In [1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab4-smoothing.ipynb")
```

In [2]:
```
import numpy as np
import pandas as pd
import altair as alt
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

pd.options.mode.chained_assignment = None  # default='warn'
```

Lab 4: Smooth visuals

So far, you've encountered a number of visualization techniques for displaying tidy data. In those visualizations, all graphic elements represent the values of a dataset -- they are visual displays of actual data.

In general, smoothing means evening out. Visualizations of actual data are often irregular -- points are distributed widely in scatterplots, line plots are jagged, bars are discontinuous. When we look at such visuals, we tend to attempt to look past these irregularities in order to discern patterns -- for example, the overall shape of a histogram or the general trend in a scatterplot. Showing what a graphic might look like with irregularities evened out often aids the eye in detecting pattern. This is what **smoothing** is: ***evening out irregularities in graphical displays of actual data***.

For our purposes, usually smoothing will consist in drawing a line or a curve on top of an existing statistical graphic. From a technical point of view, this amounts to adding derived geometric objects to a graphic that have fewer irregularities than the displays of actual data.

Objectives

In this lab, you'll learn some basic smoothing techniques -- kernel density estimation, LOESS, and linear smoothing via regression -- and how to implement them in Altair. Your focus will be on understanding the techniques *graphically*, not mathematically.

In Altair, smoothed geometric objects are constructed and plotted through what Altair describes as *transforms* -- operations that modify a dataset. Try not to get too attached to this terminology -- 'transform' and 'transformation' are used to mean a variety of things in other contexts. You'll begin with a brief introduction to Altair transforms before turning to smoothing techniques.

The **sections** of the lab are divided as follows:

1. Introduction to Altair transforms
2. Histogram smoothing: kernel density estimamtion
3. Scatterplot smoothing: LOESS and linear smoothing
4. A neat graphic

And our main **goals** are:

- Get familiar with Altair transforms for dataframe operations: filter, bin, aggregate, calculate.
- 'Handmande' histograms: step-by-step construction
- Implement kernel density estimation via `.transform_density(...)`
- Implement LOESS via `.transform_loess(...)`
- Implement linear smoothing via `transform_regression(...)`

You'll use the same data as last week to stick to a familiar example:

In [3]:
```
# import tidied lab 3 data
data = pd.read_csv('lab3-data.csv')
data.head()
```

Out[3]:

	Country Name	Year	Life Expectancy	Male Life Expectancy	Female Life Expectancy	GDP per capita	region	sub-region	Po
0	Afghanistan	2019	63.2	63.3	63.2	507.103432	Asia	Southern Asia	380
1	Afghanistan	2015	61.7	61.0	62.3	578.466353	Asia	Southern Asia	344
2	Afghanistan	2010	59.9	59.6	60.3	543.303042	Asia	Southern Asia	291
3	Albania	2019	78.0	76.3	79.9	5353.244856	Europe	Southern Europe	28
4	Albania	2015	77.8	76.1	79.7	3952.801215	Europe	Southern Europe	28

0. Background: transforms in Altair

In Altair, operations that modify a dataset are referred to as *transforms*. Mostly, these are operations that could be performed manually with ease -- the utility of transforms is that they *wrap common operations within plotting commands*, although they also make plotting codes more verbose.

Transforms encompass a broad range of types of operations, from relatively simple ones like filtering to more complex ones like smoothing. Here you'll see a few intuitive transforms in Altair that integrate simple dataframe manipulations into the plotting process.

You'll focus on the construction of histograms as a sort of case study. This will be a useful primer for histogram smoothing in the next section.

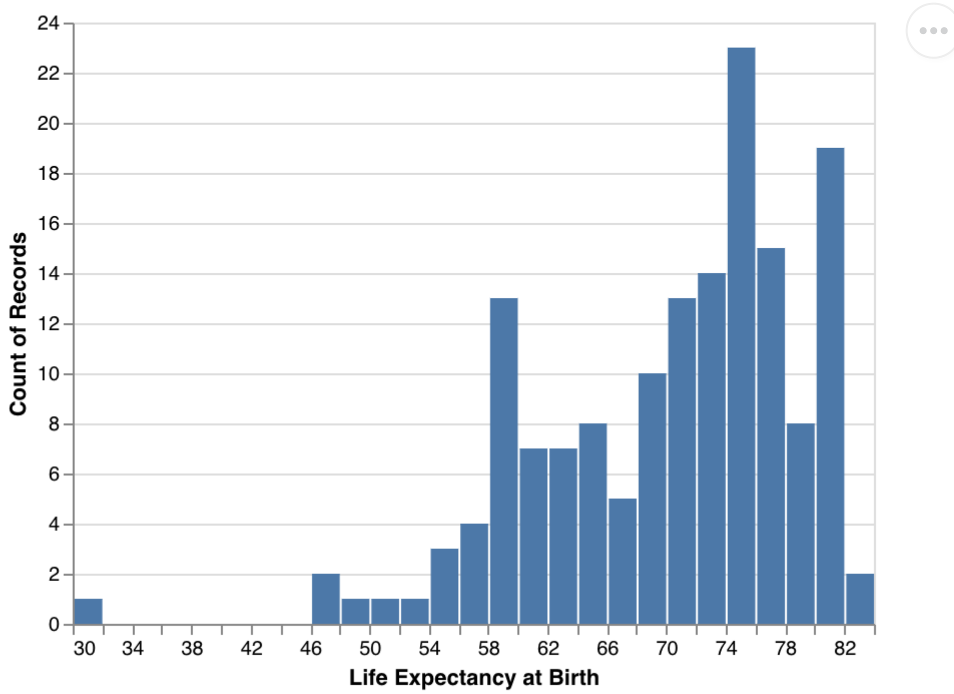
Filter transform

Last week you saw a way to make histograms. As a quick refresher, to make a histogram of life expectancies across the globe in 2010, one can filter the data and then plot using the following commands:

In [4]:
```
# filter
data2010 = data[data.Year == 2010]

# plot
alt.Chart(data2010).mark_bar().encode(
    x = alt.X('Life Expectancy',
              bin = alt.Bin(step = 2),
              title = 'Life Expectancy at Birth'),
    y = 'count()'
)
```

Out[4]:



However, the filtering step can be handled *within the plotting commands* using `.transform_filter()`.

This uses a helper command to specify the filtering condition -- in the above example, the filtering condition is that `Year` is equal to `2010`. A filtering condition is referred to in Altair as a 'field predicate'. In the above example:

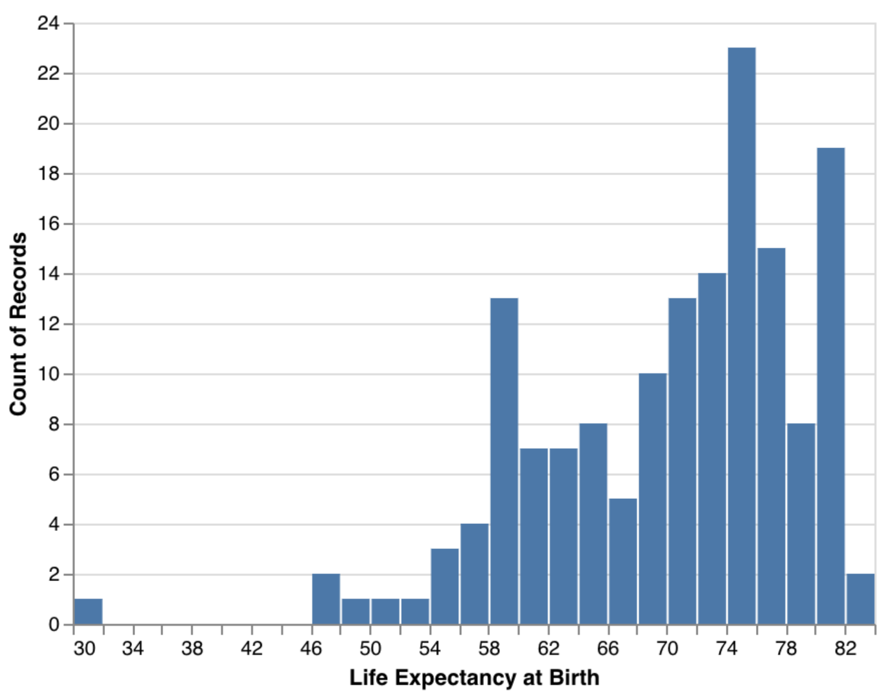
- filtering field: `Year`
- field predicate: equals `2010`

There are different helpers for different types of field predicates -- you can find a complete list in the documentation.

Here is how to use `.transform_filter()` to make the same histogram shown above, but skipping the step of storing a subset of the data under a separate name:

In [5]:
```
# filter and plot
alt.Chart(data).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
).mark_bar().encode(
    x = alt.X('Life Expectancy',
              bin = alt.Bin(step = 2),
              title = 'Life Expectancy at Birth'),
    y = 'count()'
)
```

Out[5]:

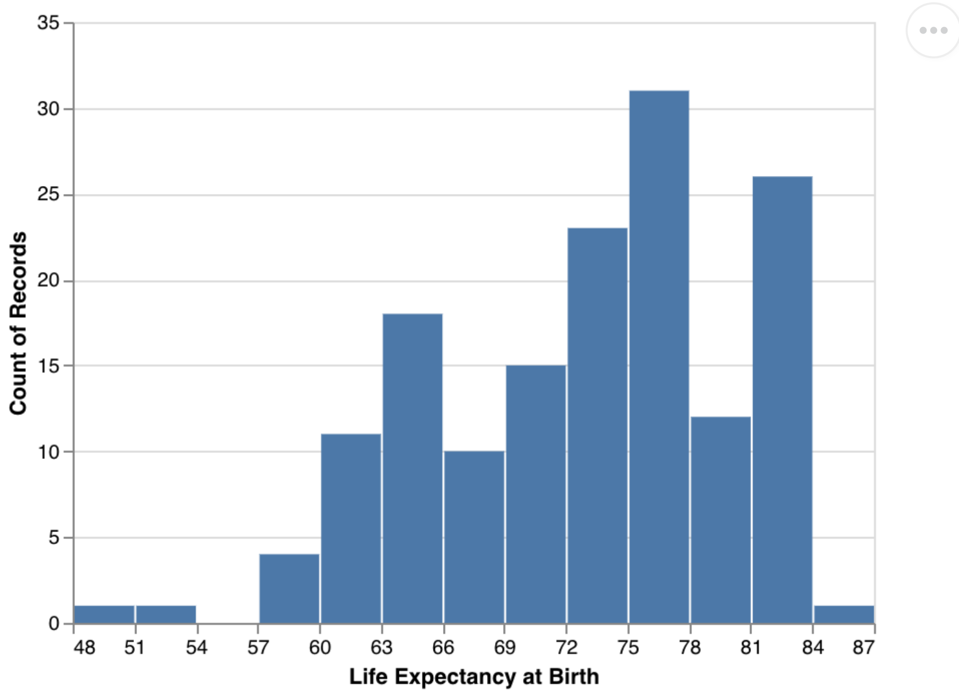


Question 0a. Filter transform

Construct a histogram of life expectancies across the globe in 2019 using a filter transform as shown above to filter the appropriate rows of the dataset. Use a bin size of three (not two) years.

In [6]:
```
# filter and plot
alt.Chart(data).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2019)
).mark_bar().encode(
    x = alt.X('Life Expectancy',
              bin = alt.Bin(step = 3),
              title = 'Life Expectancy at Birth'),
    y = 'count()'
)
```

Out[6]:



Bin transform

The codes above provide a sleek way to construct the histogram that handles binning via arguments to `alt.X(...)`. However, binning actually involves an operation: creating a new variable that is a discretization of an existing variable into contiguous intervals of a specified width.

To illustrate, have a look at how the histogram could be constructed 'manually' by the following operations.

1. Bin life expectancies
2. Count values in each bin

3. Make a bar plot of counts against bin centers.

Here's step 1:

In [7]:

```
# bin life expectancies into 20 contiguous intervals
data2010['Bin'] = pd.cut(data2010["Life Expectancy"], bins = 20)
data2010.head()
```

Out[7]:

	Country Name	Year	Life Expectancy	Male Life Expectancy	Female Life Expectancy	GDP per capita	region	sub-region
2	Afghanistan	2010	59.9	59.6	60.3	543.303042	Asia	Southern Asia
5	Albania	2010	76.2	74.2	78.3	4094.350334	Europe	Southern Europe
9	Algeria	2010	75.9	75.0	76.8	4479.341720	Africa	Northern Africa
13	Angola	2010	58.1	55.8	60.5	3587.883798	Africa	Sub-Saharan Africa
17	Antigua and Barbuda	2010	75.9	73.6	78.2	13049.257050	Americas	Latin America and the Caribbean

Here's step 2:

In [8]:

```
# count values in each bin and store midpoints
histdata = data2010.loc[:, ['Life Expectancy',
'Bin']].groupby('Bin').count()
histdata['Bin midpoint'] =
histdata.index.values.categories.mid.values
histdata
```

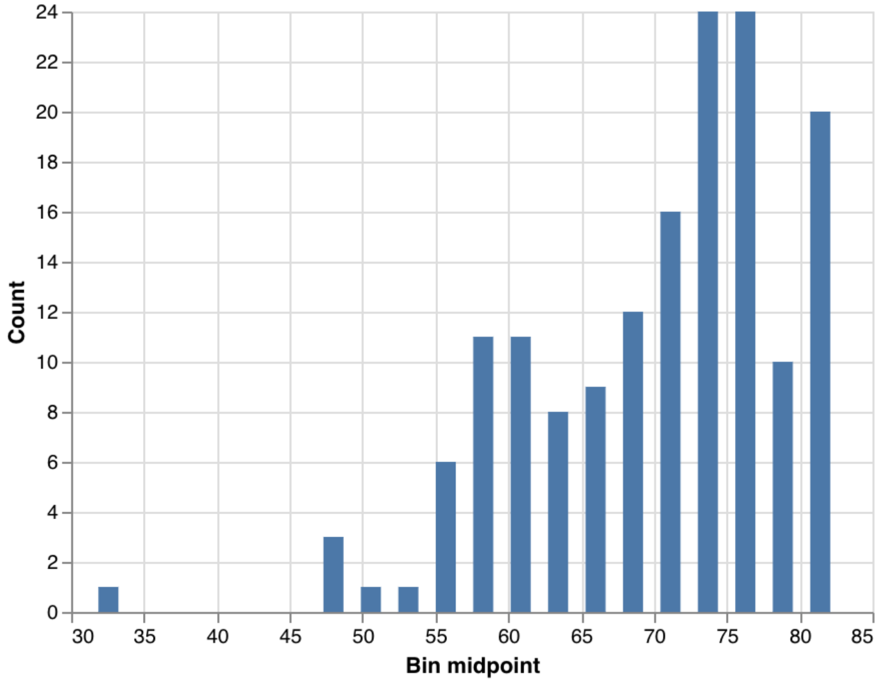
Out[8]:

Bin	Life Expectancy	Bin midpoint
(31.249, 33.87]	1	32.5595
(33.87, 36.44]	0	35.1550
(36.44, 39.01]	0	37.7250
(39.01, 41.58]	0	40.2950
(41.58, 44.15]	0	42.8650
(44.15, 46.72]	0	45.4350
(46.72, 49.29]	3	48.0050
(49.29, 51.86]	1	50.5750
(51.86, 54.43]	1	53.1450
(54.43, 57.0]	6	55.7150
(57.0, 59.57]	11	58.2850
(59.57, 62.14]	11	60.8550
(62.14, 64.71]	8	63.4250
(64.71, 67.28]	9	65.9950
(67.28, 69.85]	12	68.5650
(69.85, 72.42]	16	71.1350
(72.42, 74.99]	24	73.7050
(74.99, 77.56]	24	76.2750
(77.56, 80.13]	10	78.8450
(80.13, 82.7]	20	81.4150

And finally, step 3:

In [9]:
```
# plot histogram
alt.Chart(histdata).mark_bar(width = 10).encode(
    x = 'Bin midpoint',
    y = alt.Y('Life Expectancy', title = 'Count')
)
```

Out[9]:

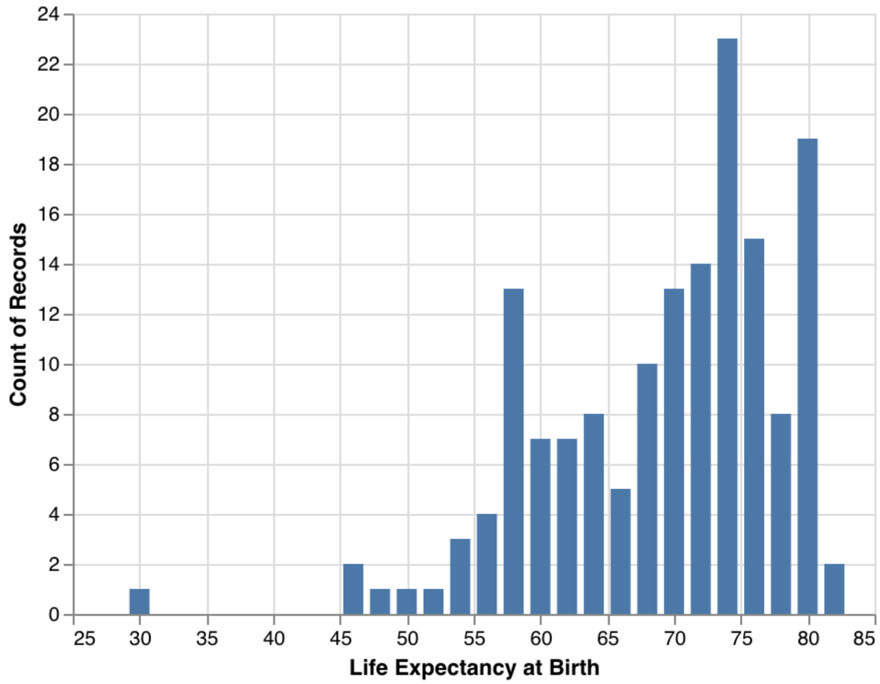


Well, these operations can be articulated as a transform in Altair using `.bin_transform()`:

In [10]:

```
# filter, bin, and plot
alt.Chart(
    data
).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
).transform_bin(
    'Life Expectancy at Birth', # name to give binned variable
    field = 'Life Expectancy', # variable to bin
    bin = alt.Bin(step = 2) # binning parameters
).mark_bar(size = 10).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'count()'
)
```

Out[10]:



The plotting codes are a little more verbose, but they're much more efficient than performing the manipulations separately in pandas!

Question 0b. Bin transform

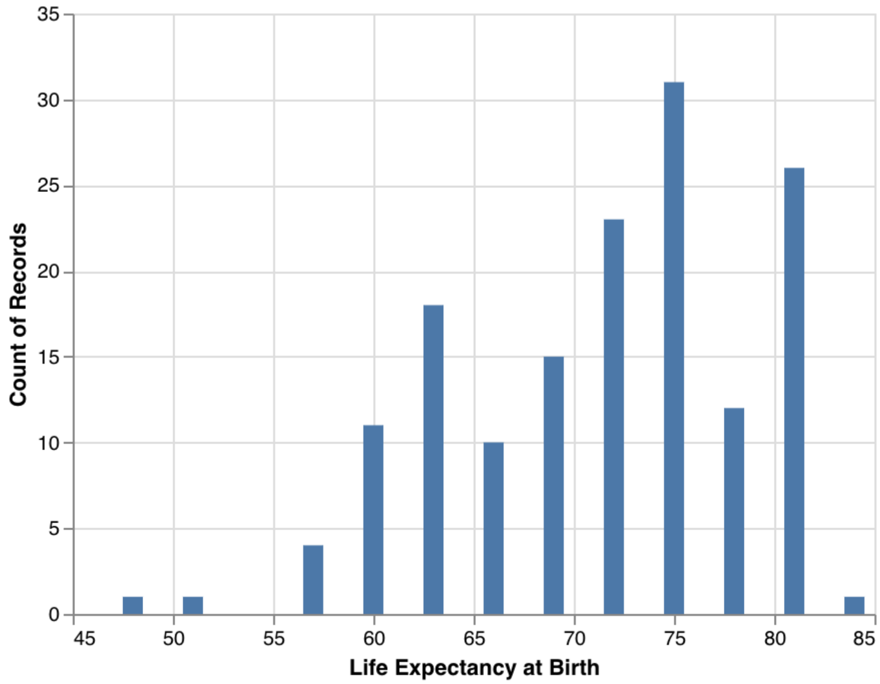
Follow the example above and make a histogram of life expectancies across the globe in 2019 using an explicit bin transform to create bins spanning three years.

(*Hint*: copy your solution to Question 0a and modify to use `.transform_bin(...)` instead of `alt.X(...)`.)

In [11]:

```
# filter, bin, and plot
alt.Chart(data).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2019)
).transform_bin(
    'Life Expectancy at Birth', # name to give binned variable
    field = 'Life Expectancy', # variable to bin
    bin = alt.Bin(step = 3) # binning parameters
).mark_bar(size = 10).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'count()'
)
```

Out[11]:



In [12]:

```
2+4+11+18+10+15+23+31+12+25+1
```

Out[12]:

```
152
```

Aggregate transform

Now, the counting of observations in each bin is *also* an under-the-hood operation in constructing the histogram. You already saw how this was done 'manually' in the example above before introducing the bin transform.

Grouped counting is a form of *aggregation*: it produces output that has fewer values than the input by combining multiple values (in this case rows) into one value (in this case a count of the number of rows).

This operation can also be made explicit using `.transform_aggregate()`. This makes use of Altair's *aggregation shorthands* for common aggregation functions; see the documentation on Altair encodings for a full list of shorthands.

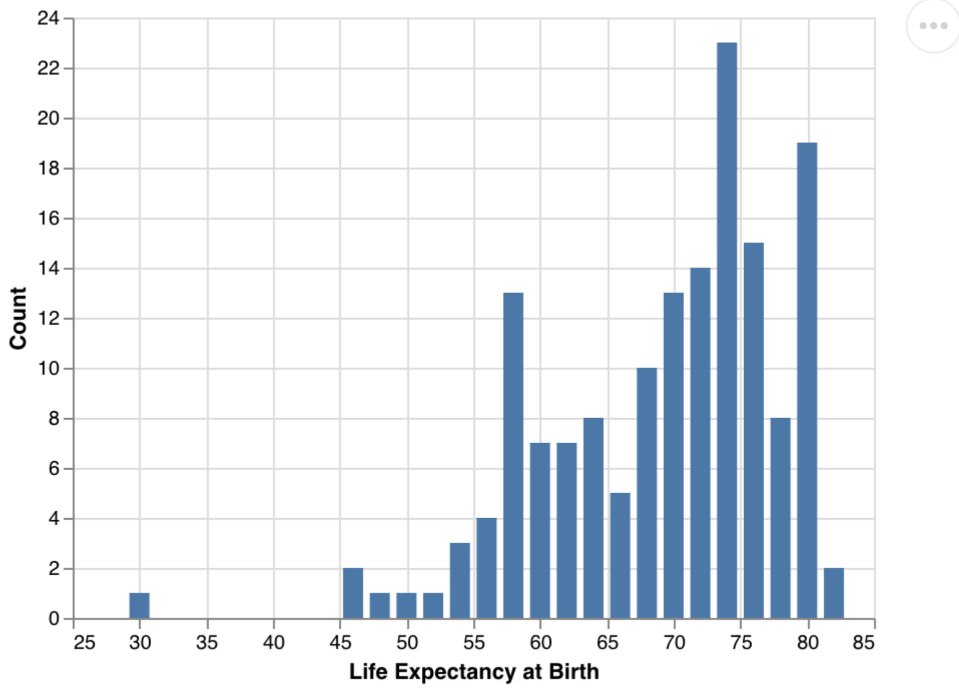
Here is how `.transform_aggregate()` would be used to perform the counting:

In [13]:

```
# filter, bin, count, and plot
alt.Chart(
    data
).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
).transform_bin(
```

```
    'Life Expectancy at Birth',
    field = 'Life Expectancy',
    bin = alt.Bin(step = 2)
).transform_aggregate(
    Count = 'count()', # altair shorthand operation -- see docs for
full list
    groupby = ['Life Expectancy at Birth'] # grouping variable(s)
).mark_bar(size = 10).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Count:Q'
)
```

Out[13]:



In [14]:

```
1+2+3+3+4+13+14+8+5+23+14+23+15+8+19+2
```

Out[14]:

```
157
```

Calculate transform

One peculiarity of Altair's histograms is that they are displayed on the *count scale* rather than the *density scale*, and there is no simple option to change this.

The **count scale** means that the y-axis shows ***counts of observations in each bin***.

By contrast, on the **density scale**, the y-axis would show ***proportions of total bar area*** (so that the area of plotted bars sums to 1).

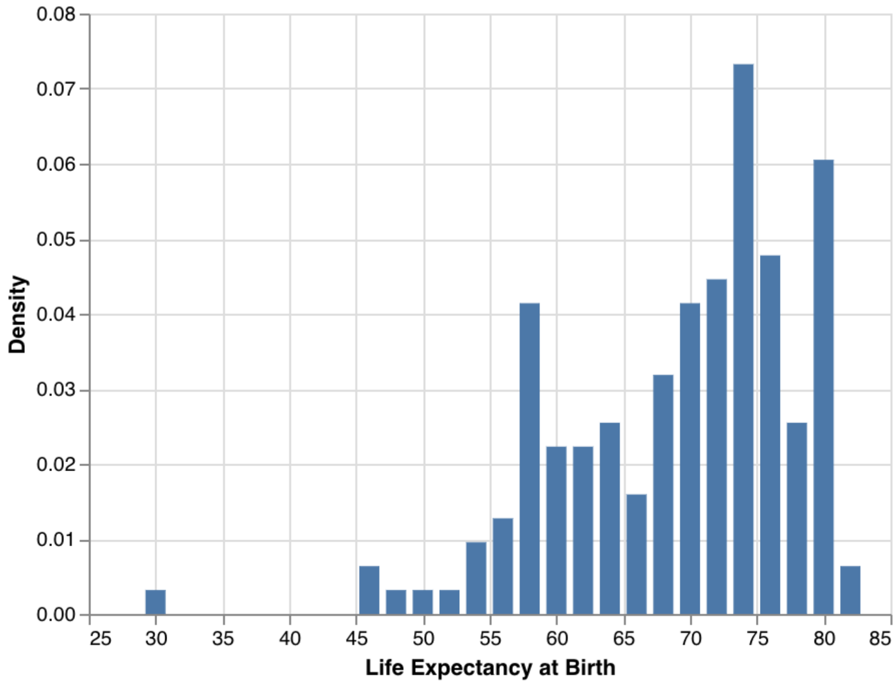
It might seem like a silly distinction -- after all, the two scales differ simply by a proportionality constant (the sample size times the bin width) -- but as you will see shortly, the density scale is more useful for statistical thinking about the distribution of values.

The scale conversion can be done using `.transform_calculate()`, which computes derived variables using arithmetic operations. In this case, one only needs to divide the count by the total number of observations.

In [15]:

```
# filter, bin, count, convert scale, and plot
alt.Chart(
    data
).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
).transform_bin(
    'Life Expectancy at Birth',
    field = 'Life Expectancy',
    bin = alt.Bin(step = 2)
).transform_aggregate(
    Count = 'count()',
    groupby = ['Life Expectancy at Birth']
).transform_calculate(
    Density = 'datum.Count/(2*157)' # divide counts by sample size x binwidth
).mark_bar(size = 10).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Density:Q'
)
```

Out[15]:



Question 0c. Density scale histogram

Follow the example above and convert your histogram from Question 0b (with the year 2019, the step size of 3, and the usage of `.transform_bin(...)`) to the density scale. First calculate the count explicitly using `.transform_aggregate(...)` and then convert to a proportion using `.transform_calculate(...)` .

(*Note*: you will need to find the sample size separately and hard-code this into the calculate step.)

Question 0ci. Density scale histogram

First, calculate the sample size and store the result in `sample_size` . Store the step size as `bin_width` .

In [16]:
```
# find sample size
sample_size = 152
bin_width = 3
```

In [17]:
```
grader.check("q0_ci")
```

Out[17]:

q0_ci passed! 🍀

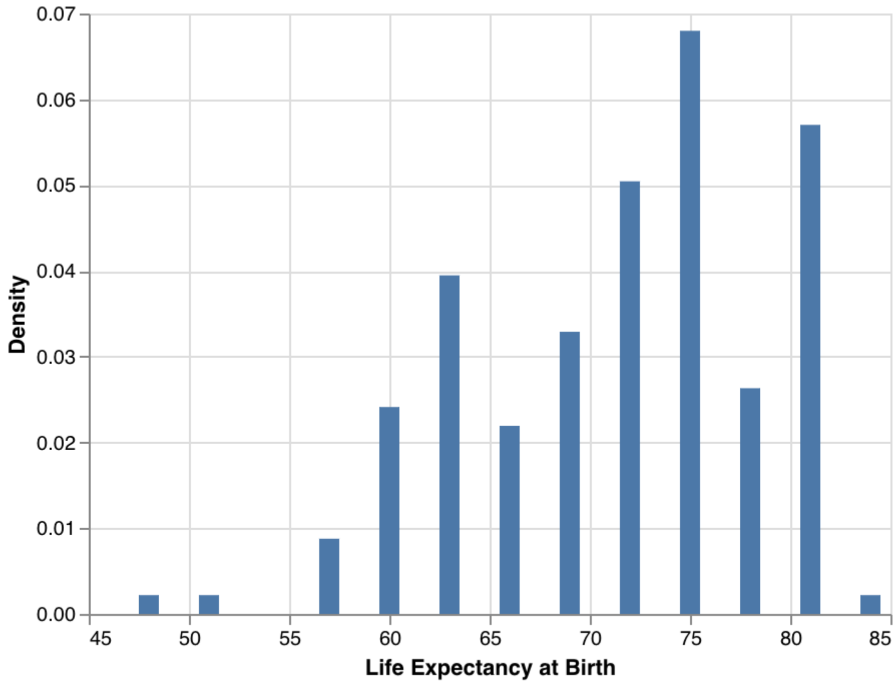
Question 0cii. Density scale histogram

Convert your histogram from Question 0b (with the year 2019, the step size of 3, and the usage of `.transform_bin(...)`) to the density scale. First calculate the count explicitly using `.transform_aggregate(...)` and then convert to a proportion using `.transform_calculate(...)`. Multiply `sample_size` with `bin_width` calculated in Question 0ci and hardcode it into your implementation.

In [18]:

```
# construct histogram
alt.Chart(
    data
).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2019)
).transform_bin(
    'Life Expectancy at Birth',
    field = 'Life Expectancy',
    bin = alt.Bin(step = 3)
).transform_aggregate(
    Count = 'count()',
    groupby = ['Life Expectancy at Birth']
).transform_calculate(
    Density = 'datum.Count/(3*152)' # divide counts by sample size x
binwidth
).mark_bar(size = 10).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Density:Q'
)
```

Out[18]:



1. Density estimation

Now that you have a sense of how transforms work, we can explore transforms that perform more sophisticated operations. We're going to focus on a technique known as *kernel density estimation*.

Histograms show the distribution of values in the sample. Let's call the density-scale histogram the *empirical density*. A **kernel density estimate** is simply ***a smoothing of the empirical density***. (It's called an 'estimate' because it's often construed as an approximation of the distribution of population values that the sample came from.)

Often the point of visualizing the distribution of a variable is to discern the shape, spread, center, and tails of the distribution to answer certain questions:

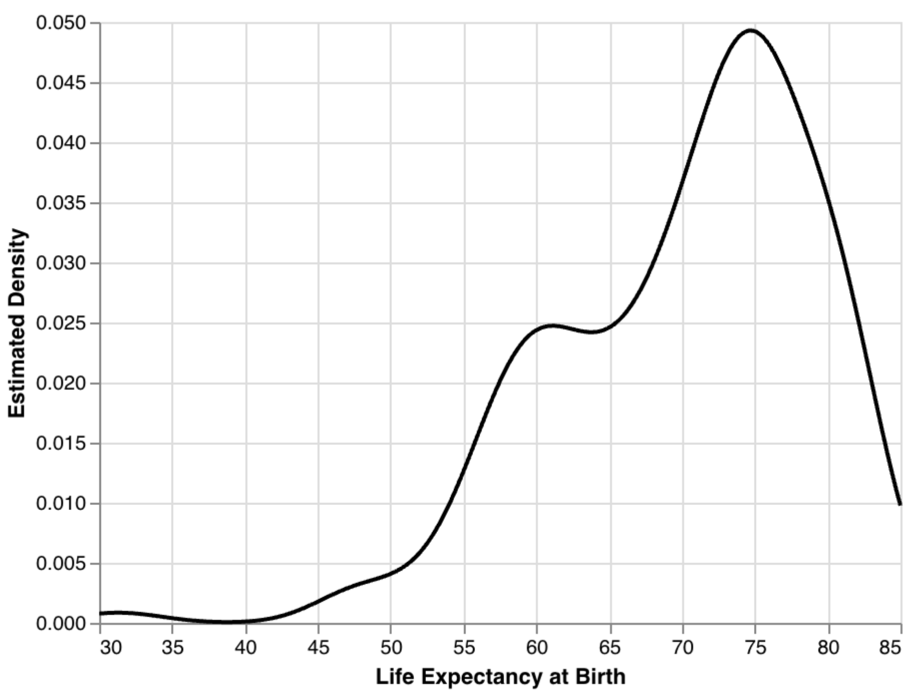
- what's a typical value?
- are there multiple typical values (multi-modal)?
- are there outliers?
- is the distribution skewed?

Density estimates are often easier to work with in exploratory analysis because it is visually easier to distinguish the shape of a smooth curve than the shape of a bunch of bars (unless you're really far away!).

'Kernel density estimate' sounds fancy, but it's surprisingly easy to plot using `.transform_density()`. The cell below generates a density estimate of life expectancies across the globe in 2010. Notice the commented lines explaining the syntax.

In [19]:
```
# plot kernel density estimate of life expectancies in 2010
alt.Chart(
    data
).transform_filter(
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
).transform_density(
    density = 'Life Expectancy', # variable to smooth
    as_ = ['Life Expectancy at Birth', 'Estimated Density'], # names of outputs
    bandwidth = 3, # how smooth?
    extent = [30, 85], # domain on which the smooth is defined
    steps = 1000 # for plotting: number of points to generate for plotting line
).mark_line(color = 'black').encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated Density:Q'
)
```

Out[19]:



This estimate can be layered onto the empirical density to get a better sense of the relationship between the two. The cell below accomplishes this. Notice that the plot elements are constructed as separate *layers*.

In [20]:
```
# base plot
base = alt.Chart(data).transform_filter(
```

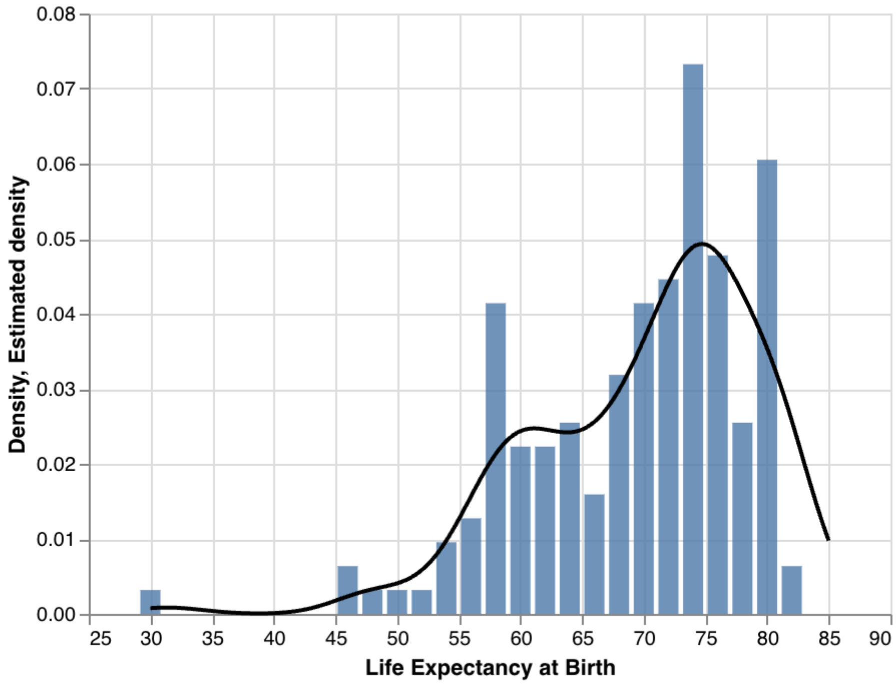
```
    alt.FieldEqualPredicate(field = 'Year',
                            equal = 2010)
)

# empirical density
hist = base.transform_bin(
    as_ = 'Life Expectancy at Birth',
    field = 'Life Expectancy',
    bin = alt.Bin(step = 2)
).transform_aggregate(
    Count = 'count()',
    groupby = ['Life Expectancy at Birth']
).transform_calculate(
    Density = 'datum.Count/(2*157)'
).mark_bar(size = 10, opacity = 0.8).encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Density:Q'
)

# kernel density estimate
smooth = base.transform_density(
    density = 'Life Expectancy',
    as_ = ['Life Expectancy at Birth', 'Estimated density'],
    bandwidth = 3,
    extent = [30, 85],
    steps = 1000
).mark_line(color = 'black').encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated density:Q'
)

# layer
hist + smooth
```

Out[20]:



What if you want a different amount of smoothing? That's what the `extent` parameter is for. The smoothing is *local*, in the following sense: at any given point, the kernel density estimate averages bar heights in a neighborhood of nearby bars proportional to how far the bars are from the point in question.

The `extent` parameter specifies the size of the smoothing neighborhood in standard deviations. For instance, above `extent = 3`, which means that the empirical density is smoothed 3SD in either direction to produce the kernel density estimate. This is also known as the *bandwidth*.

- If the bandwidth is increased, averaging is more global, so the density estimate will get smoother.
- If the bandwidth is decreased, averaging is more local, so the density estimate will get wiggly.

There are some methods out there for automating bandwidth choice, but often it is done by hand. Arguably this is preferable, as it allows the analyst to see a few possibilities and decide what best captures the shape of the distribution.

Question 1a. Selecting a bandwidth

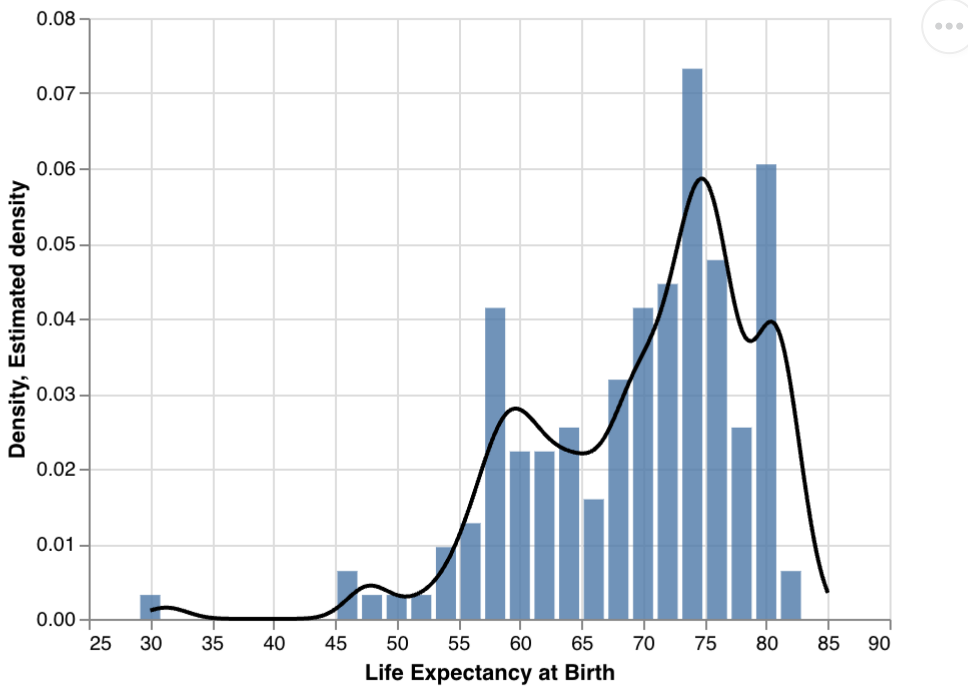
Modify the ploting codes by *decreasing* the bandwidth parameter. Try several values, and then choose one that you feel captures the shape of the distribution well without getting too wiggly.

In [21]:

```
# # change bandwidth
hist + base.transform_density(
```

```
    density = 'Life Expectancy',
    as_ = ['Life Expectancy at Birth', 'Estimated density'],
    bandwidth = 1.7, # play here
    extent = [30, 85],
    steps = 1000
).mark_line(color = 'black').encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated density:Q'
)
```

Out[21]:



Comparing distributions

The visual advantage of a kernel density estimate for discerning shape is even more apparent when comparing distributions.

As you will see as the course progresses, a major task in exploratory analysis is understanding how the distribution of a variable of interest changes depending on other variables -- for example, you have already seen in the last lab that life expectancy seems to change over time. We can explore this phenomenon from a different angle by comparing distributions in different years.

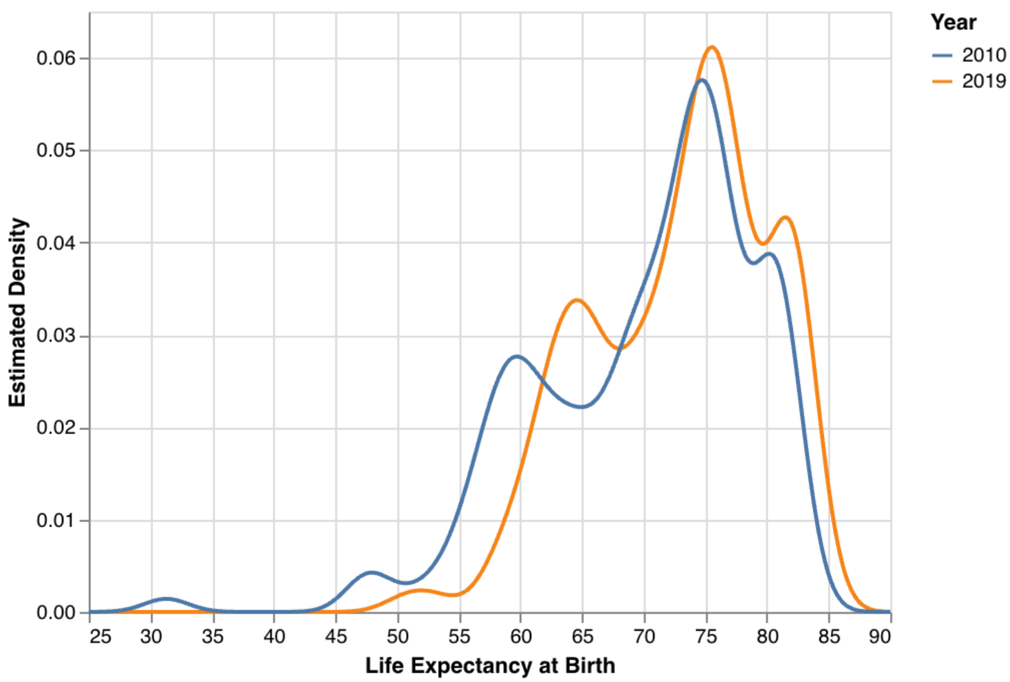
Multiple density estimates can be displayed on the same plot by passing a grouping variable (or set of variables) to `.transform_density(...)`. For example, the cell below computes density estimates of life expectancies for each of two years.

In [22]:

```
alt.Chart(data).transform_filter(
    alt.FieldOneOfPredicate(field = 'Year',
```

```
                              oneOf = [2010, 2019])
).transform_density(
    density = 'Life Expectancy',
    groupby = ['Year'],
    as_ = ['Life Expectancy at Birth', 'Estimated Density'],
    bandwidth = 1.8,
    extent = [25, 90],
    steps = 1000
).mark_line().encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated Density:Q',
    color = 'Year:N'
)
```

Out[22]:



Often the area beneath each density estimate is filled in. This can be done by simply appending a `.mark_area()` call at the end of the plot.

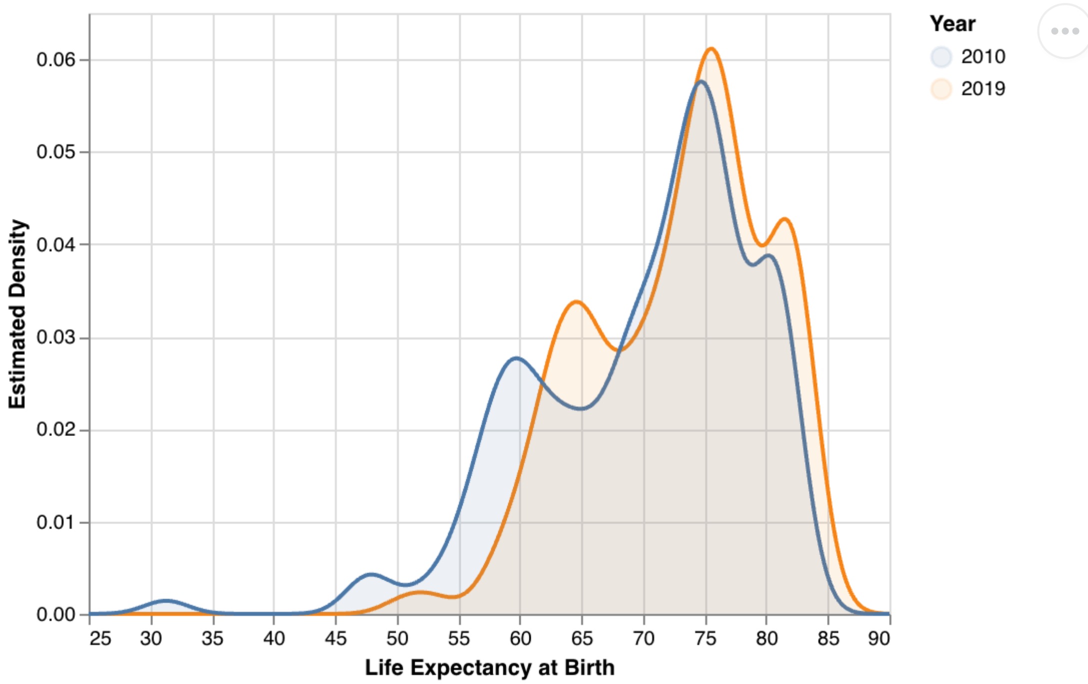
In [23]:

```
p = alt.Chart(data).transform_filter(
    alt.FieldOneOfPredicate(field = 'Year',
                              oneOf = [2010, 2019])
).transform_density(
    density = 'Life Expectancy',
    groupby = ['Year'],
    as_ = ['Life Expectancy at Birth', 'Estimated Density'],
    bandwidth = 1.8,
```

```
    extent = [25, 90],
    steps = 1000
).mark_line().encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated Density:Q',
    color = 'Year:N'
)

p + p.mark_area(opacity = 0.1)
```

Out[23]:



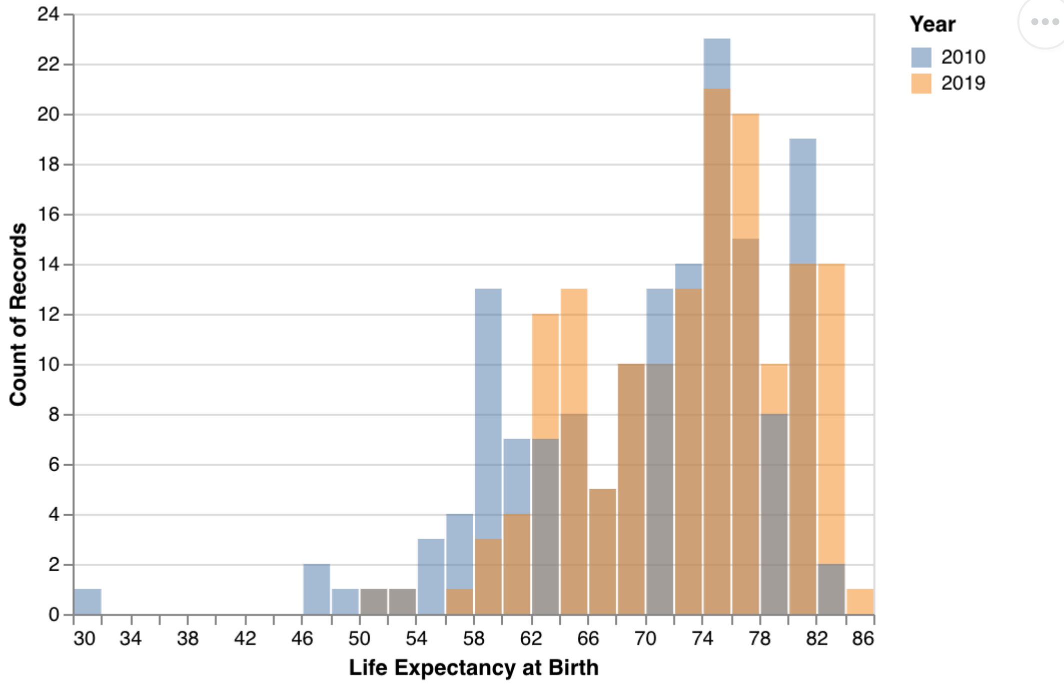
Notice that this makes it much easier to compare the distributions between years -- you can see a pronounced rightward shift of the smooth for 2019 compared with 2010.

We could make the same comparison based on the histograms, but the shift is a lot harder to make out.

In [24]:

```
alt.Chart(data).transform_filter(
    alt.FieldOneOfPredicate(field = 'Year',
                            oneOf = [2010, 2019])
).mark_bar(opacity = 0.5).encode(
    x = alt.X('Life Expectancy', bin = alt.Bin(maxbins = 30), title
= 'Life Expectancy at Birth'),
    y = alt.Y('count()', stack = None),
    color = 'Year:N'
)
```

Out[24]:



Question 1b. Multiple density estimates

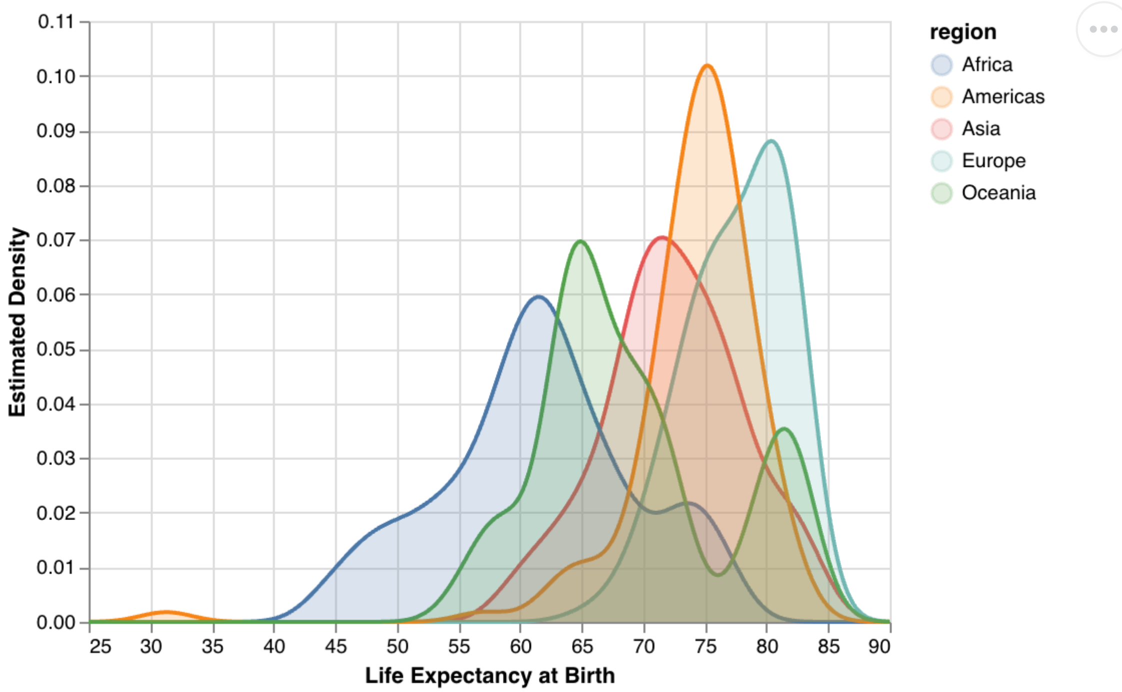
Follow the appropriate example above to construct a plot showing separate density estimates of life expectancy for each region in the 2010. You can choose whether you prefer to fill in the area beneath the smooth curves, or not. Be sure to play with the bandwidth parameter and choose a value that seems sensible to you.

In [25]:

```
p = alt.Chart(data).transform_filter(
    alt.FieldOneOfPredicate(field = 'region',
                            oneOf = ['Africa', 'Asia', 'Europe',
'Americas', 'Oceania'])
).transform_density(
    density = 'Life Expectancy',
    groupby = ['region'],
    as_ = ['Life Expectancy at Birth', 'Estimated Density'],
    bandwidth = 2.1,
    extent = [25, 90],
    steps = 1000
).mark_line().encode(
    x = 'Life Expectancy at Birth:Q',
    y = 'Estimated Density:Q',
    color = 'region:N'
)

p + p.mark_area(opacity = 0.2)
```

Out[25]:



Question 1c. Interpretation

Do the distributions of life expectancies seem to differ by region? If so, what is one difference that you notice? Answer in 1-2 sentences.

The distributions are all the same and follow a standard normal distribution.

Question 1d. Outlier

Notice that little peak way off to the left in the distribution of life expectancies in the Americas. That's an outlier.

(i) Which country is it?

Check by filtering `data` appropriately and using `.sort_values(...)` to find the lowest life expectancy in the Americas. Save the row of `data` that shows the outlying observation in `lowest_Americas`.

(*Hint*: You might want to check the documentation.)

In [26]:

```
# show row of outlier
lowest_Americas = data.sort_values(by=['Life Expectancy'])
lowest_Americas
```

Out[26]:

	Country Name	Year	Life Expectancy	Male Life Expectancy	Female Life Expectancy	GDP per capita	region	sub regio
246	Haiti	2010	31.3	28.0	35.4	1172.098543	Americas	Lat Americ and th Caribbea
98	Burundi	2000	43.8	41.8	45.9	136.463971	Africa	Sub Sahara Afric
118	Central African Republic	2000	44.3	43.2	45.5	251.206877	Africa	Sub Sahara Afric
615	Zambia	2000	44.5	43.7	45.2	345.689554	Africa	Sub Sahara Afric
346	Malawi	2000	44.7	44.0	45.6	156.385719	Africa	Sub Sahara Afric
...	...	...	...	...	...	...	...	
528	Spain	2019	83.2	80.7	85.7	29600.378250	Europe	Souther Europ
510	Singapore	2019	83.2	81.0	85.5	65233.282440	Asia	South easter As
548	Switzerland	2019	83.4	81.8	85.1	81993.727130	Europe	Wester Europ
288	Japan	2015	83.6	80.7	86.4	34524.469860	Asia	Easter As
287	Japan	2019	84.3	81.5	86.9	40246.880130	Asia	Easter As

620 rows × 9 columns

In [27]:

```
grader.check("q1_d_i")
```

Out[27]:

q1_d_i passed! 🍀

(ii) What was the life expectancy for that country in other years?

Now filter the data to examine the life expectancy in the country you identified as the outlier in all years. Save the corresponding four rows of `data` : one row for each year into `outlier_country`.

(*Hint*: filter by country.)

In [28]:
```
# show rows for all years for country of interest
outlier_country = data['Life Expectancy'].loc[data['Country Name']
== 'Haiti']
outlier_country
```

Out[28]:
```
244    64.1
245    62.6
246    31.3
247    57.0
Name: Life Expectancy, dtype: float64
```

(iii) What Happened in 2010?

Can you explain why the life expectancy was so low in that country for that particular year?

(*Hint*: if you don't remember, Google the country name and year in question.)

There was a huge earthquake.

---

2. Scatterplot smooths

In this brief section you'll see two techniques for smoothing scatterplots: LOESS, which produces a curve; and regression, which produces a linear smooth.

The next parts will modify the dataframe `data` by adding a column. We'll create a copy `data_mod1` of the original dataframe `data` to modify as to not lose track of our previous work:

In [29]:
```
data_mod1 = data.copy()
```

LOESS

Locally w**e**ighted **s**catterplot **s**moothing (LOESS) is a flexible smoothing technique for visualizing trends in scatterplots. The technical details are a little beyond us at the moment, but it's easy enought to implement.

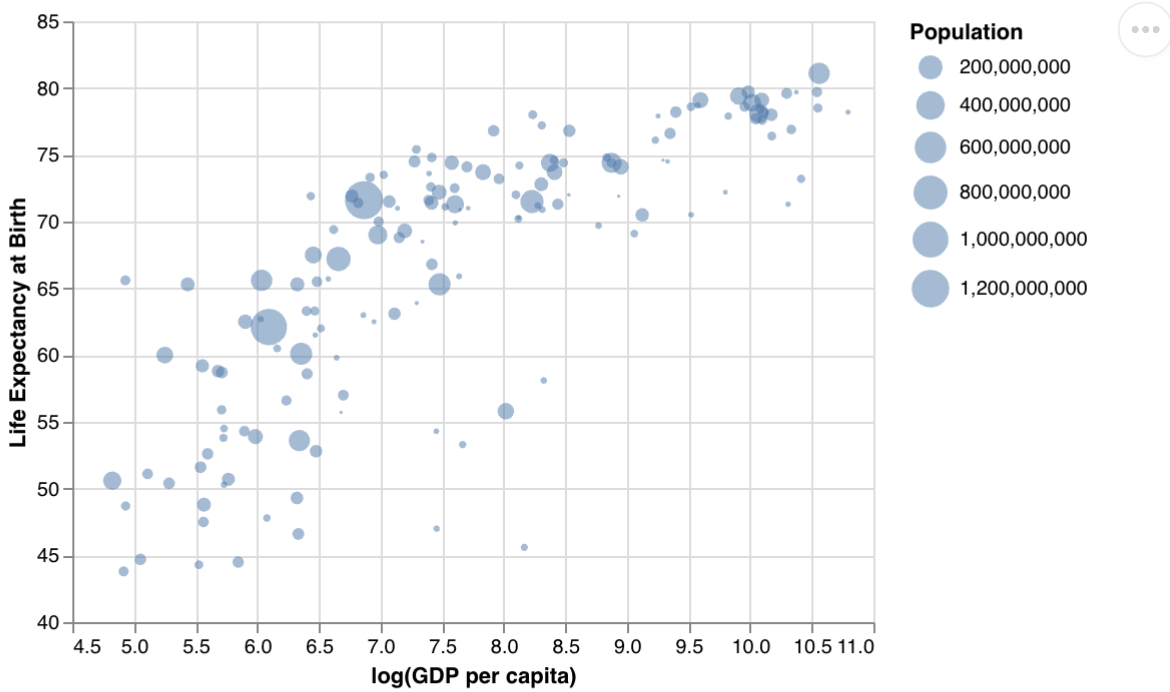
To illustrate, consider the scatterplots you made in lab 3 showing the relationship between life expectancy and GDP per capita. The plot for 2010 looked like this:

In [30]:
```
# log transform gdp explicitly
data_mod1['log(GDP per capita)'] = np.log(data_mod1['GDP per
capita'])
```

```
# scatterplot
scatter = alt.Chart(data_mod1).transform_filter(
    alt.FieldEqualPredicate(field = 'Year', equal = 2000)
).mark_circle(opacity = 0.5).encode(
    x = alt.X('log(GDP per capita)', scale = alt.Scale(zero =
False)),
    y = alt.Y('Life Expectancy', title = 'Life Expectancy at Birth',
scale = alt.Scale(zero = False)),
    size = alt.Size('Population', scale = alt.Scale(type = 'sqrt'))
)

# show
scatter
```

Out[30]:



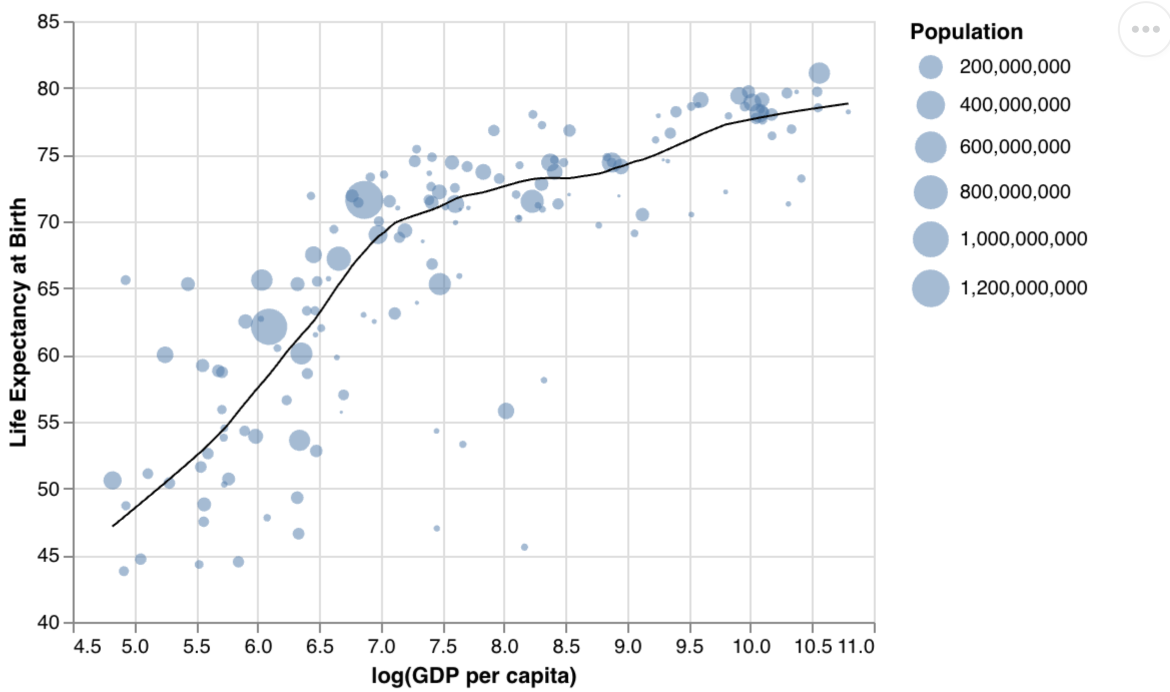
To add a LOESS curve, simply append `.transform_loess()` to the base plot:

In [31]:

```
# compute smooth
smooth = scatter.transform_loess(
    on = 'log(GDP per capita)', # x variable
    loess = 'Life Expectancy', # y variable
    bandwidth = 0.25 # how smooth?
).mark_line(color = 'black')

# add as a layer to the scatterplot
scatter + smooth
```

Out[31]:



Just as with kernel density estimates, LOESS curves have a bandwidth parameter that controls how smooth or wiggly the curve is. In Altair, the LOESS bandwidth is a unitless parameter between 0 and 1.

Question 2a. LOESS bandwidth selection

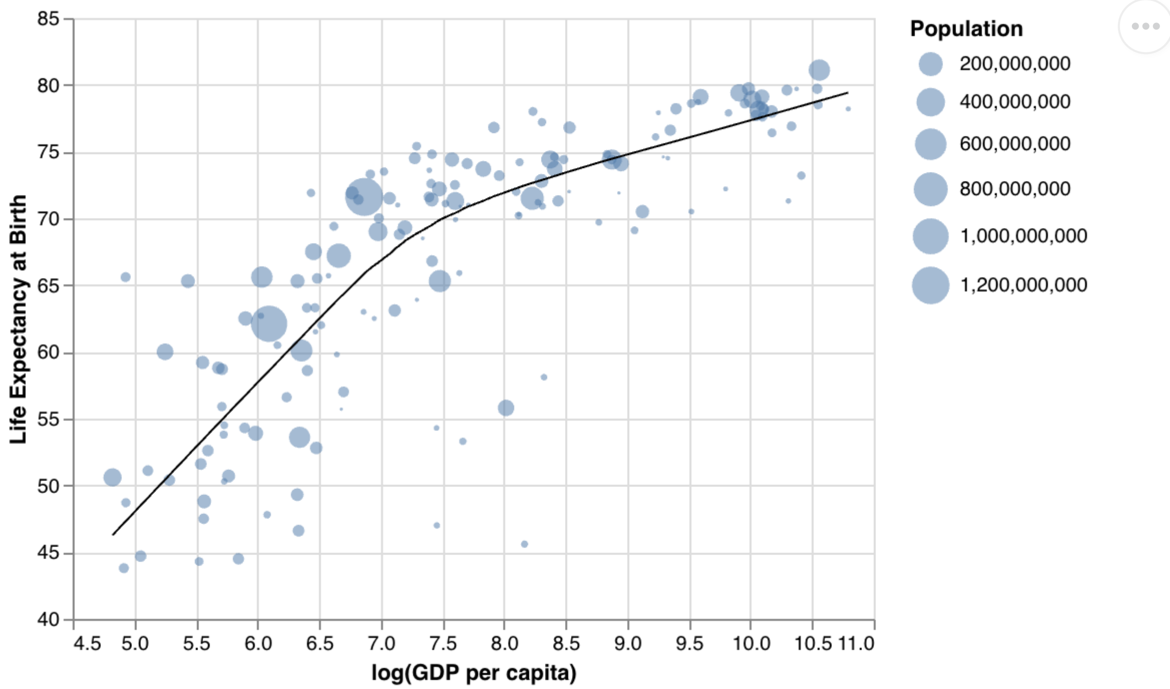
Tinker with the bandwidth parameter to see its effect in the cell below. Then choose a value that produces a smoothing you find appropriate for indicating the general trend shown in the scatter.

In [32]:

```
# compute smooth
smooth = scatter.transform_loess(
    on = 'log(GDP per capita)', # x variable
    loess = 'Life Expectancy', # y variable
    bandwidth = 0.7 # how smooth?
).mark_line(color = 'black')

# add as a layer to the scatterplot
scatter + smooth
```

Out[32]:



LOESS curves can also be computed groupwise. For instance, to display separate curves for each region, one need only pass a `groupby = ...` argument to `.transform_loess()`:

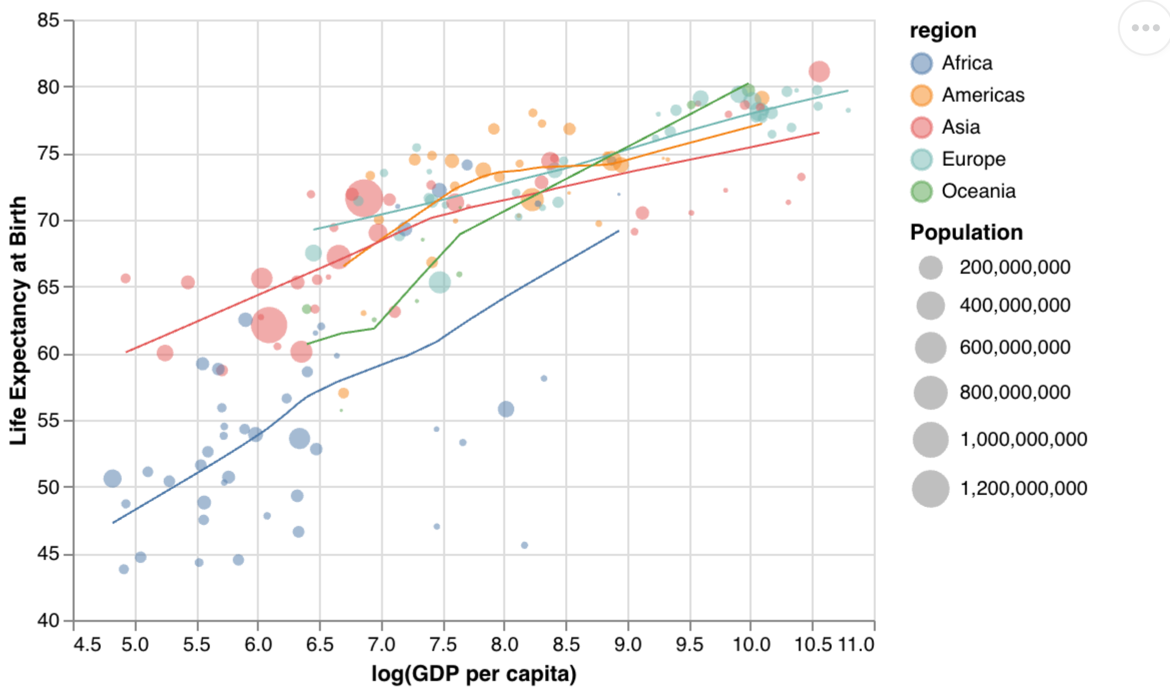
In [33]:

```
# scatterplot
scatter = alt.Chart(data_mod1).transform_filter(
    alt.FieldEqualPredicate(field = 'Year', equal = 2000)
).mark_circle(opacity = 0.5).encode(
    x = alt.X('log(GDP per capita)', scale = alt.Scale(zero = False)),
    y = alt.Y('Life Expectancy', title = 'Life Expectancy at Birth', scale = alt.Scale(zero = False)),
    size = alt.Size('Population', scale = alt.Scale(type = 'sqrt')),
    color = 'region'
)

# compute smooth
smooth = scatter.transform_loess(
    groupby = ['region'], # add groupby
    on = 'log(GDP per capita)',
    loess = 'Life Expectancy',
    bandwidth = 0.8
).mark_line(color = 'black')
```

```
# add as a layer to the scatterplot
scatter + smooth
```

Out[33]:



The curves are a little jagged because there aren't very many countries in each region.

In [34]:

```
data_mod1[data_mod1.Year == 2000].groupby('region').count().iloc[:,
[0]]
```

Out[34]:

	Country Name
region	
Africa	45
Americas	27
Asia	38
Europe	35
Oceania	9

Regression

You will be learning more about linear regression later in the course, but we can introduce regression lines now as a visualization technique. As with LOESS, you don't need to concern yourself with the mathematical details (yet!). From this perspective, regression is a form of *linear* smoothing -- a regression smooth is a straight line. By contrast, LOESS smooths have *curvature* -- they are not straight lines.

In the example above, the LOESS curves don't have much curvature. So it may be a cleaner choice visually to show linear smooths. This can be done using

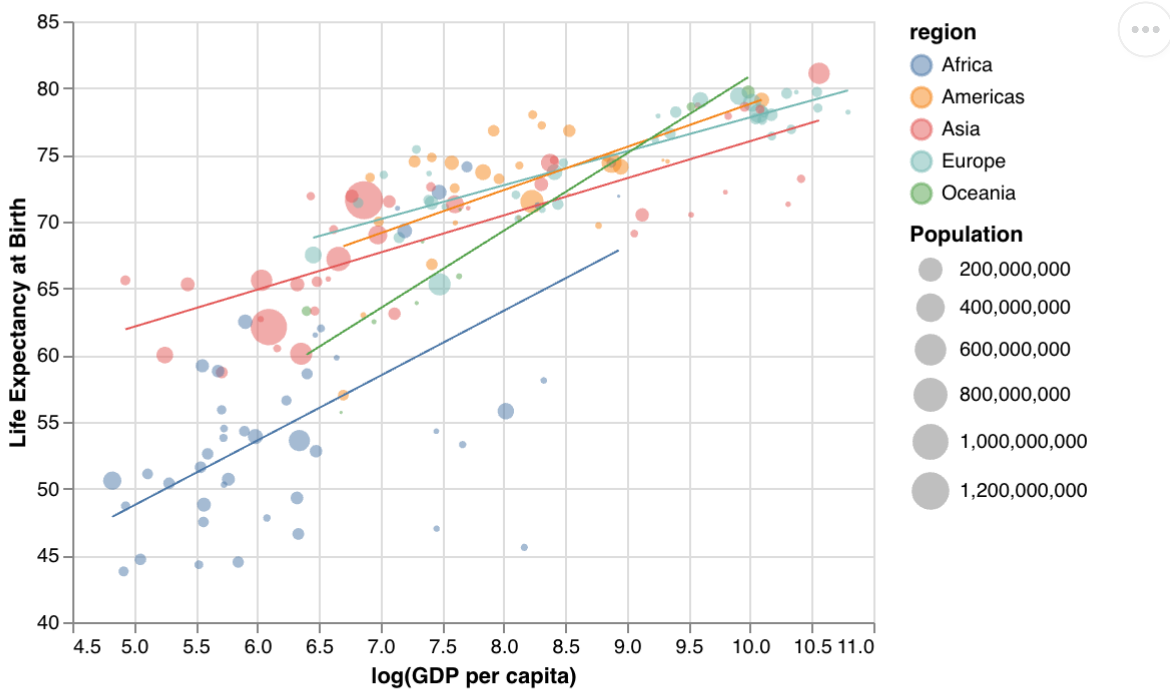
`.transform_regression(...)` with a similar argument structure.

In [35]:
```
# scatterplot
scatter = alt.Chart(data_mod1).transform_filter(
    alt.FieldEqualPredicate(field = 'Year', equal = 2000)
).mark_circle(opacity = 0.5).encode(
    x = alt.X('log(GDP per capita)', scale = alt.Scale(zero =
False)),
    y = alt.Y('Life Expectancy', title = 'Life Expectancy at Birth',
scale = alt.Scale(zero = False)),
    size = alt.Size('Population', scale = alt.Scale(type = 'sqrt')),
    color = 'region'
)

# compute smooth
smooth = scatter.transform_regression(
    groupby = ['region'],
    on = 'log(GDP per capita)',
    regression = 'Life Expectancy'
).mark_line(color = 'black')

# add as a layer to the scatterplot
scatter + smooth
```

Out[35]:



Question 2b. Simple regression line

Based on the example immediately above, construct a scatterplot of life expectancy against log GDP per capita in 2010 with points sized according to population (and no distinction between regions). Layer a single linear smooth on the scatterplot using `.transform_regression(...)`.

(*Hint*: remove the color aesthetic and grouping from the previous plot.)

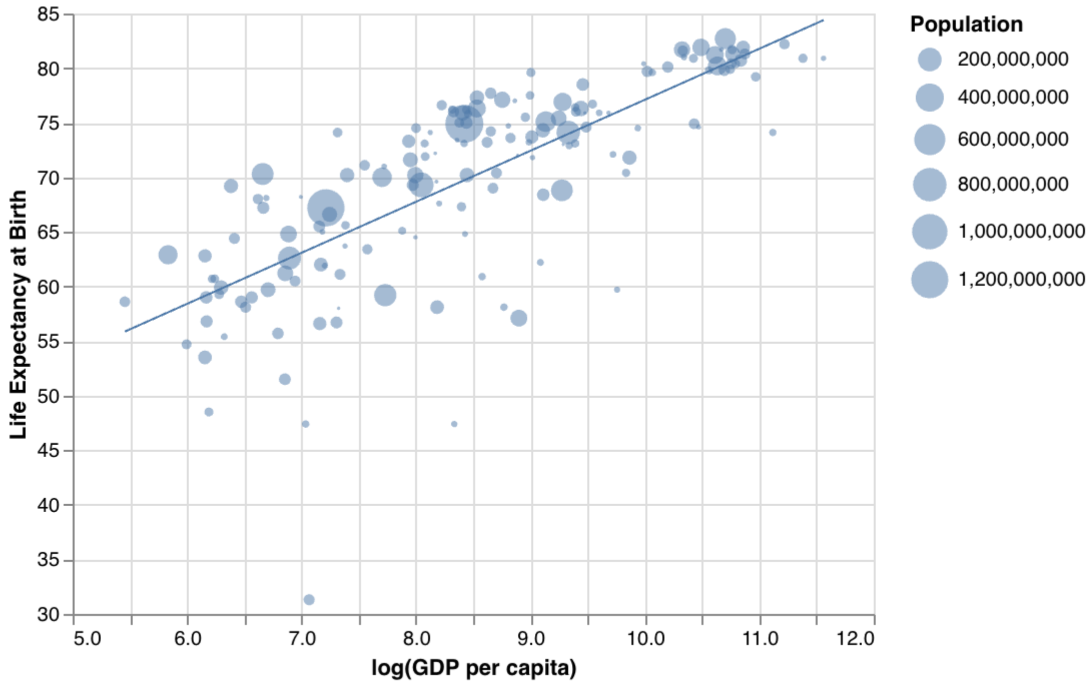
In [36]:

```
# scatterplot
scatter = alt.Chart(data_mod1).transform_filter(
    alt.FieldEqualPredicate(field = 'Year', equal = 2010)
).mark_circle(opacity = 0.5).encode(
    x = alt.X('log(GDP per capita)', scale = alt.Scale(zero = 
False)),
    y = alt.Y('Life Expectancy', title = 'Life Expectancy at Birth', 
scale = alt.Scale(zero = False)),
    size = alt.Size('Population', scale = alt.Scale(type = 'sqrt'))
)

# compute smooth
smooth = scatter.transform_regression(
    on = 'log(GDP per capita)',
    regression = 'Life Expectancy'
).mark_line()

# add as a layer to the scatterplot
scatter + smooth
```

Out[36]:



Submission Checklist

1. Save file to confirm all changes are on disk
2. Run *Kernel > Restart & Run All* to execute all code from top to bottom
3. Save file again to write any new output to disk
4. Select *File > Download as > HTML*.
5. Open in Google Chrome and print to PDF on A3 paper in portrait orientation.
6. Submit to Gradescope

To double-check your work, the cell below will rerun all of the autograder tests.

In [37]:
```
grader.check_all()
```

Out[37]:
```
q0_ci results: All test cases passed!

q1_d_i results: All test cases passed!
```