Problem 1

The `laptops` DataFrame contains information on various factors that influence the pricing of laptops. Each row represents a laptop, and the columns are:

- `"Mfr" (str)`: the company that manufactures the laptop, like “Apple” or “Dell”.
- `"Model" (str)`: the model name of the laptop, such as “MacBook Pro”.
- `"OS" (str)`: the operating system, such as “macOS” or “Windows 11”.
- `"Screen Size" (float)`: the diagonal length of the screen, in inches.
- `"Price" (float)`: the price of the laptop, in dollars.

Problem 1.1

Without using `groupby`, write an expression that evaluates to the average price of laptops with the `"macOS"` operating system (the same quantity as above).

Click to view the solution.

Problem 1.2

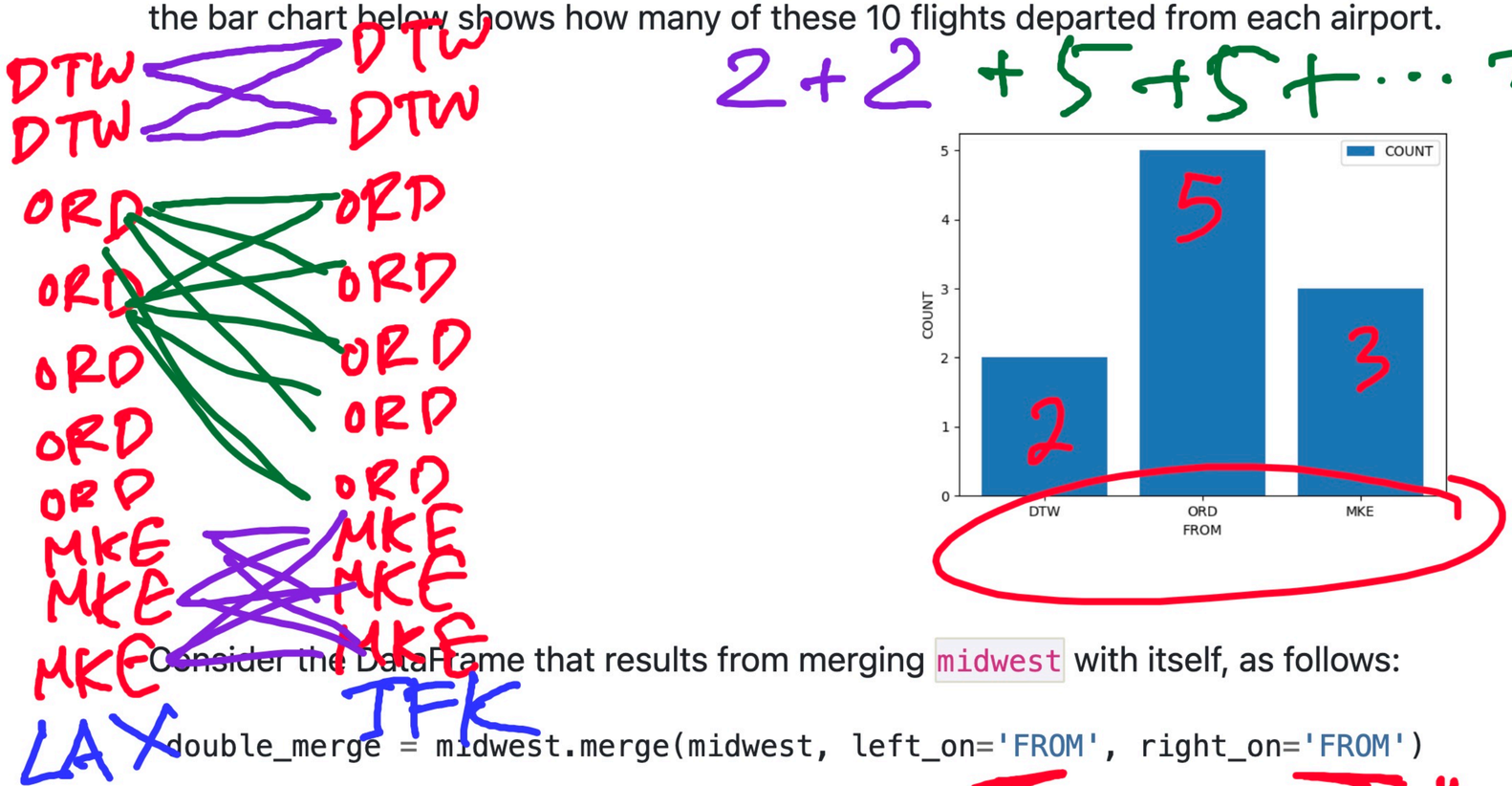
Using `groupby`, write an expression that evaluates to the average price of laptops with the `"macOS"` operating system.

Click to view the solution.

Click to view the solution.

Problem 4

Suppose we create a DataFrame called `midwest` containing Nishant's flights departing from DTW, ORD, and MKE. `midwest` has 10 rows; the bar chart below shows how many of these 10 flights departed from each airport.



Consider the DataFrame that results from merging `midwest` with itself, as follows:

```
double_merge = midwest.merge(midwest, left_on='FROM', right_on='FROM')
```

How many rows does `double_merge` have?

Click to view the solution.

Problem 5

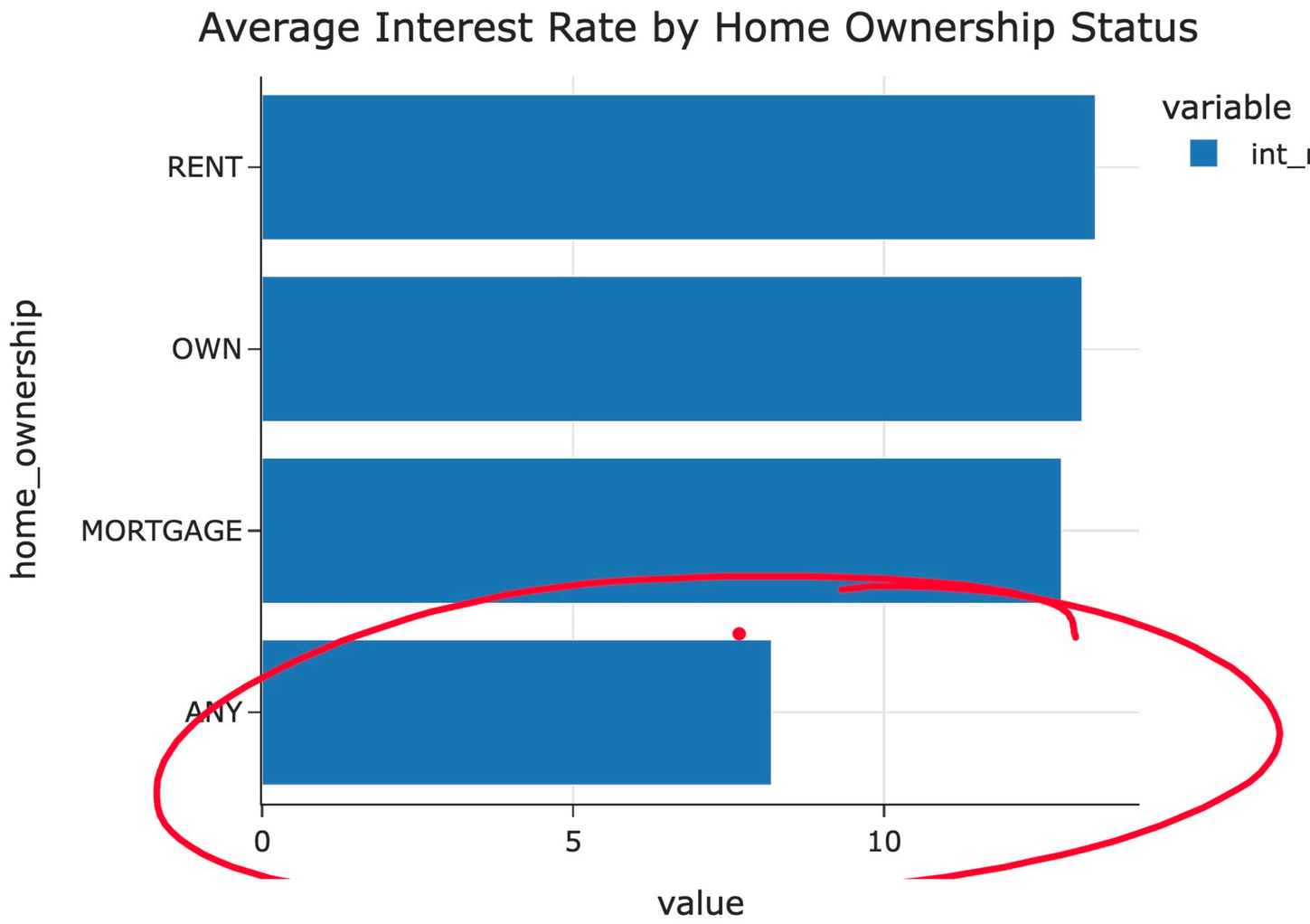
terms and interest rates:

In [3]:
```
1 display_df(loans.loc[loans['loan_amnt'] == 3600, ['loan_amnt', 'term', 'int_rate']], rows=17)
```

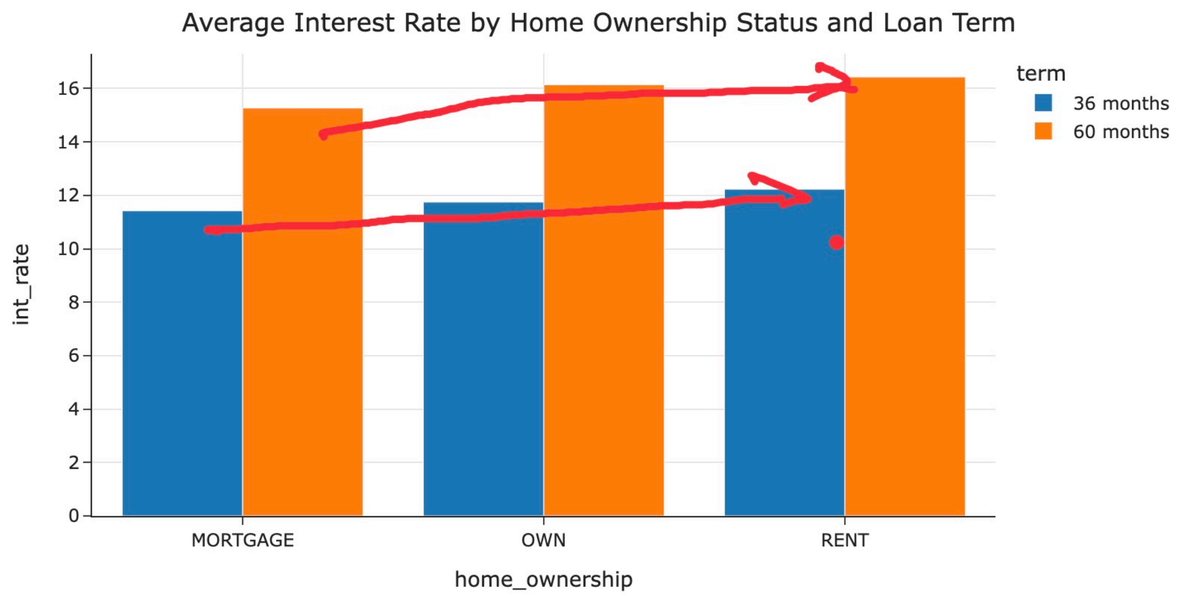
	loan_amnt	term	int_rate
249	3600.0	36	24.50
626	3600.0	36	13.99
1020	3600.0	36	11.49
2141	3600.0	36	10.08
2145	3600.0	36	5.32
2584	3600.0	36	16.29
2739	3600.0	36	8.24
3845	3600.0	36	13.66
4153	3600.0	36	12.59
4368	3600.0	36	14.08
4575	3600.0	36	16.46
4959	3600.0	36	10.75
4984	3600.0	36	13.99
5478	3600.0	60	10.59
5560	3600.0	36	19.99
5693	3600.0	36	13.99
6113	3600.0	36	15.59

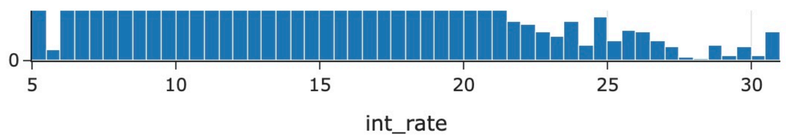
very different!

```
    .groupby('home_ownership')
    ['int_rate']
    .mean()
    .plot(kind='barh', title='Average Interest Rate by Home Ownership Sta
)
```



```
    loans
    .assign(term=loans['term'].astype(str) + ' months')
    .groupby('home_ownership')
    .filter(lambda df: df.shape[0] > 1)
    .groupby(['home_ownership', 'term'])
    [['int_rate']]
    .mean()
    .reset_index()
    .plot(kind='bar',
          y='int_rate',
          x='home_ownership',
          color='term',
          barmode='group',
          title='Average Interest Rate by Home Ownership Status and Loan Term',
          width=800)
)
```



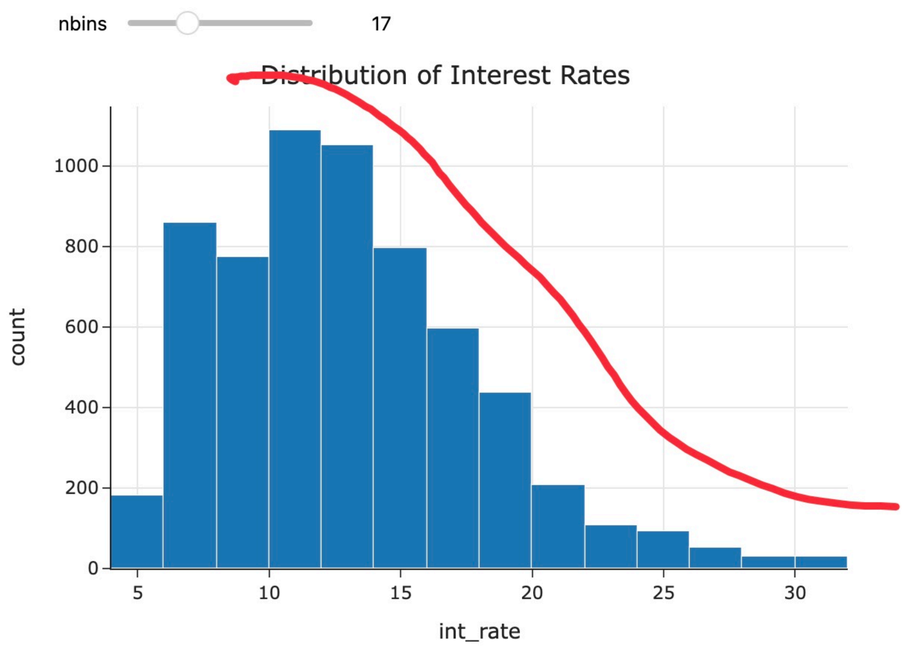


- With fewer bins, we see less detail (and less noise) in the shape of the distribution.

  Play with the slider that appears when you run the cell below!

In [45]:

```
def hist_bins(nbins):
    (
        loans
        .plot(kind='hist', x='int_rate', nbins=nbins, title='Distribution of Interest Rates')
        .show()
    )
interact(hist_bins, nbins=(1, 51));
```



right skewed / right tailed.

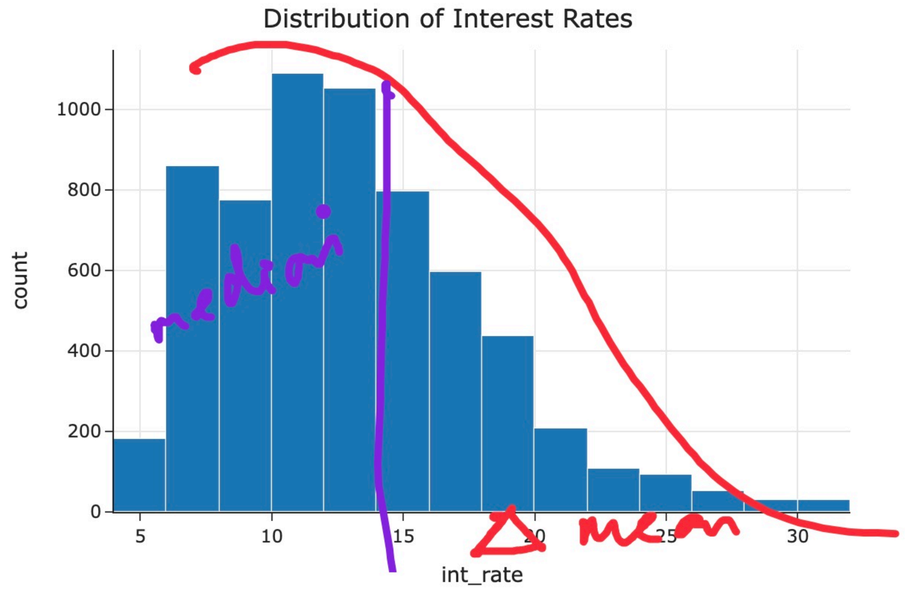
Remember that you can always ask questions anonymously at the link above!

Based on the histogram below, what is the relationship between the mean and median interest rate?

- A. Mean > median.
- B. Mean ≈ median.
- C. Mean < median.

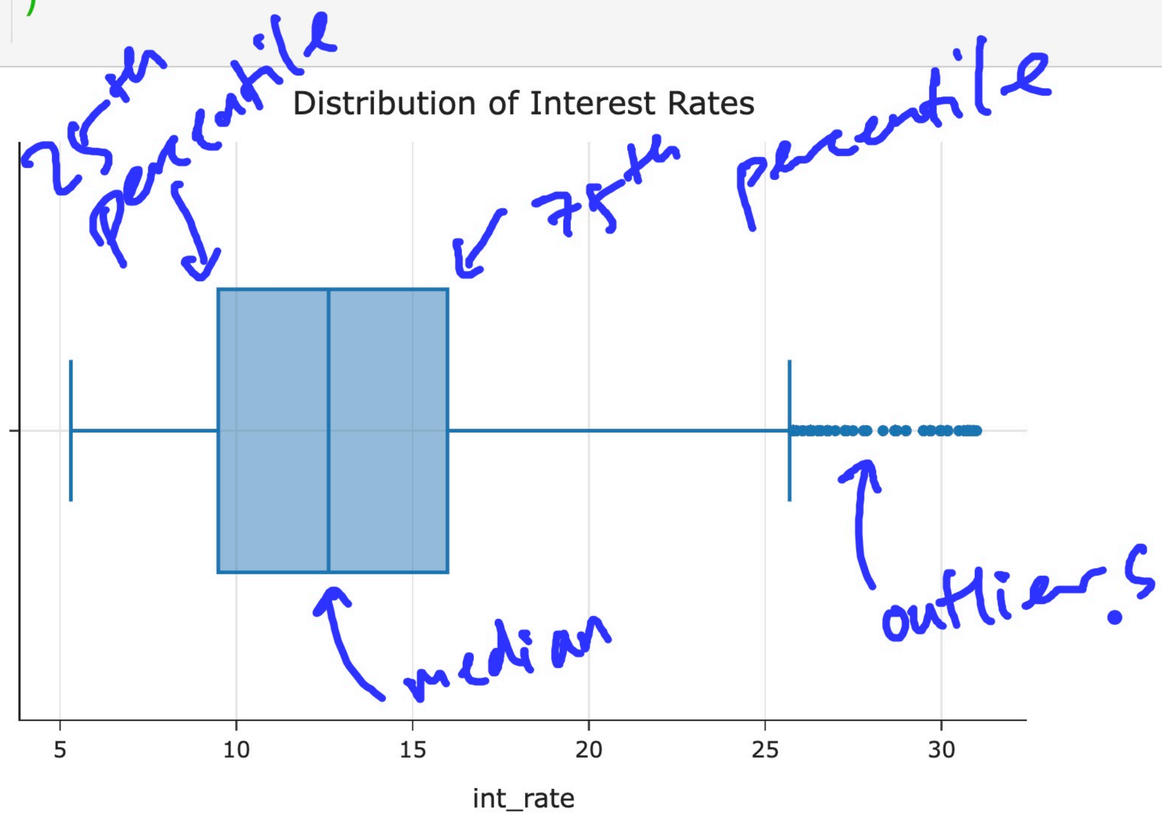
In [46]:

```
(
    loans
    .plot(kind='hist', x='int_rate', title='Distribution of Interest Rates', nbins=20)
)
```



- Example: What is the distribution of `'int_rate'`?

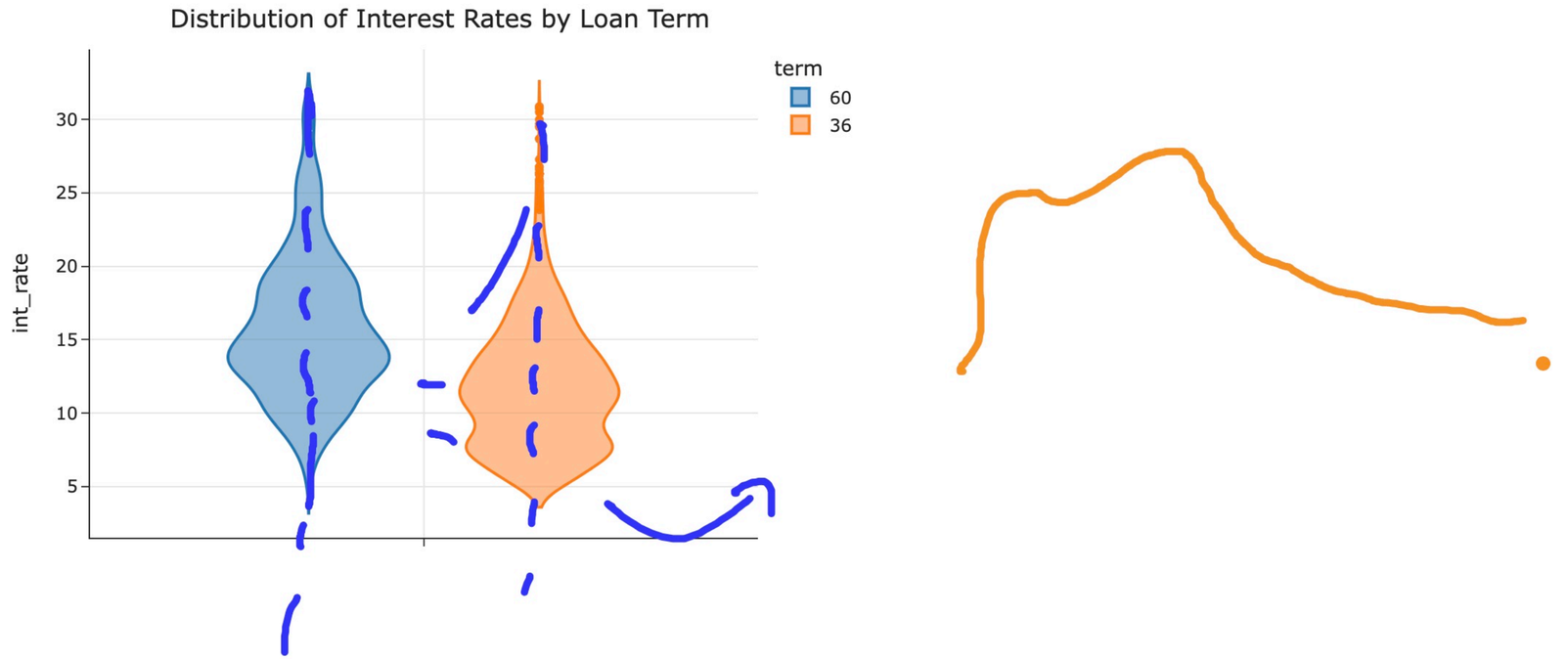
In [47]:
```
(
    loans
    .plot(kind='box', x='int_rate', title='Distribution of Interest Rates')
)
```

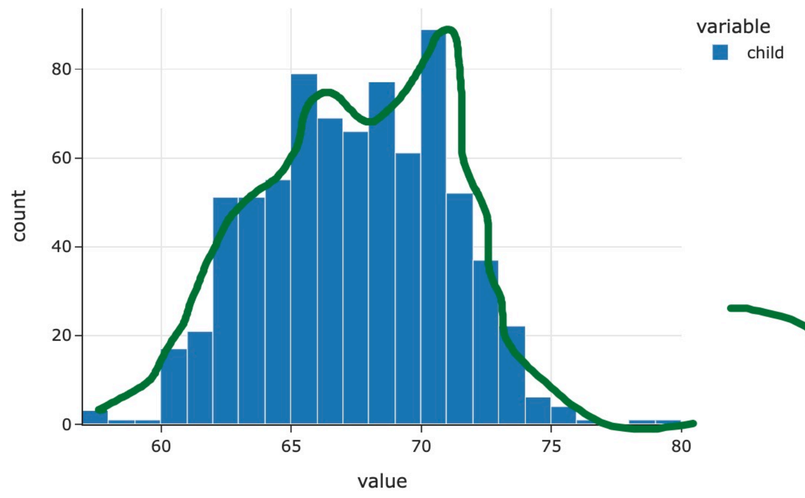


?

In [49]:

```
(
    loans
    .plot(kind='violin', y='int_rate', color='term', orientation='v',
          title='Distribution of Interest Rates by Loan Term')
)
```

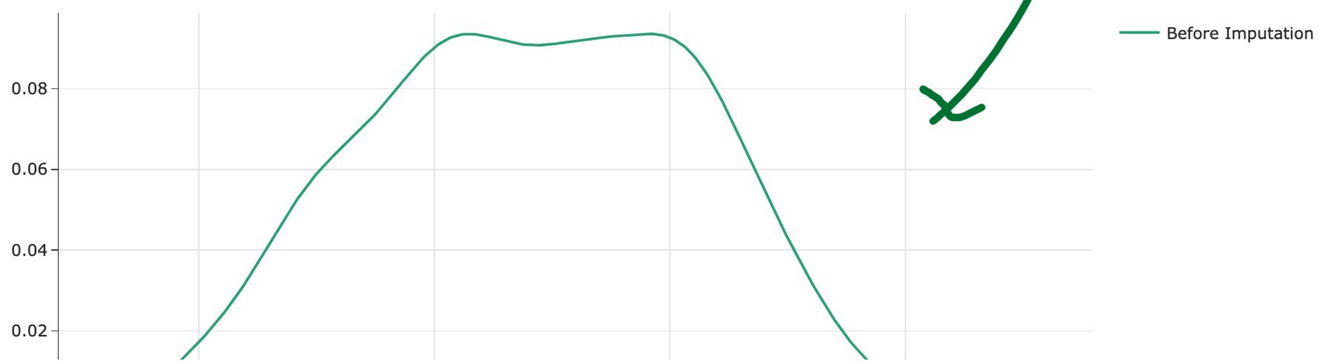




smooth histogram

In [81]:

```
def multiple_kdes(ser_map, title=""):
    values = [ser_map[key].dropna() for key in ser_map]
    labels = list(ser_map.keys())
    fig = ff.create_distplot(
        hist_data=values,
        group_labels=labels,
        show_rug=False,
        show_hist=False,
        colors=px.colors.qualitative.Dark2[: len(ser_map)],
    )
    return fig.update_layout(title=title, width=1000).update_xaxes(title="child")
multiple_kdes({'Before Imputation': heights['child']})
```



```
2    69.0
3    69.0
4    67.1
Name: child, dtype: float64
```

- The mean of `mean_imputed` is the **same** as the mean of `'child'` **before** we imputed.

  You proved this in Homework 1!

In [99]:
```
# Mean before imputation:
heights['child'].mean()
```

Out[99]:
```
67.10339869281046
```

In [98]:
```
# Mean after imputation:
mean_imputed.mean()
```

Out[98]:
```
67.10339869281046
```

- What do you think a *histogram* of `mean_imputed` would look like?

In [100]:
```
mean_imputed.value_counts()
```

Out[100]:
```
child
67.1    169
70.0     54
68.0     50
       ...
63.2      1
62.2      1
59.0      1
Name: count, Length: 64, dtype: int64
```

number of missing values!

```
heights.groupby('gender')['child'].mean()
```

Out[106]:

```
gender
female    64.03
male      69.13
Name: child, dtype: float64
```

In [112]:

```
heights.head()
```

Out[112]:

	father	mother	gender	child
0	78.5	67.0	male	NaN
1	78.5	67.0	female	69.2
2	78.5	67.0	female	69.0
3	78.5	67.0	female	69.0
4	75.5	66.5	male	NaN

In [113]:

```
# Note the first missing 'child' height is filled in with
# 69.13, the mean of the observed 'male' heights, since
# they are a 'male' child!
conditional_mean_imputed = (
    heights
    .groupby('gender')
    ['child']
    .transform(lambda s: s.fillna(s.mean()))
)
conditional_mean_imputed.head()
```

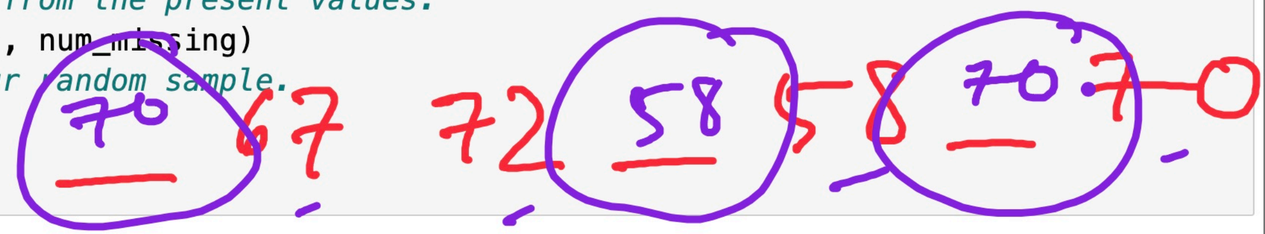
Out[113]:

```
0    69.13
1    69.20
2    69.00
3    69.00
4    69.13
Name: child, dtype: float64
```

- We could fill in missing values using a **random sample** of observed values.

  This avoids the key issue with mean imputation, where we fill all missing values with the same one value. It also limits the bias present if the missing values weren't a representative sample, since we're filling them in with a range of different values.

In [118]:
```
# impute_prob should take in a Series with missing values and return an imputed Series.
def impute_prob(s):
    s = s.copy()
    # Find the number of missing values.
    num_missing = s.isna().sum()
    # Take a sample of size num_missing from the present values.
    sample = np.random.choice(s.dropna(), num_missing)
    # Fill in the missing values with our random sample.
    s.loc[s.isna()] = sample
    return s
```



- Each time we run the cell below, the missing values in `heights['child']` are filled in with a different sample of present values in `heights['child']`!

In [ ]:
```
# The number at the very top is constantly changing!
prob_imputed = impute_prob(heights['child'])
print('Mean:', prob_imputed.mean())
prob_imputed
```