFrame
marital_rates
```

Out[102]:

	year	marital_status	proportion
0	1972.0	MARRIED	0.723391
1	1972.0	NEVER MARRIED	0.128744
2	1972.0	WIDOWED	0.084130
3	1972.0	DIVORCED	0.039516
4	1972.0	SEPARATED	0.024219
...	...	...	...
145	2014.0	MARRIED	0.458918
146	2014.0	NEVER MARRIED	0.265331
147	2014.0	DIVORCED	0.163126
148	2014.0	WIDOWED	0.081363
149	2014.0	SEPARATED	0.031263

150 rows × 3 columns

In [103]:
```
# Create a new figure object
fig = px.line(marital_rates,
              x="year",
              y="proportion",
              color="marital_status",
              template="plotly_white",
              title="Marital status over time"
             )

# Update the y-axis to show percentages
fig.update_yaxes(tickformat=".0%")

# Show the plot
fig.show()
```



In [ ]:

In [ ]: