CSSS 569 · Visualizing Data and Models

GRAPHICAL PROGRAMMING IN R

Christopher Adolph

Department of Political Science

and

Center for Statistics and the Social Sciences

University of Washington, Seattle

Overview of R graphics facilities

Overview of R programming basics

Graphics devices and graphics systems

Vector vs. raster graphics devices

Imperative vs. declarative graphics systems

Overview of the ggplot2 system

Making and polishing a scatterplot with ggplot2

A few key concepts from the base graphics system

Essentials of the grid graphics system

A grid graphics example

Graphics devices, graphics systems

To understand how to make graphs in R, the first step is to distinguish graphics devices, low-level graphics systems, and high-level graphics systems:

A **graphics device** is a canvas (window or file) on which a graph is drawn and saved for export to other applications

Ex: a PDF file; a PNG file; the RStudio graphics window

A **low-level graphics system** provides a coherent set of primitive tools for drawing graphs on devices

Complete list: `base`, `grid`

A **high-level graphics system** works on top of *one specific* low-level system to provide efficient and powerful user-end tools

Ex: `ggplot2`, `tile`, `lattice`

Some example combinations of graphics devices and systems

Graphics device	Low-level system	High-level system
PDF	base	
PDF	grid	
PDF	grid	ggplot2
RStudio Graphics Device	grid	ggplot2
PNG	grid	ggplot2
PDF	grid	tile
...		

All R graphs must have a graphics device and a low-level graphics system

Most also use a high-level graphics system, which tends to parse everything for the device and low-level system

All modern R graphics are made using sophisticated high-level systems built on the powerful low-level system `grid`

Original low-level `base` system a legacy to avoid (but some basic concepts useful)

Some example combinations of graphics devices and systems

Graphics device	Low-level system	High-level system
PDF	base	
PDF	grid	
PDF	grid	ggplot2
RStudio Graphics Device	grid	ggplot2
PNG	grid	ggplot2
PDF	grid	tile
...		

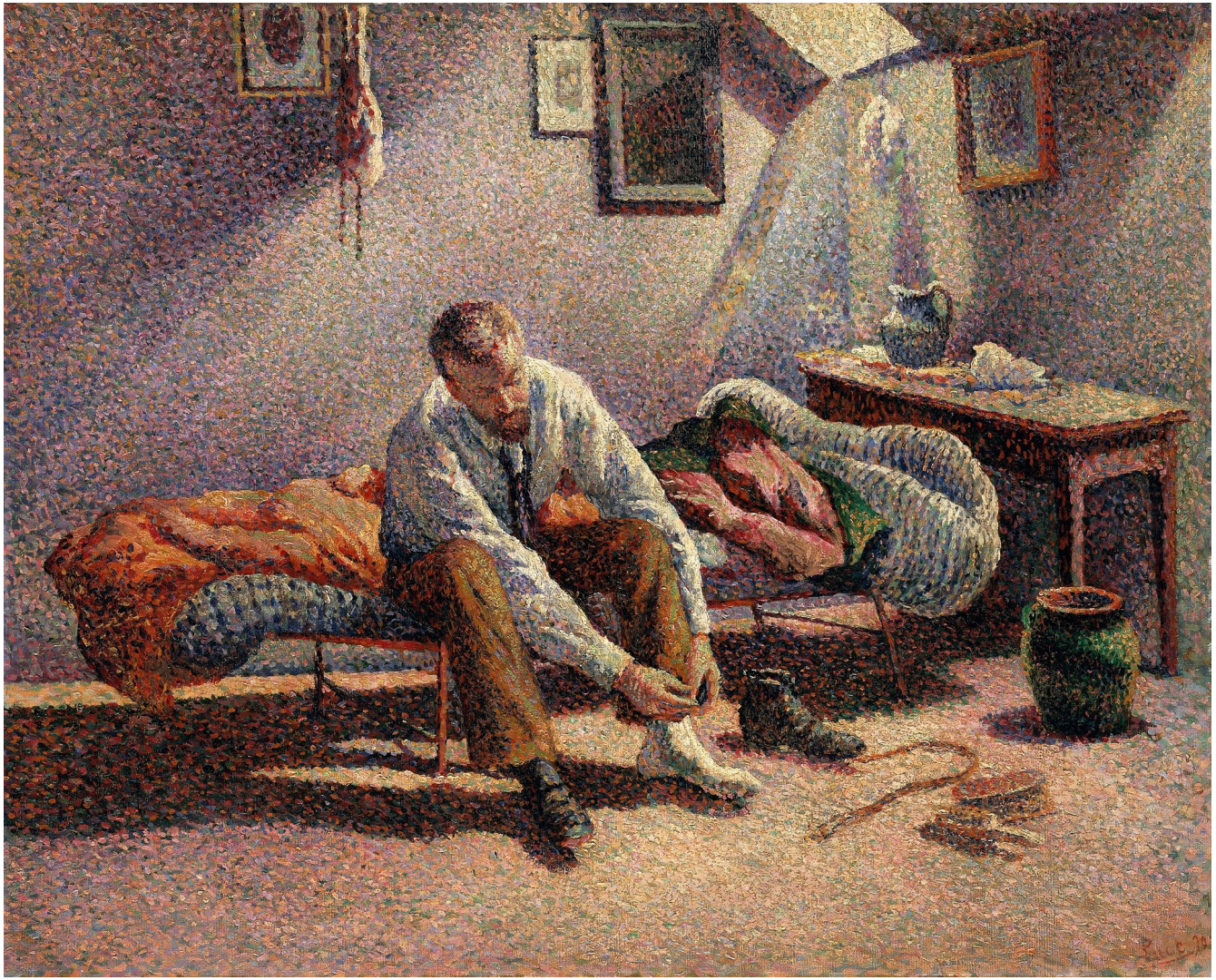
Any R graphics system can plot results to *any* graphics device

Graphics devices are blank canvases – files or windows – on which graphs are drawn

If you don't think you are using a graphics device, one has been chosen for you

Exporting from one graphical device to another is fraught with error,
so set the system you want from the start – don't export from the screen to a file!

Raster vs. Vector Graphics



Maximilien Luce · *Morning, Interior* · 1890 · Metropolitan Museum of Art

Raster graphics = pointalism

Color each pixel on an *N* by *M* grid

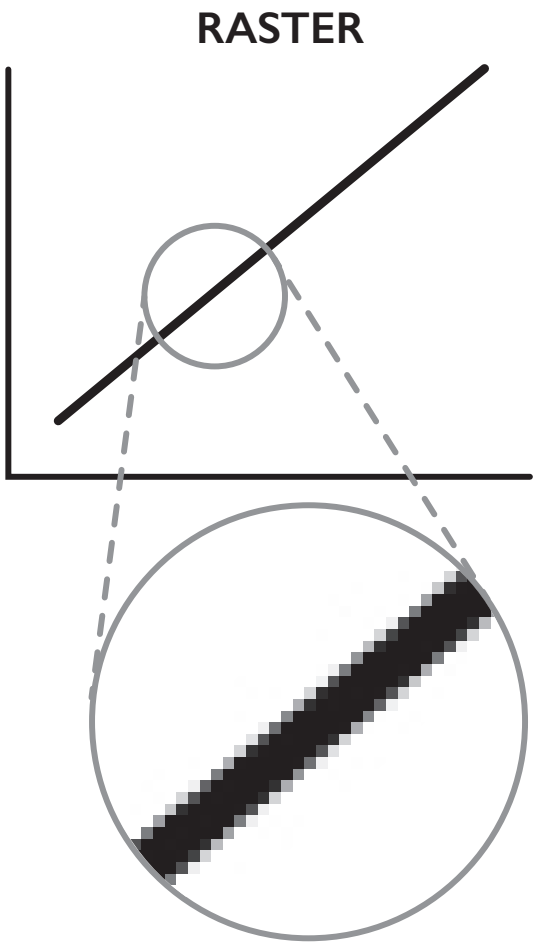
Pixel = Point = Raster

Good for photos and paintings

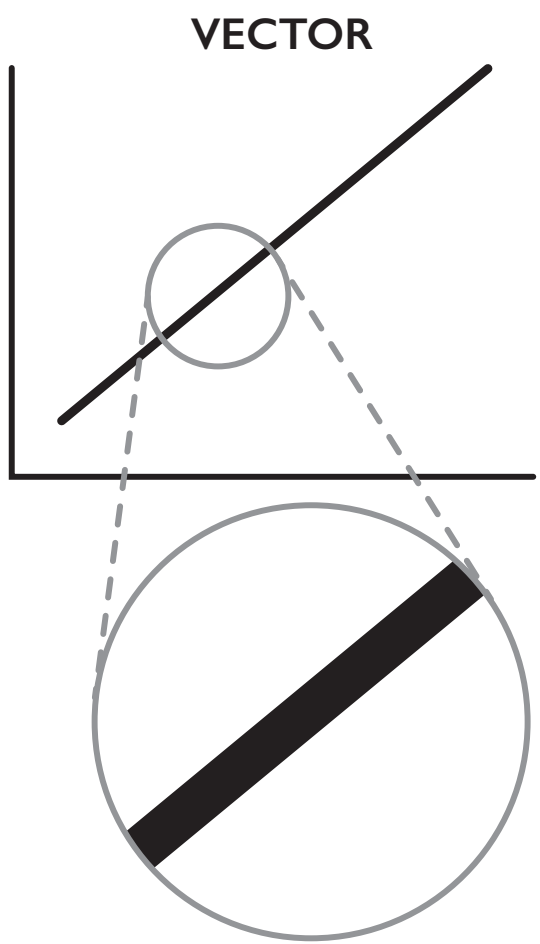
Bad for graphics

Accidental default of the web

Raster vs. Vector Graphics



Draw a black pixel here, and a gray pixel here, and another gray pixel here...



Draw $y = ax + b$ exactly here

Easily manipulable/modifiable groupings of objects

Much smaller file sizes

Easy to scale objects larger or smaller/arbitrary precision

Inefficient for photos/paintings

Poorly supported on the web

Some example combinations of graphics devices and systems

	Lossy	Lossless
Raster	.jpeg, .gif	.png, .tiff
Vector	–	.pdf, .ps, .eps, .ai

Lossy means during file compression, some data is (intentionally) lost

Avoid lossy formats whenever possible (e.g., use PNG instead of JPEG)

Don't copy and paste an image, e.g., to place it in Microsoft Word
This may rasterize your vector graphic to a lossy format!

This is where "blurry" graphs in journal articles come from

Solution: It's best to make final scientific graphics directly as a PDF

High-level graphic systems: Differing philosophies

Decided on a graphics device?
Then it's time to start building a graphic with using the coherent tools of a single graphics system

R graphics systems implement different philosophies of computer programming

Imperative programming

Step-by-step instructions to control the exact construction of output

Hands on and more work: craft your solution the way you want it

Uses procedures and object-orientation for convenience

Ex: `base`, `grid`, `tile`

High-level graphic systems: Differing philosophies

Decided on a graphics device?
Then it's time to start building a graphic with using the coherent tools of a single graphics system

R graphics systems implement different philosophies of computer programming

Imperative programming

Step-by-step instructions to control the exact construction of output

Hands on and more work: craft your solution the way you want it

Uses procedures and object-orientation for convenience

Ex: `base`, `grid`, `tile`

Declarative programming

Define your problem; allow software to apply a standard solution

Defaults may be pretty good; if not, you may customize with a stylesheet

Major changes can be hard without switching to procedural coding

Ex: `ggplot2`

Declarative graphics using `ggplot2`

Hadley Wickham's `ggplot2` implements Leland Wilkinson's "grammar of graphics" within the powerful `grid` graphics system

Essential idea: graphics link data to dimensions of specific aesthetic objects, which are distinguishable by their geometric structure and modifiable in scale & style

Implementation: users supply the data, request a geometry; software handles details

Result is a widely used and extended suite of standard graphics types and tools with strongly defined default styles and solutions to specific graphical problems

Declarative graphics using `ggplot2`

Hadley Wickham's `ggplot2` implements Leland Wilkinson's "grammar of graphics" within the powerful `grid` graphics system

Essential idea: graphics link data to dimensions of specific aesthetic objects, which are distinguishable by their geometric structure and modifiable in scale & style

Implementation: users supply the data, request a geometry; software handles details

Result is a widely used and extended suite of standard graphics types and tools with strongly defined default styles and solutions to specific graphical problems

Three key steps to creating a `ggplot2` graphic

1. Establish a **mapping** between data variables & plotting dimensions/elements
2. Apply the mapping to one or more standardized **aesthetic** elements
3. Draw the resulting set of graphical objects to a graphics device

Three key steps to creating a ggplot2 graphic — *Step 1*

1. Establish a **mapping** between data variables & plotting dimensions/elements

For example, to map the variables `Var1` and `Var2` from the `myData` dataframe to the *x* and *y* coordinates of a plot, we start with:

```
p <- ggplot(data = myData, mapping=aes(x = Var1, y = Var2))
```

Other options for `aes()` inputs include `color=`, `size=`, `shape=`, etc.

This step initializes a set of graphical objects saved as `p`

2. Apply the mapping to one or more standardized **aesthetic** elements

To make scatterplot relying on the mapping of `Var1` and `Var2` to the *x* and *y* coordinates of a plot, we then run:

```
p <- p + geom_point()
```

Many other plotting geometries exist and can be combined, including...

`geom_point`	scatterplots; dotplots if one dimension is categorical
`geom_pointrange`	dot-and-whisker plot
`geom_smooth`	apply regression fitting to points, with shaded CIs
`geom_bar`	barplot
`geom_histogram`	histogram
`geom_density`	density plot (smoothed histogram)
`geom_density2d`	2d kernel density plot
`geom_raster`	image plots
`geom_contour`	contour plots
`geom_rug`	show marginal distributions on axes
`geom_text`	plot text labels
`geom_line`	draw connected line segments
`geom_polygon`	draw polygons
`geom_step`	draw a stair-stepped line

Each `geom` we add to the plot with a "+" is drawn on top of the existing plot and may modify its axes, labels, legends, and layout

3. Draw the resulting set of graphical objects to a graphics device

To save as a PDF, then run:

```
ggsave(''myplot.pdf'')
```

Important options at this step include:
alternative graphics devices (use an appropriate extention, such as .png)
width and height of the final graphic file

In practice, there is a fourth step:
revising the first three steps and re-running code until the output is perfected

Advantages and disadvantages of `ggplot2`

`ggplot2` is widely used and rapidly expanding with add-on packages

Plays well with other packages like Wickham's `tidyverse`, RMarkdown, and `Shiny`

Very easy to get started and make something decent

Advantages and disadvantages of `ggplot2`

On the other hand, many of the defaults are problematic cognitively and aesthetically

Customization using themes solves some problems,
but others require original step-by-step code or awkward workarounds

That can be hard in a system that wants to implement a standardized set of solutions to a structured graphical "grammar"

Deeper concern: if `ggplot2` is all you know,
it's hard to ever start with a blank canvas and make something truly new

Declarative programming assumes a single correct solution for each problem:
Can this be fully true of a creative, aesthetic field like data visualization?

Learning ggplot2 – strengths, limits & workarounds – by example

Kieran Healy's *Data Visualization* (2018, Princeton U.P.) does a good job of surveying some of the common geom's and general tricks

Instead, let's take a deeper look at ggplot2 using a single example

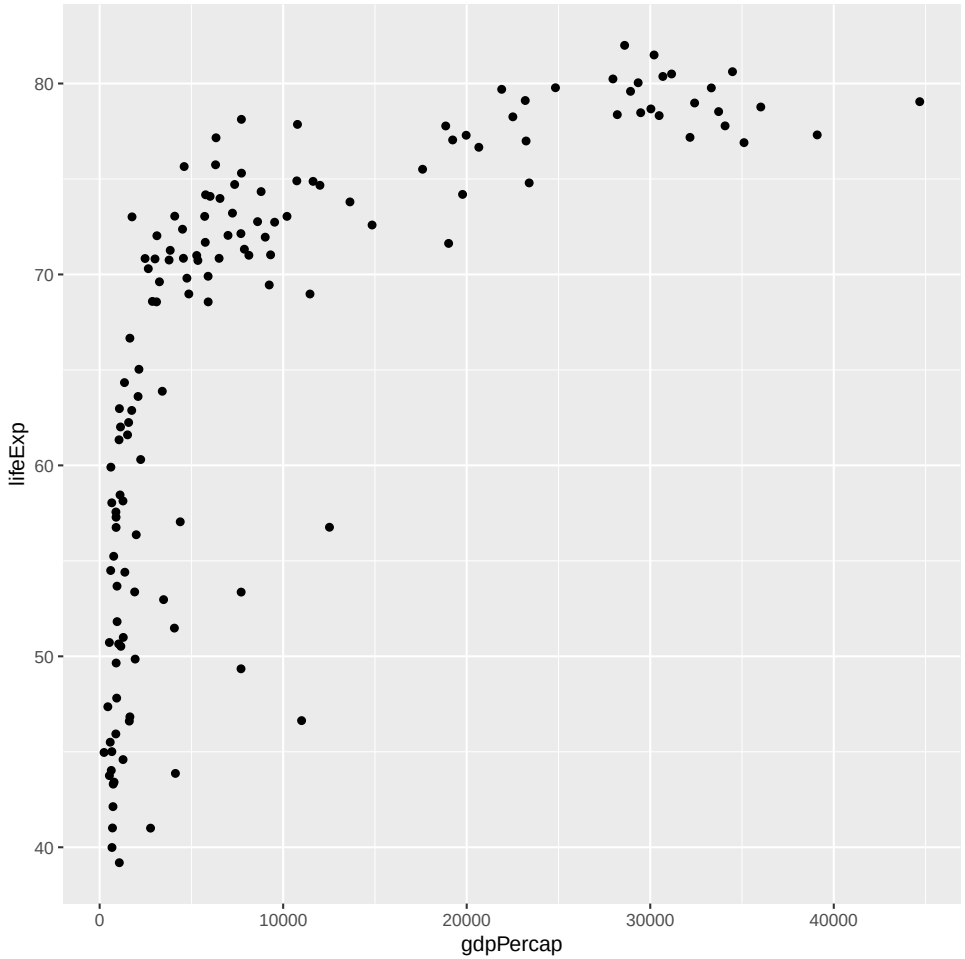
We draw on Hans Rosling's Gapminder data – gapminder.org –
which famously illustrates the relationship between life expectancy and GDP per capita

Rosling's original presentation – youtube.com/watch?v=jbkSRLYSojo –
looked at this relationship over time for all countries; we mostly focus on 2002 data

As we work through the example, keep in mind ways you could improve the graph scientifically, statistically, cognitively, and stylistically

Build a scatterplot using `ggplot2`

1.1 Default settings



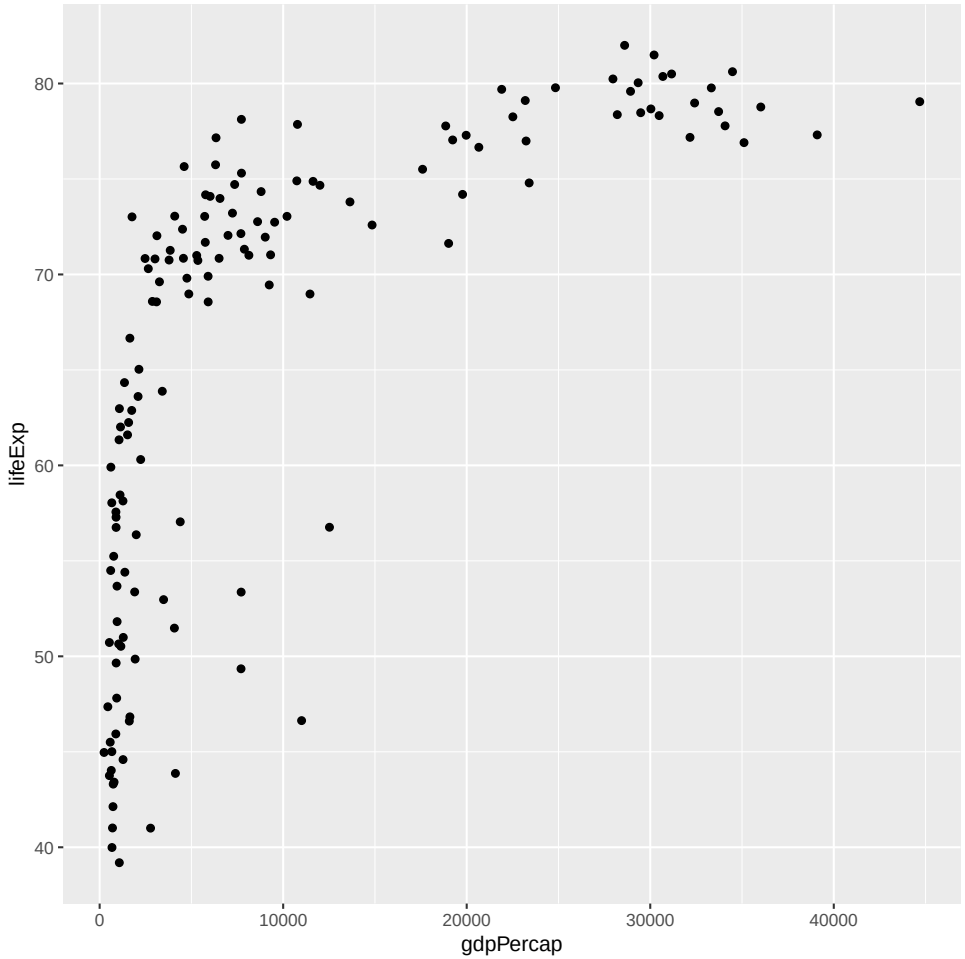
We begin with a simple scatterplot of life expectancy in years vs. GDP per capita in constant US dollars for all countries in 2002

Our plot is made using all the default settings in `ggplot2`

We've just mapped the data directly from the dataframe to `ggplot`'s scatterplot geometry

Critiques of this plot from cognitive and aesthetic perspectives?

Ideas for adding to and improving the plot?



Critiques of this plot from cognitive and aesthetic perspectives?

Gray background reduces contrast, pre-attentive distinction

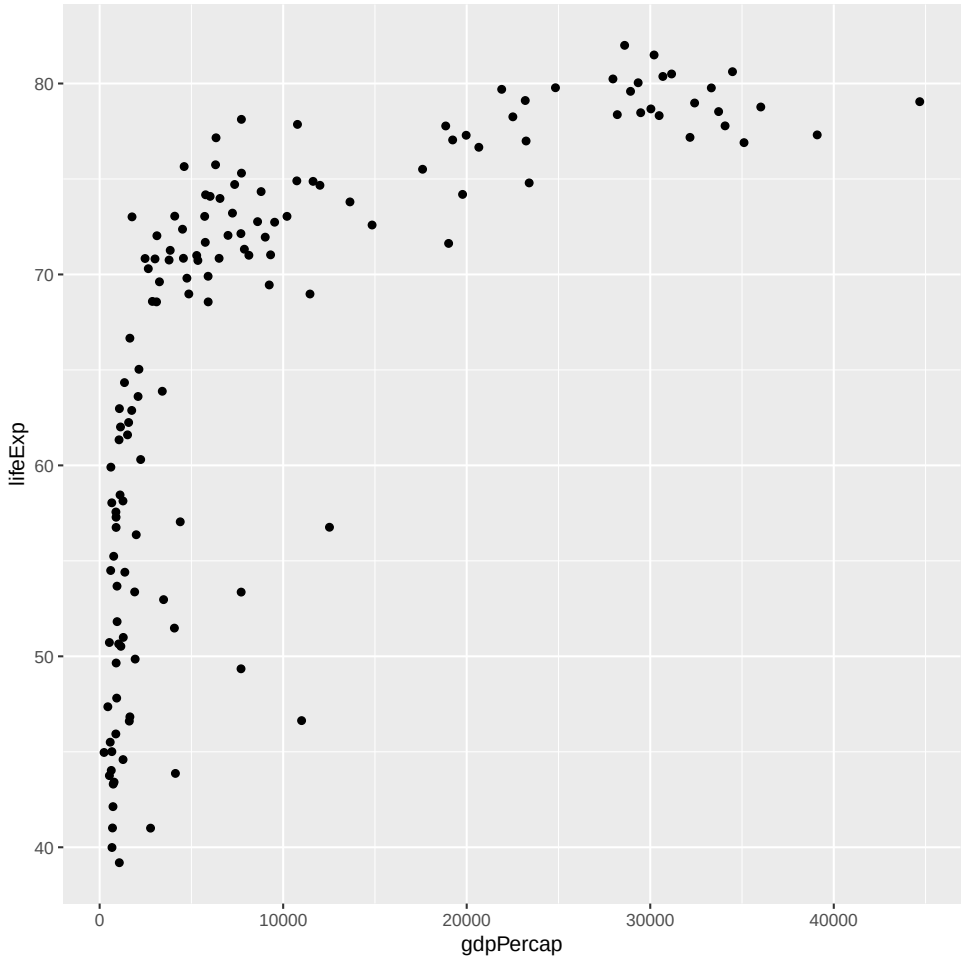
Two thicknesses of gridlines are excessive

Tick contrast is too high

Would a wider aspect be better?

Larger, more consistent text?

Hard to assess density with overlapping solid circles



Ideas for adding to and improving the plot?

Clear, interpretable titles

Candidate for log-scaling of *x*

Label (some) points?

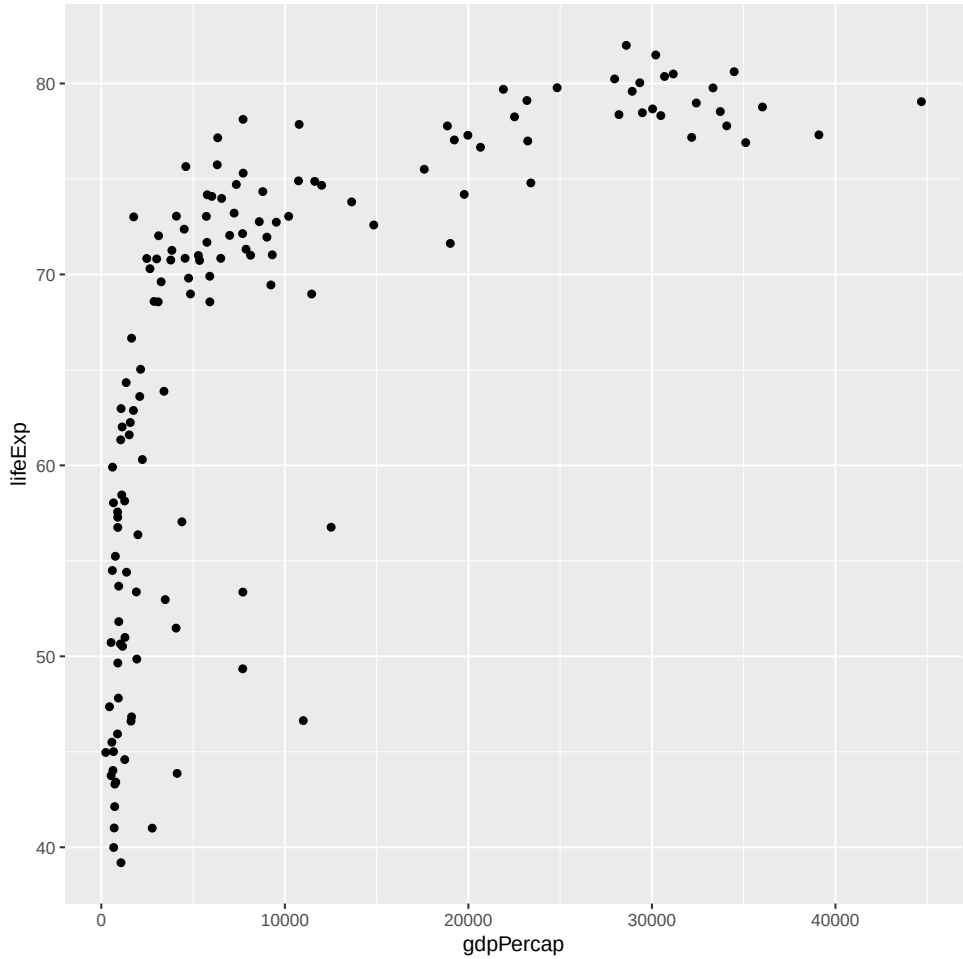
Add fit lines and CIs?

Use color and size to code other variables?

Small multiples?

Complete graphics code:

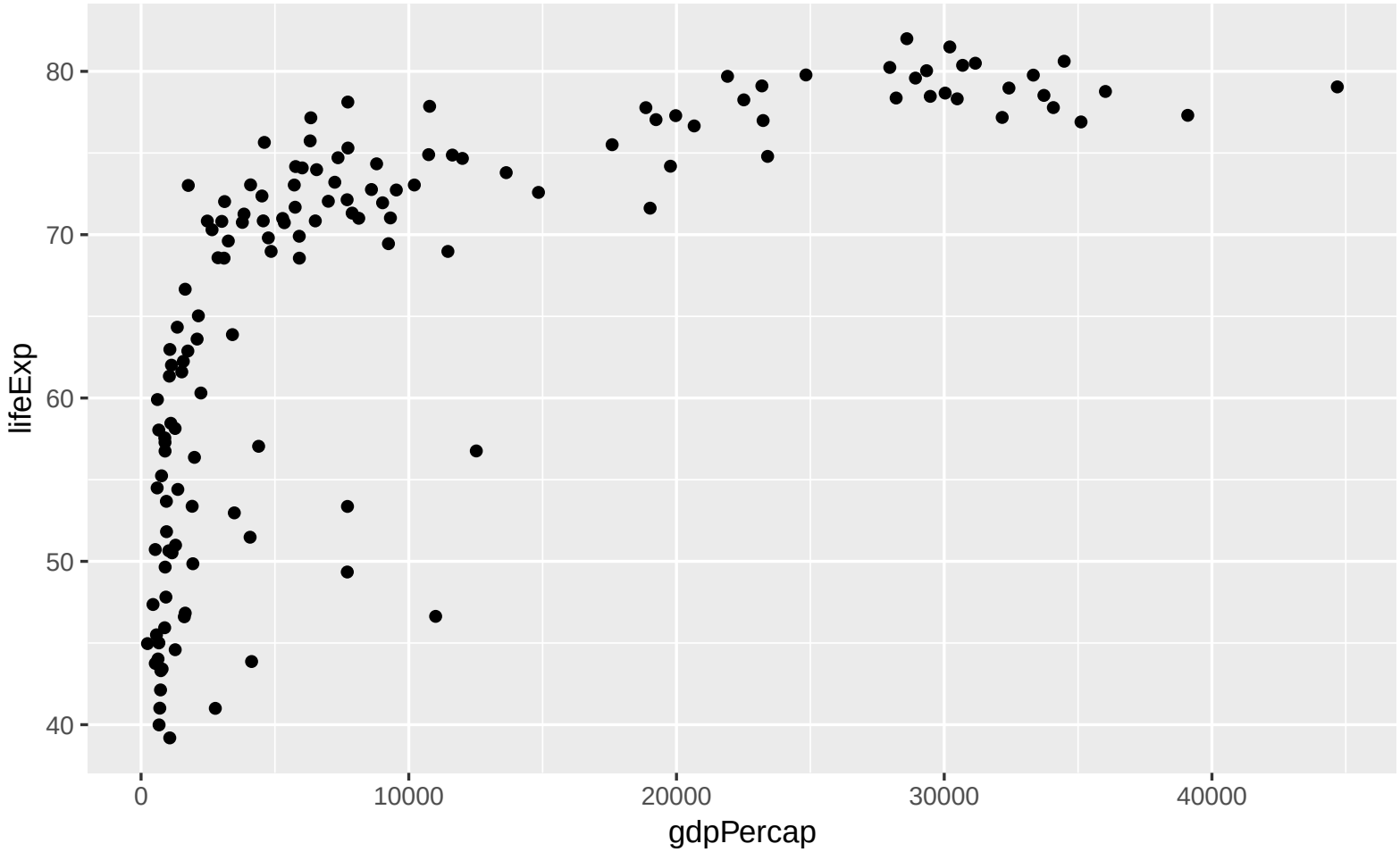
```
## Map data to graphical dimensions
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp))

## Establish graphical style
p <- p +
     geom_point()

## Save resulting plot to pdf
ggsave("ggScatterEx1_1.pdf")
```



We will review changes to this code step-by-step

Build a scatterplot using `ggplot2` *All 30 steps*

1.1	Default settings
1.2	Change aspect ratio
1.3	Custom axis titles
2.1	Adding color to glyphs
2.2	Changing color schemes
2.3	Device dimensions
3.1	Logging axes
3.2	Custom axis labels
3.3	Transparent points
4.1	A cleaner theme
5.1	Fits and smooths
5.2	Selecting data to fit
5.3	Layer graphical objects
5.4	Linear Fits
5.5	Custom Fits
6.1	Move the legend
6.2	Remove the legend
6.3	Legend in the plot
7.1	Bubble plots
7.2	Custom legend labels
7.3	Labeling points
7.4	Selective labeling
8.1	Small multiples
8.2	Remove unused color
8.3	Efficient layout
9.1	Annotate for storytelling
9.2	Select categorical colors
9.3	Clean up missing data
10.1	Zoom in on the story
10.2	Color cleanup

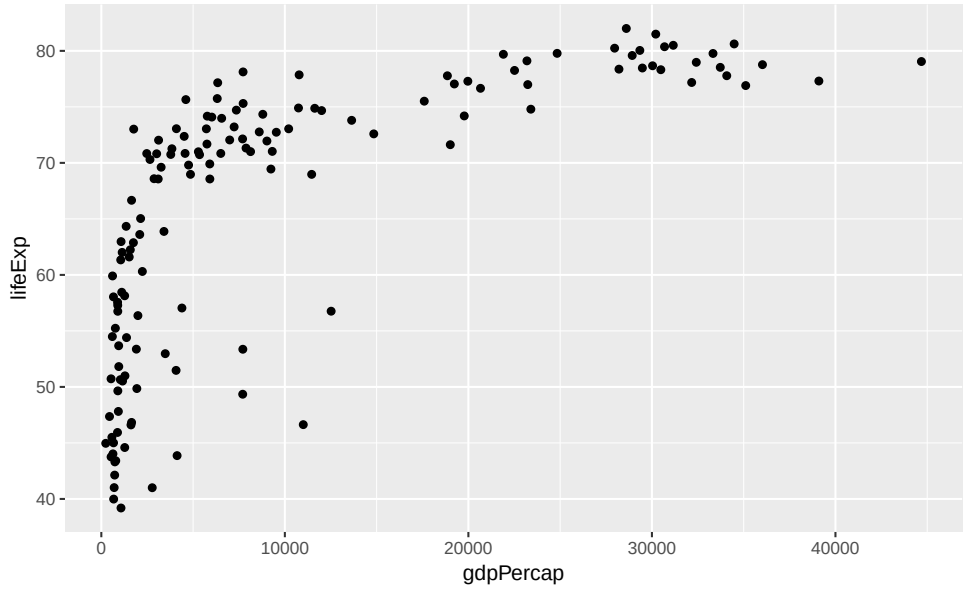


For these data, a golden rectangle seems like a better aspect ratio than a square

An easy way to implement this is by changing the ggsave() parameters

Complete graphics code:

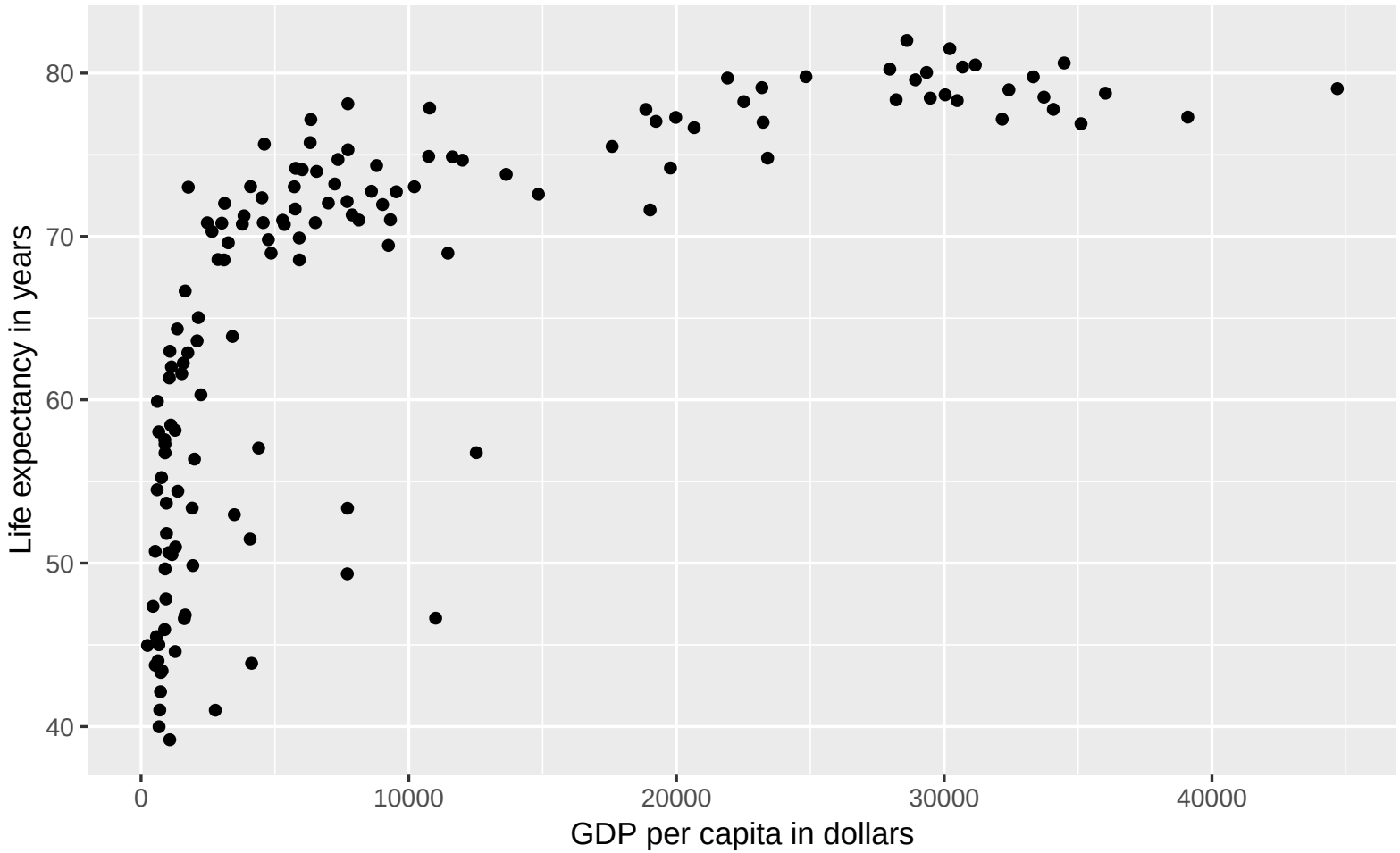
```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp))
p <- p +
     geom_point()

width <- 7
ggsave("ggScatterEx1_2.pdf",
       width=width,
       height=width/1.618)
```



Unless told otherwise,
ggsave() *creates a 7x7 inch graphic*

1.618 is the "Golden ratio"

Next: let's fix those axis titles

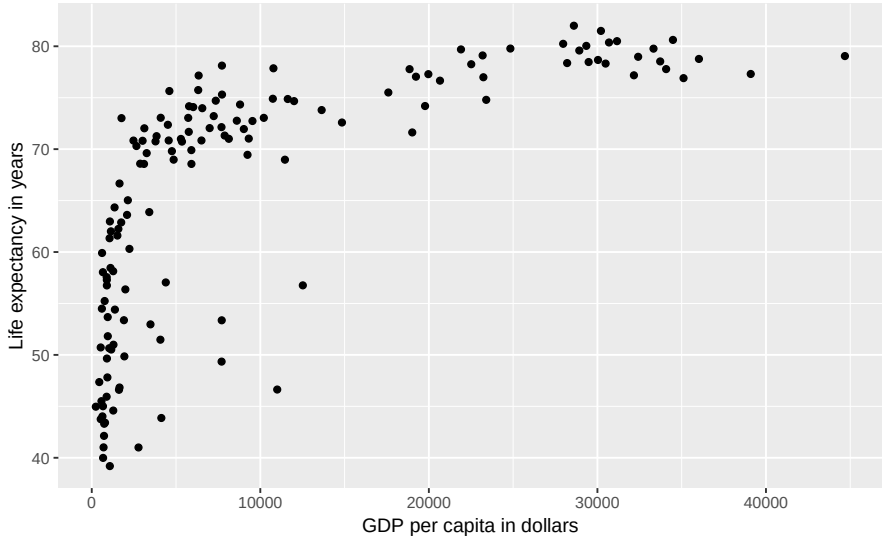


ggplot2 will always try to name axes and legends after the (often cryptic) names and values of variables: indeed, what else could it do?

This is seldom desirable – you should craft and add clear labels

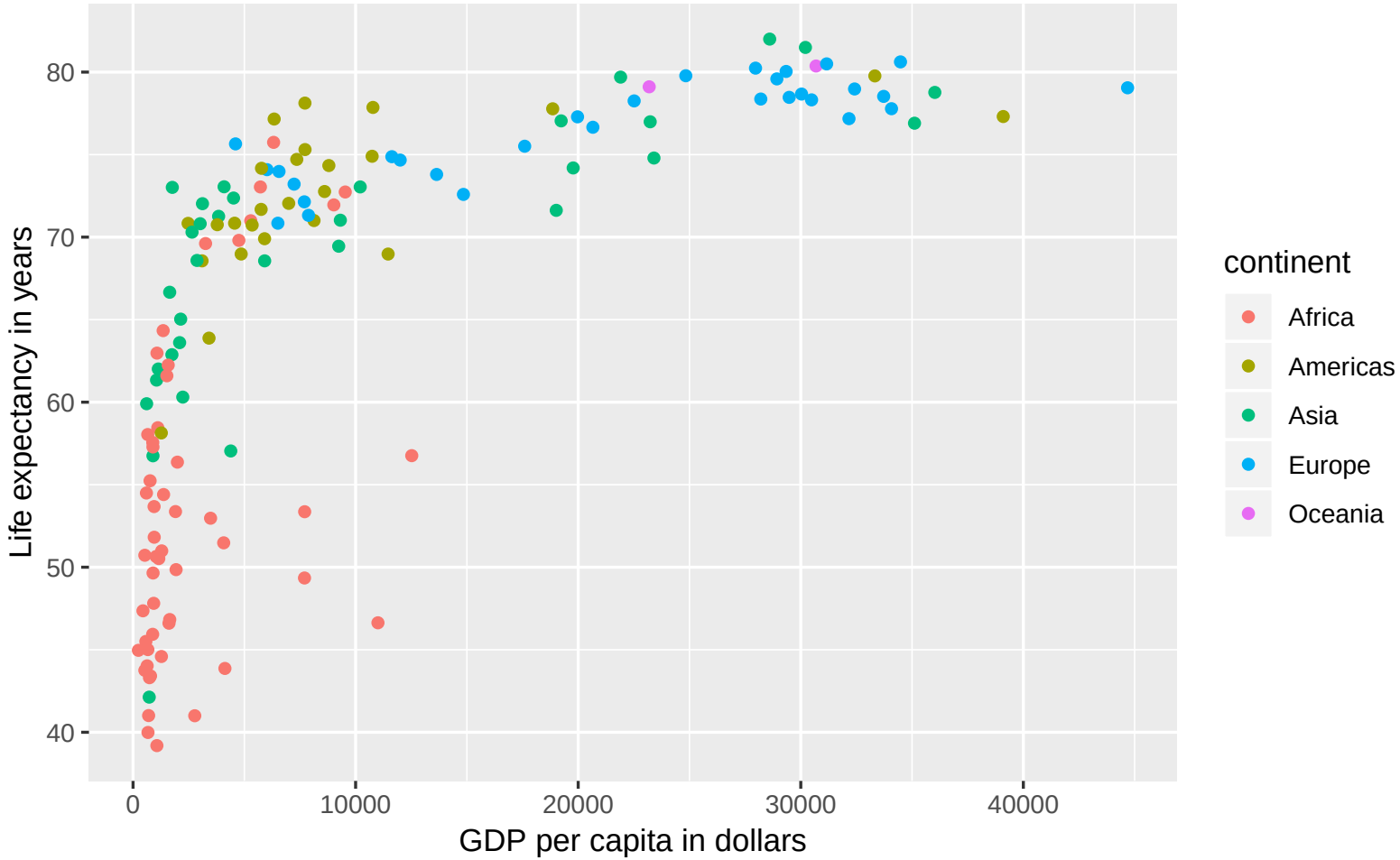
Complete graphics code:

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp))
p <- p +
     geom_point() +
     labs(x="GDP per capita in dollars",
          y="Life expectancy in years")

width <- 7
ggsave("ggScatterEx1_2.pdf",
       width=width,
       height=width/1.618)
```



Many labels can be modified using the `labs()` *function*

Next: How to add color to your glyphs to code qualitative variables

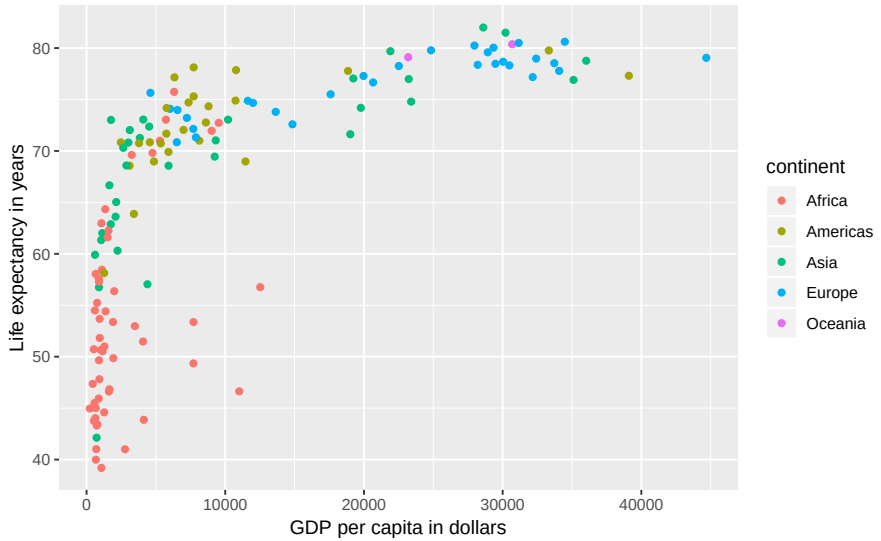


ggplot2 uses "Set2" from RColorBrewer by default

Cognitively, does this color scheme work? How could we improve it?

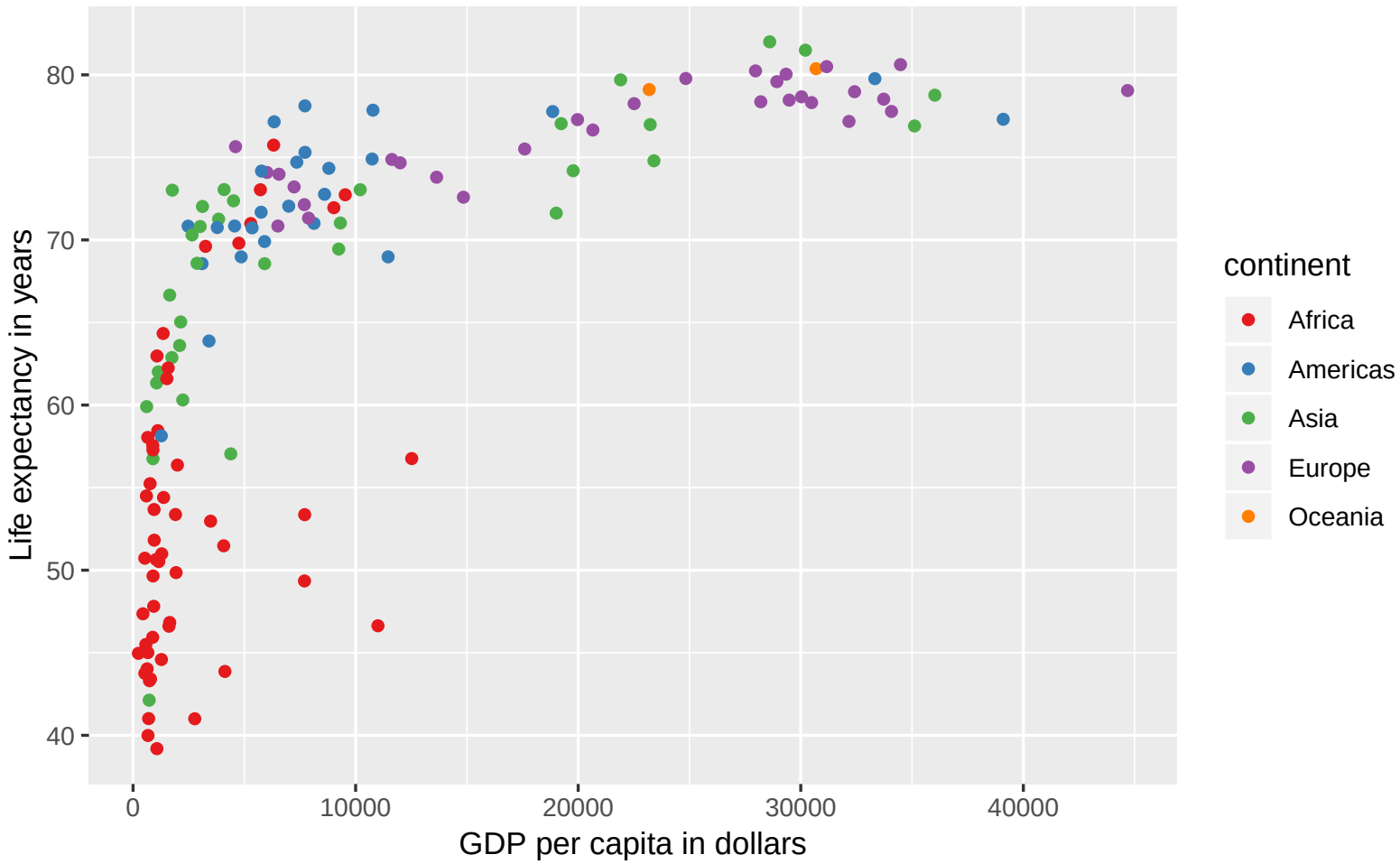
Complete graphics code:

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))
p <- p +
     geom_point() +
     labs(x="GDP per capita in dollars",
          y="Life expectancy in years")

width <- 7
ggsave("ggScatterEx2_1.pdf",
       width=width,
       height=width/1.618)
```



Aesthetic argument of the mapping function now globally defines region as the variable linked to color

For a more limited scope, instead add `color=continent` *to a specific* `geom`

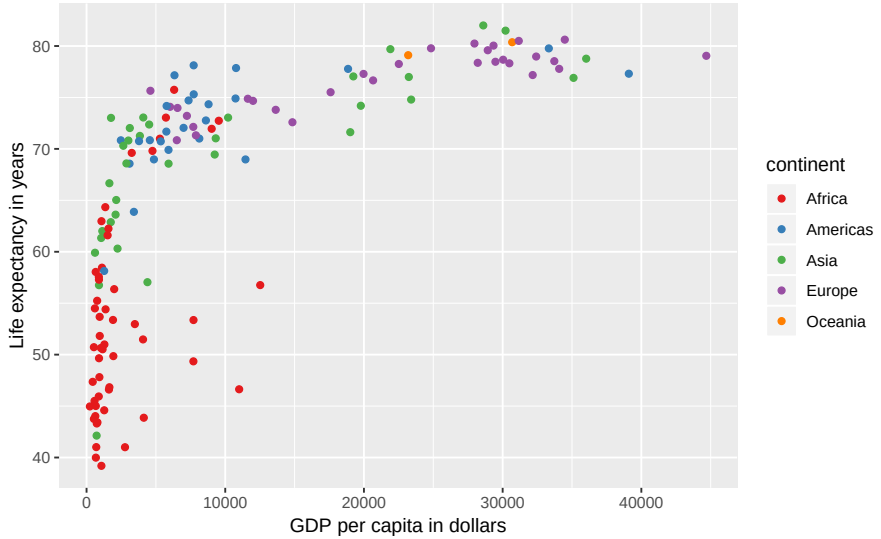


With 5 or more categories, RColorBrewer's Set2 probably works better than Set1

Recall the key to equal pairwise distinctions: similar distances between colors in CIElab

Complete graphics code:

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))
p <- p +
     geom_point() +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in dollars",
          y="Life expectancy in years")

width <- 7
ggsave("ggScatterEx2_1.pdf",
       width=width,
       height=width/1.618)
```



Why is this function called `scale_color_brewer`*?*

Check out the `ggplot2` *help index for many more* `scale_` *functions*

Aside: Colors in R

Three ways to specify a color to an R function (for all R graphics tools):

1. **color names, like** `"red"` **or** `"lightblue"` **– see** `colors()` **for a list**

Aside: Colors in R

Three ways to specify a color to an R function (for all R graphics tools):

1. color names, like `"red"` or `"lightblue"` – see `colors()` for a list
2. numerical color codes from `rgb()`, `hsv()`, or `hcl()`
   e.g., `hcl()` gives equal perceptual changes for hue, chroma, and luminance (brightness)
   Also useful: `col2rgb()`, `rgb2hsv()`, etc. for conversions among these functions

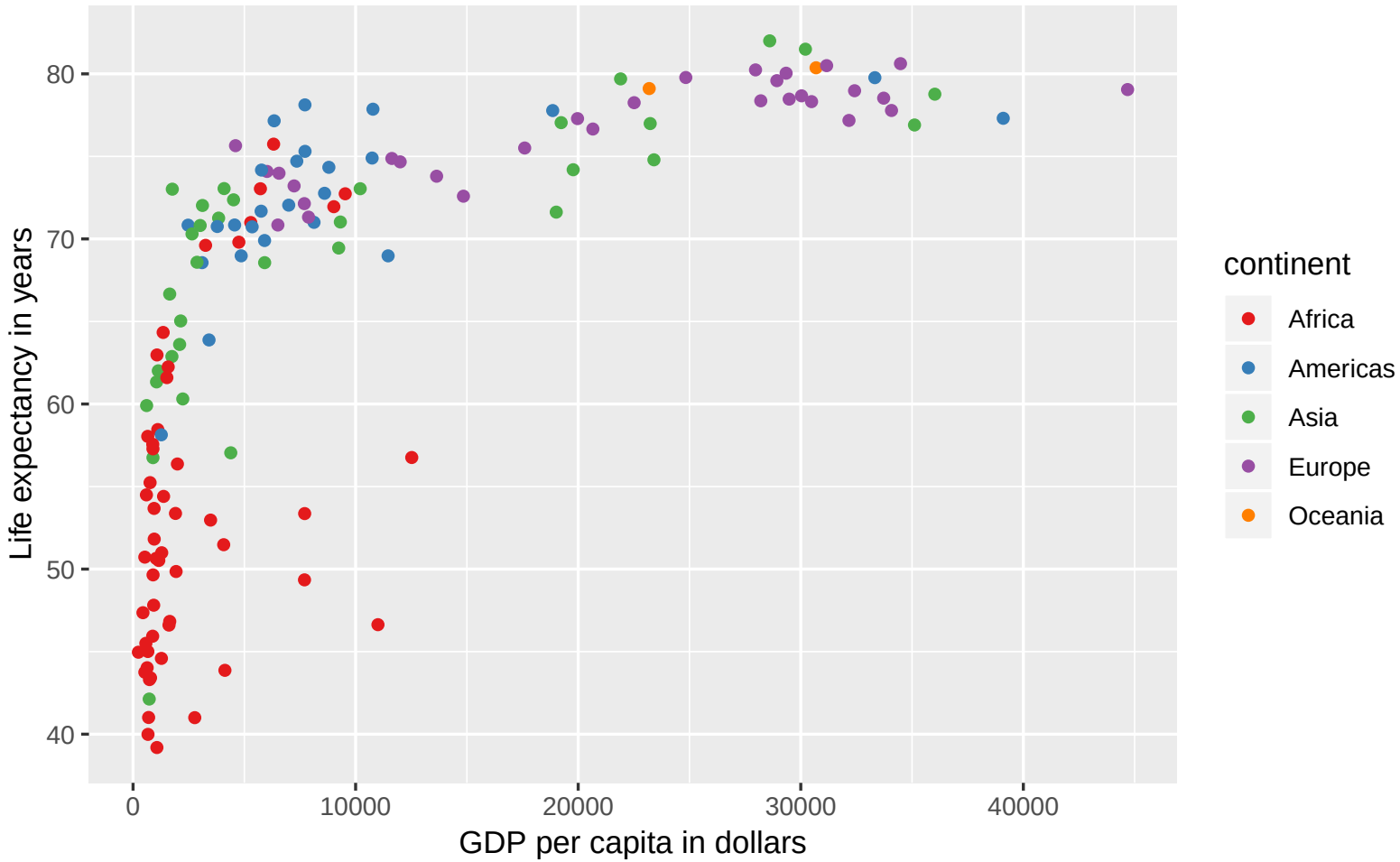
Aside: Colors in R

Three ways to specify a color to an R function (for all R graphics tools):

1. color names, like `"red"` or `"lightblue"` – see `colors()` for a list
2. numerical color codes from `rgb()`, `hsv()`, or `hcl()`
   e.g., `hcl()` gives equal perceptual changes for hue, chroma, and luminance (brightness)
   Also useful: `col2rgb()`, `rgb2hsv()`, etc. for conversions among these functions
3. hexadecimal color codes, usually offered by packages for selecting cognitively valid palattes optimized to your required number of colors and level of measurement (categorical, ordered, interval):

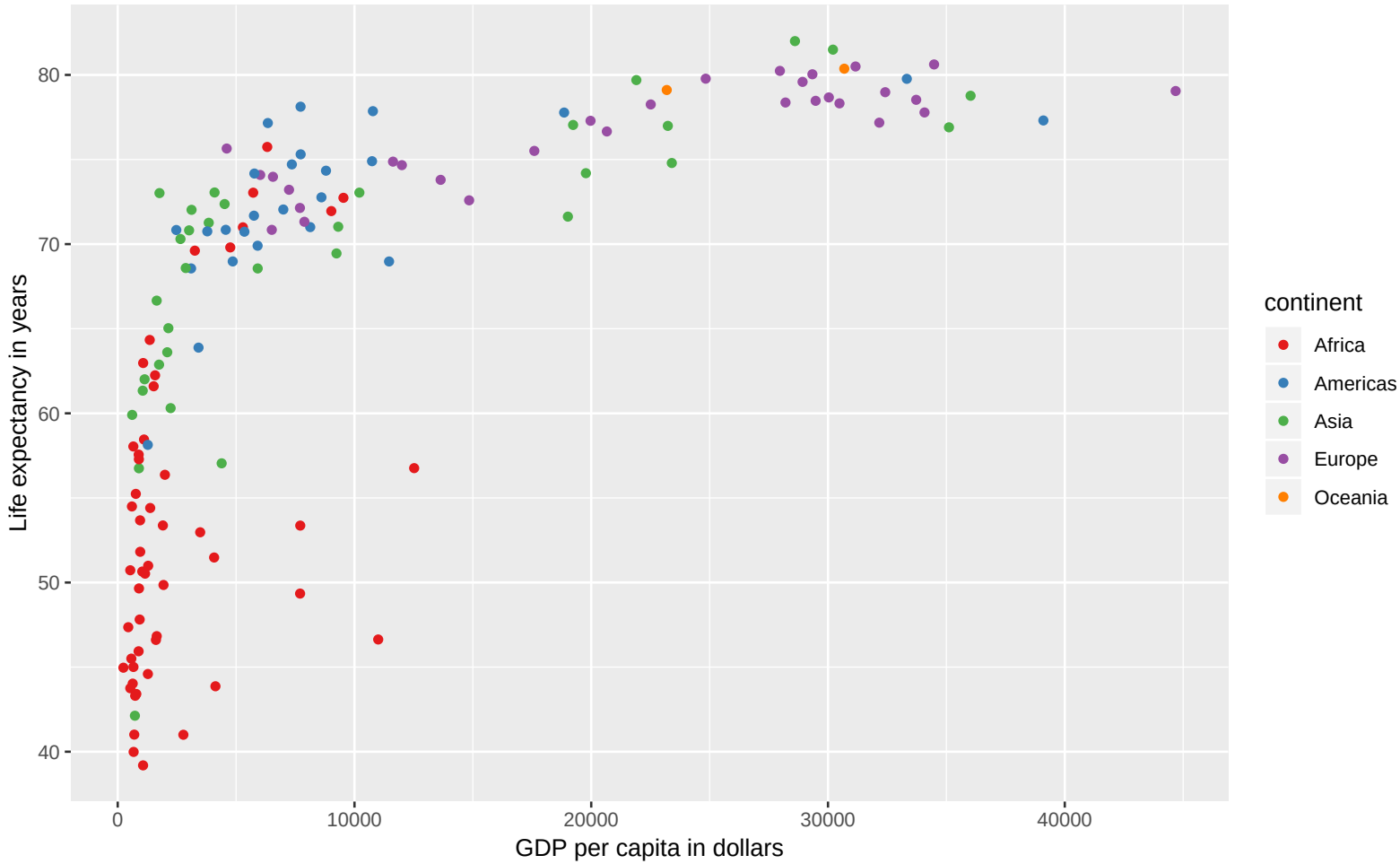
Package	Functions	Advantage
`RColorBrewer`	`brewer.pal()`	*fast & easy*
`colorspace`	`sequential_hcl()`, `diverge_hcl()`	*powerful*

I often save colors as objects for convenience – see top of code example



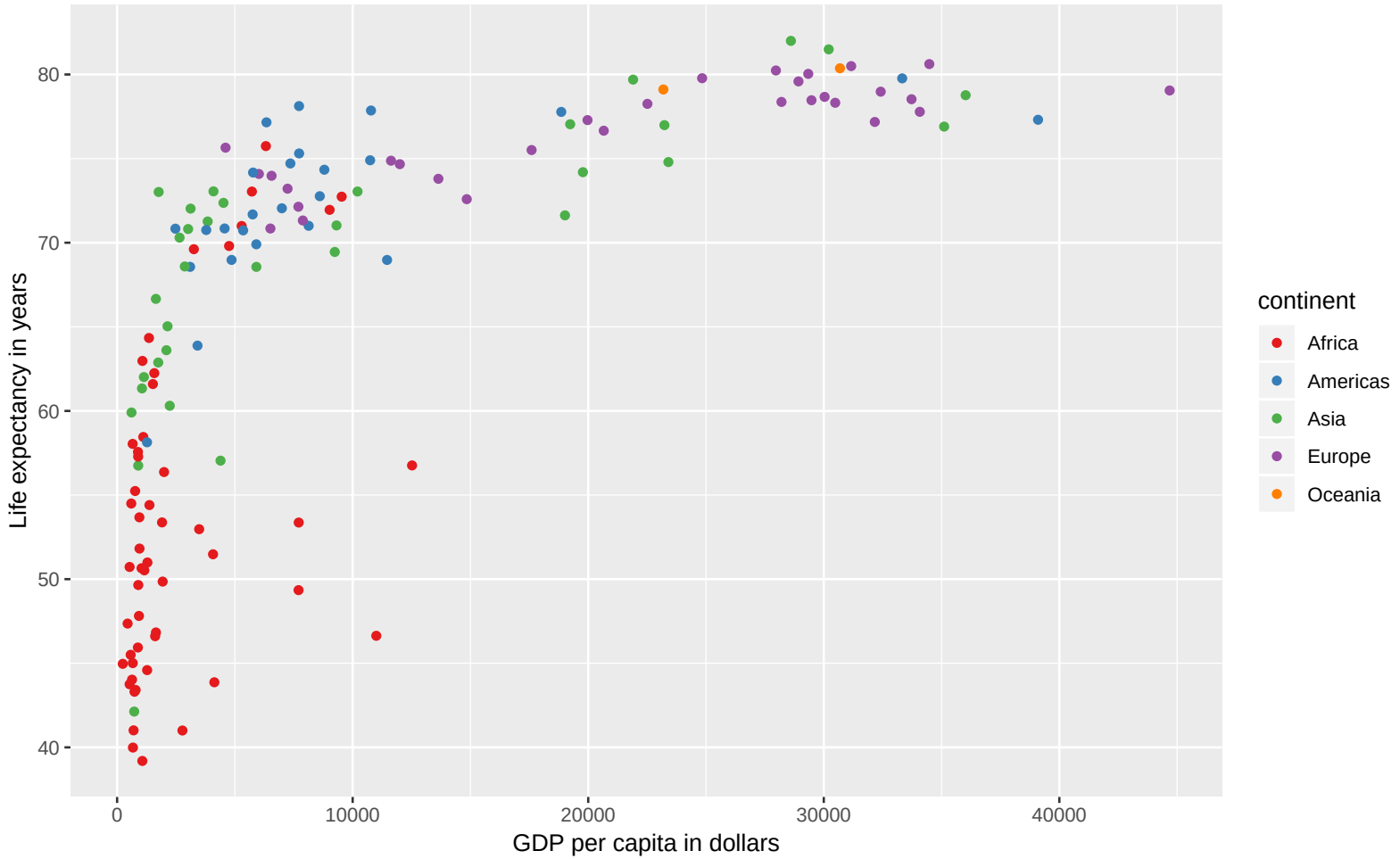
When you change the width & height arguments,
`ggsave()` tries to resize all plot elements appropriately

The above is the default width of 7 inches. Note the size of text and glyphs.



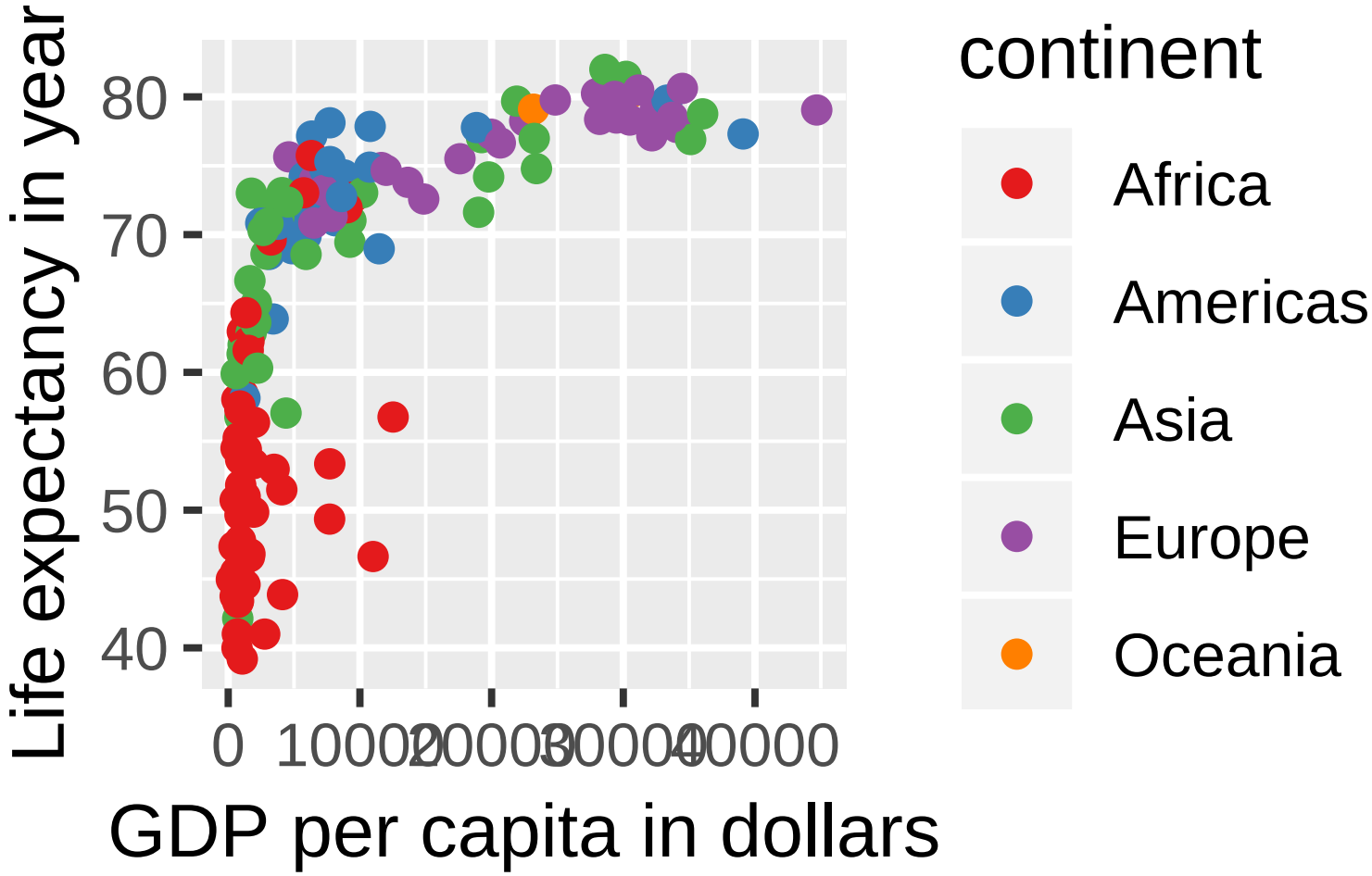
When you change the width & height arguments,
ggsave() tries to resize all plot elements appropriately

Now the width is 9 inches. Everything seems to have shrunk (counterintuitive?)



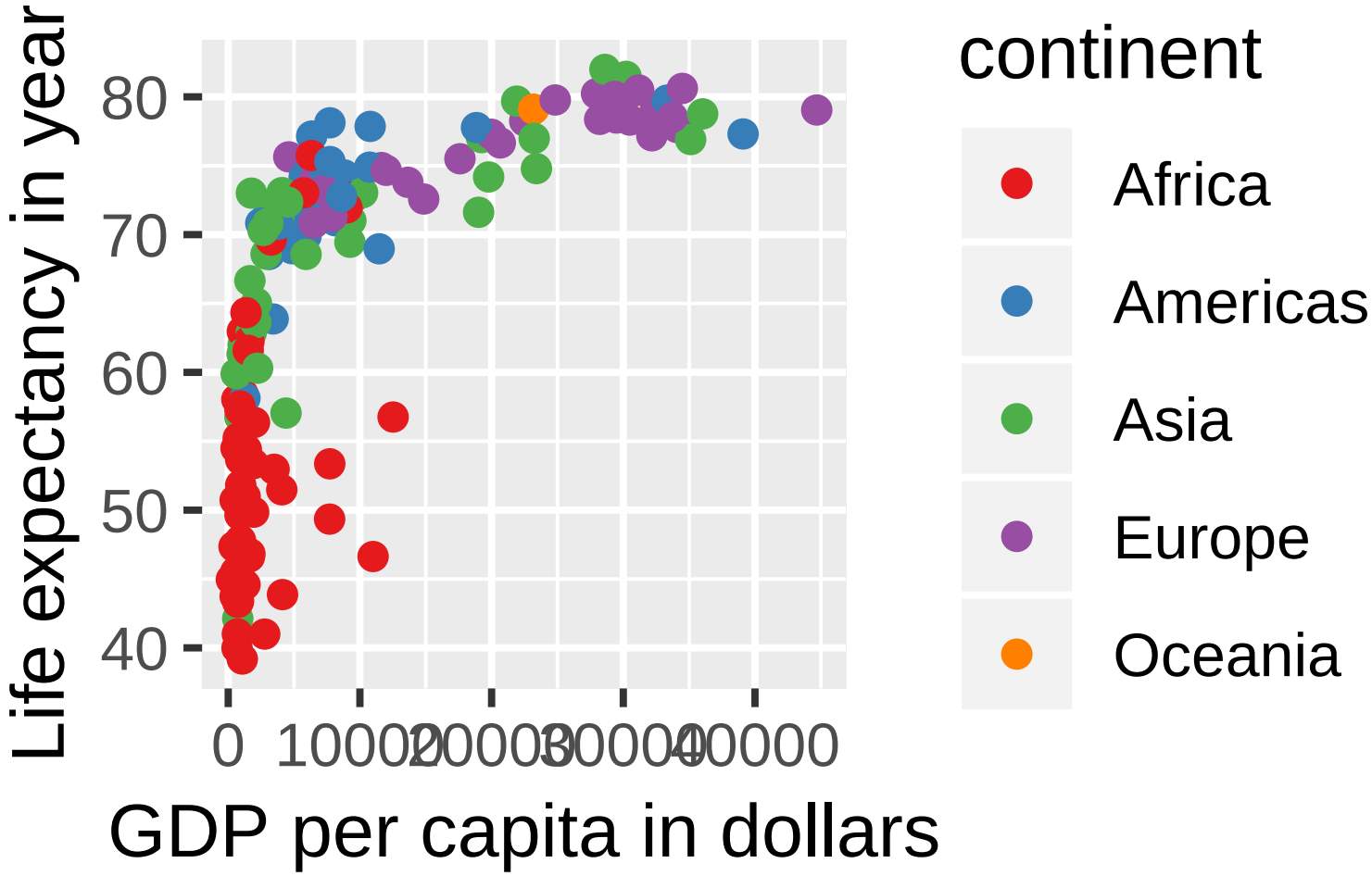
When you change the width & height arguments,
ggsave() tries to resize all plot elements appropriately

Unpleasant side effect: colors of points are harder to distinguish (cognitive limits)



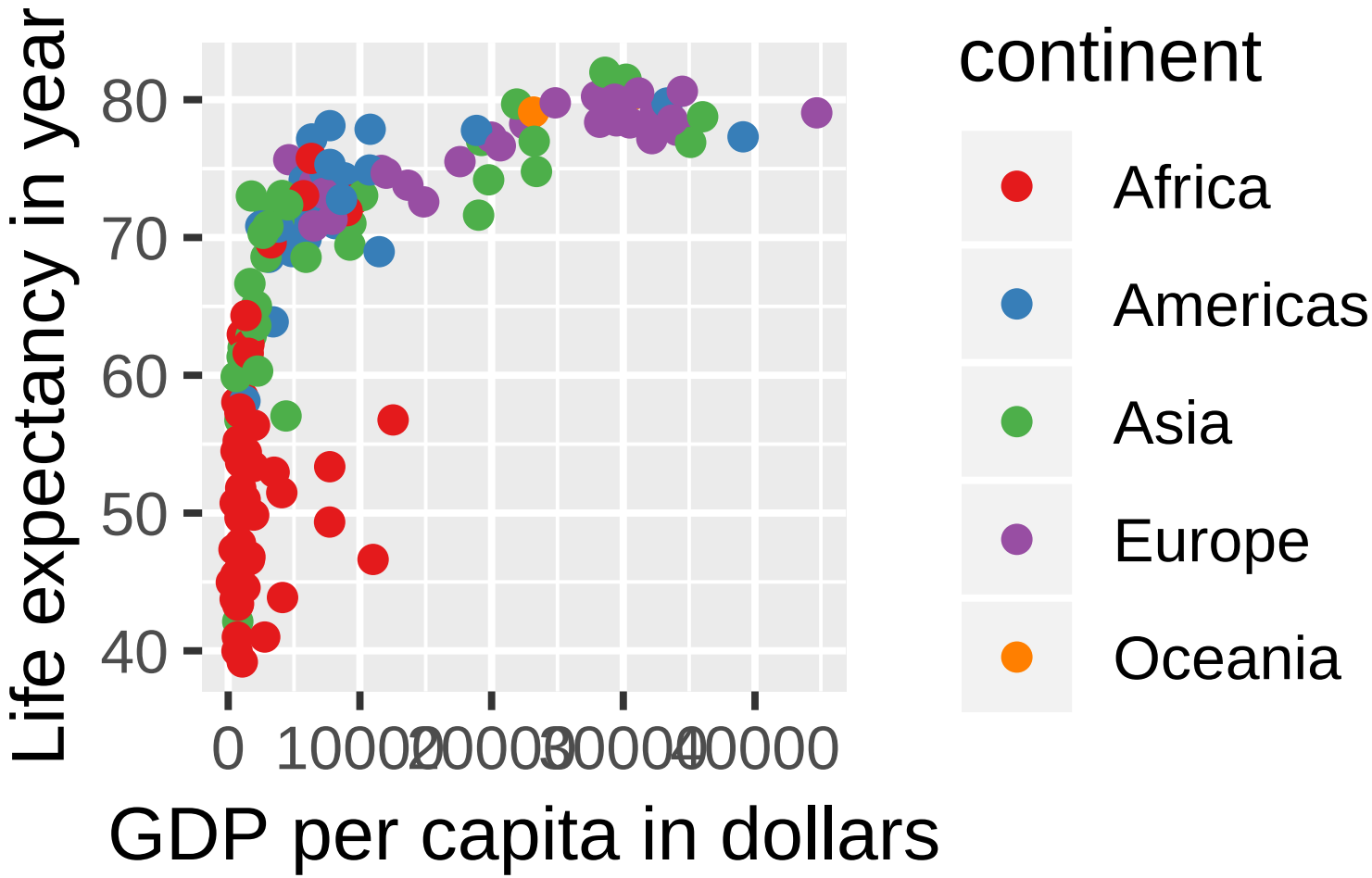
When you change the width & height arguments, `ggsave()` tries to resize all plot elements appropriately

Now the width is 3 inches. Everything seems to have grown (counterintuitive?)



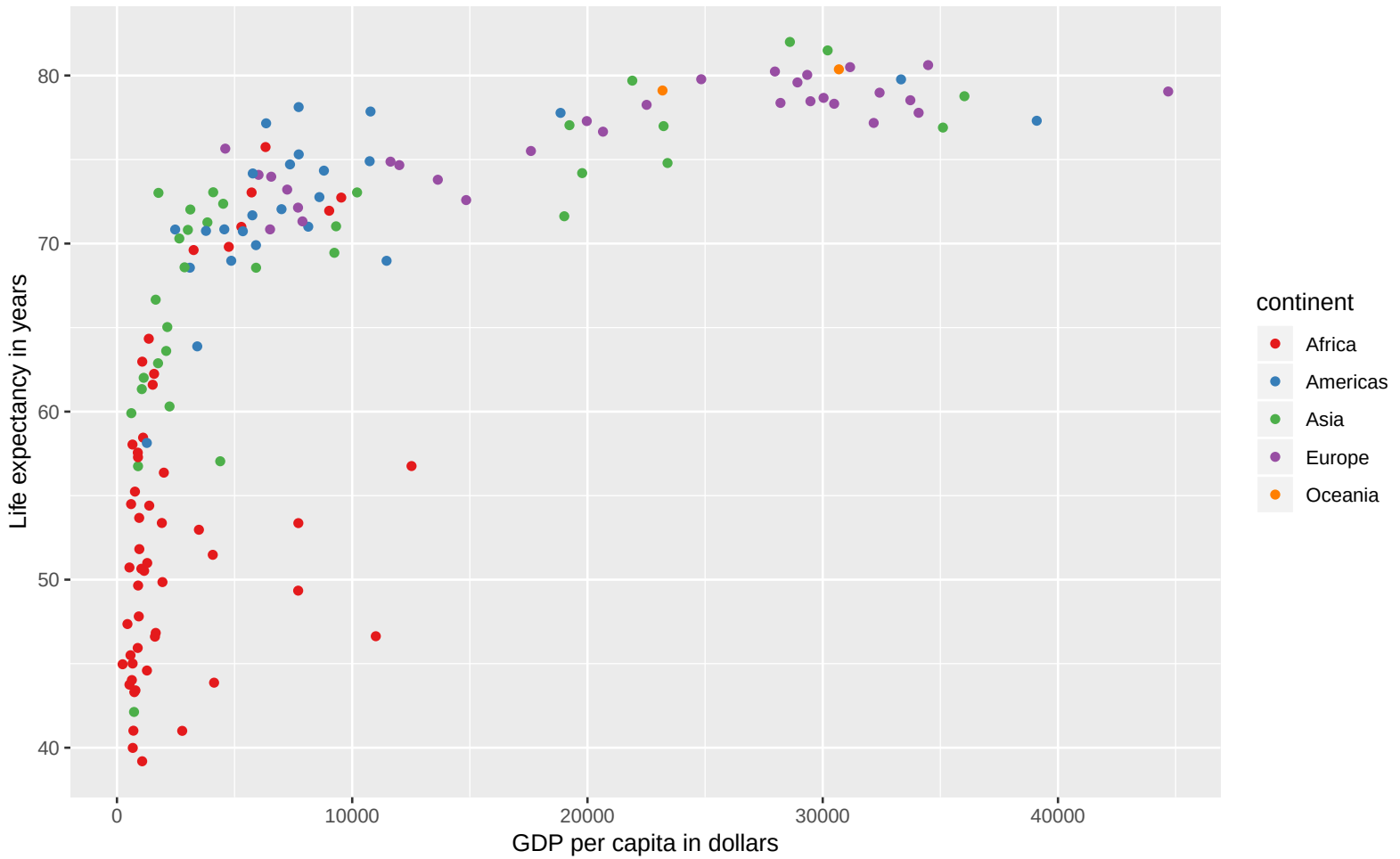
When you change the width & height arguments, `ggsave()` tries to resize all plot elements appropriately

Unpleasant side effects: plot elements overlap, and plotting region shrinks



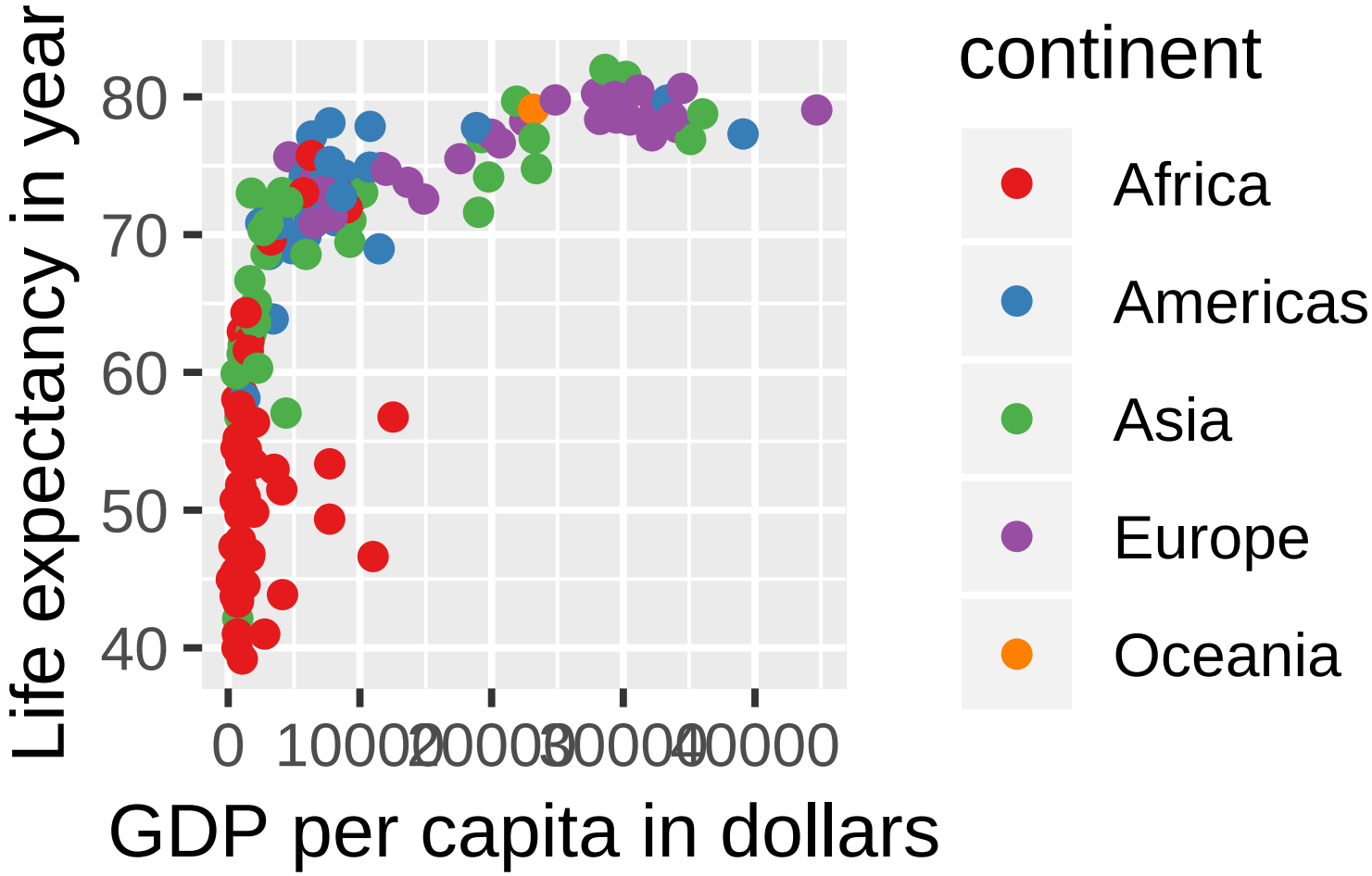
Consider this analogy business card : poster :: narrow device : wide device

If you print a business card to the width of a page, you'd have huge letters



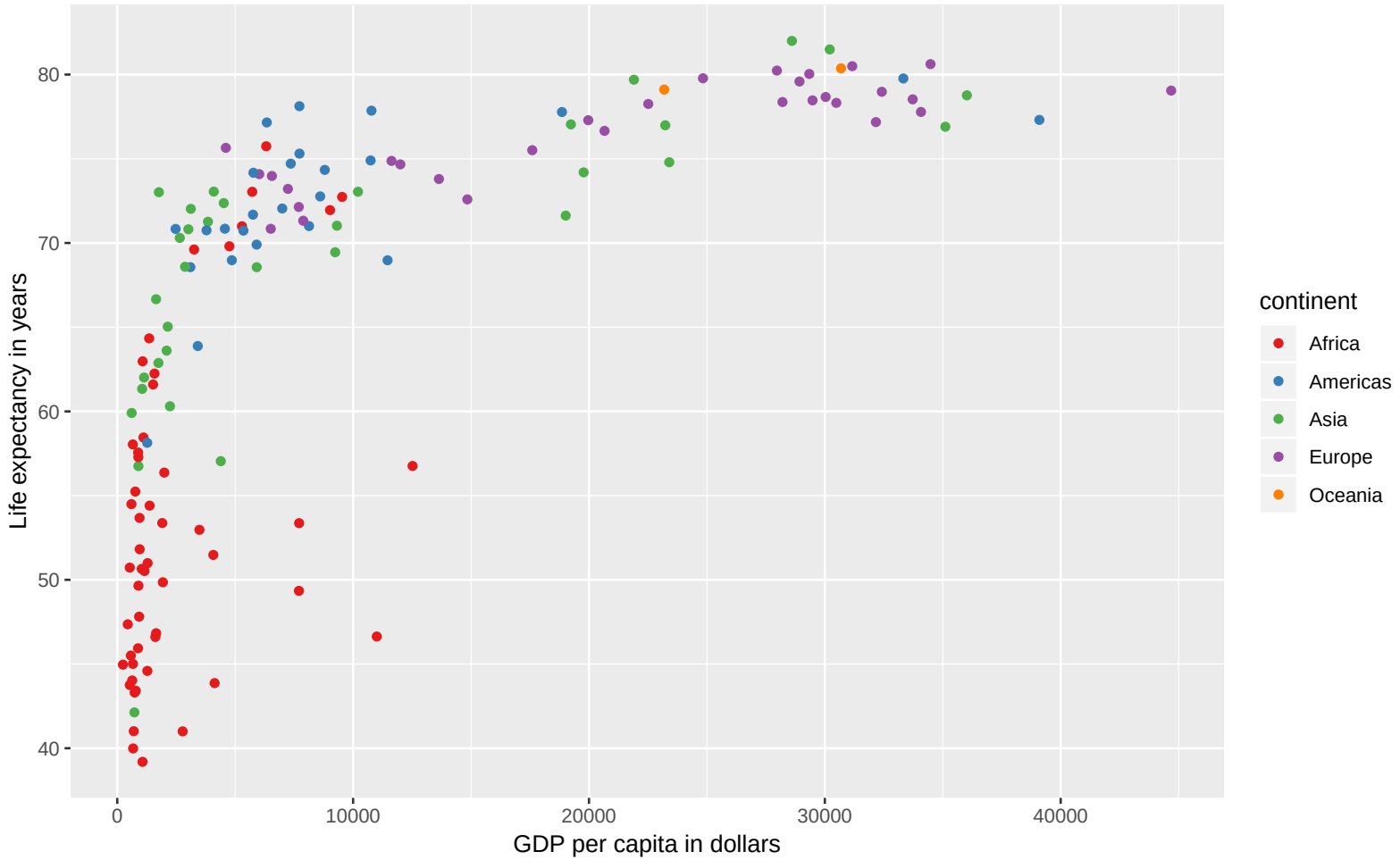
Consider this analogy business card : poster :: narrow device : wide device

If you print a poster to the width of a page, you'd have tiny letters



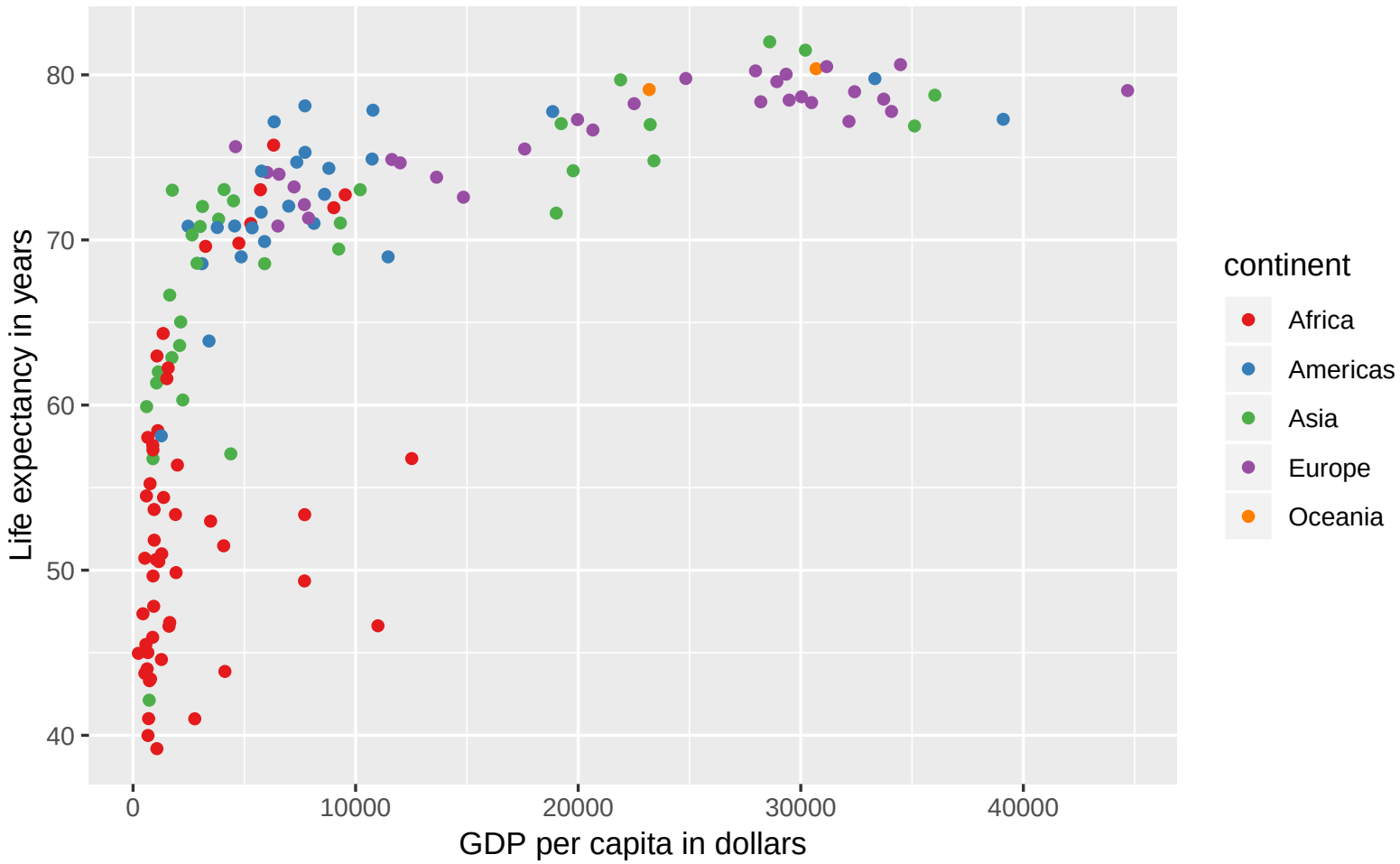
The lesson: you can change most of the sizes in your plot in one line of code by resizing the *device* instead of the elements themselves

If everything is too big (and overlapping), make the device *bigger!*

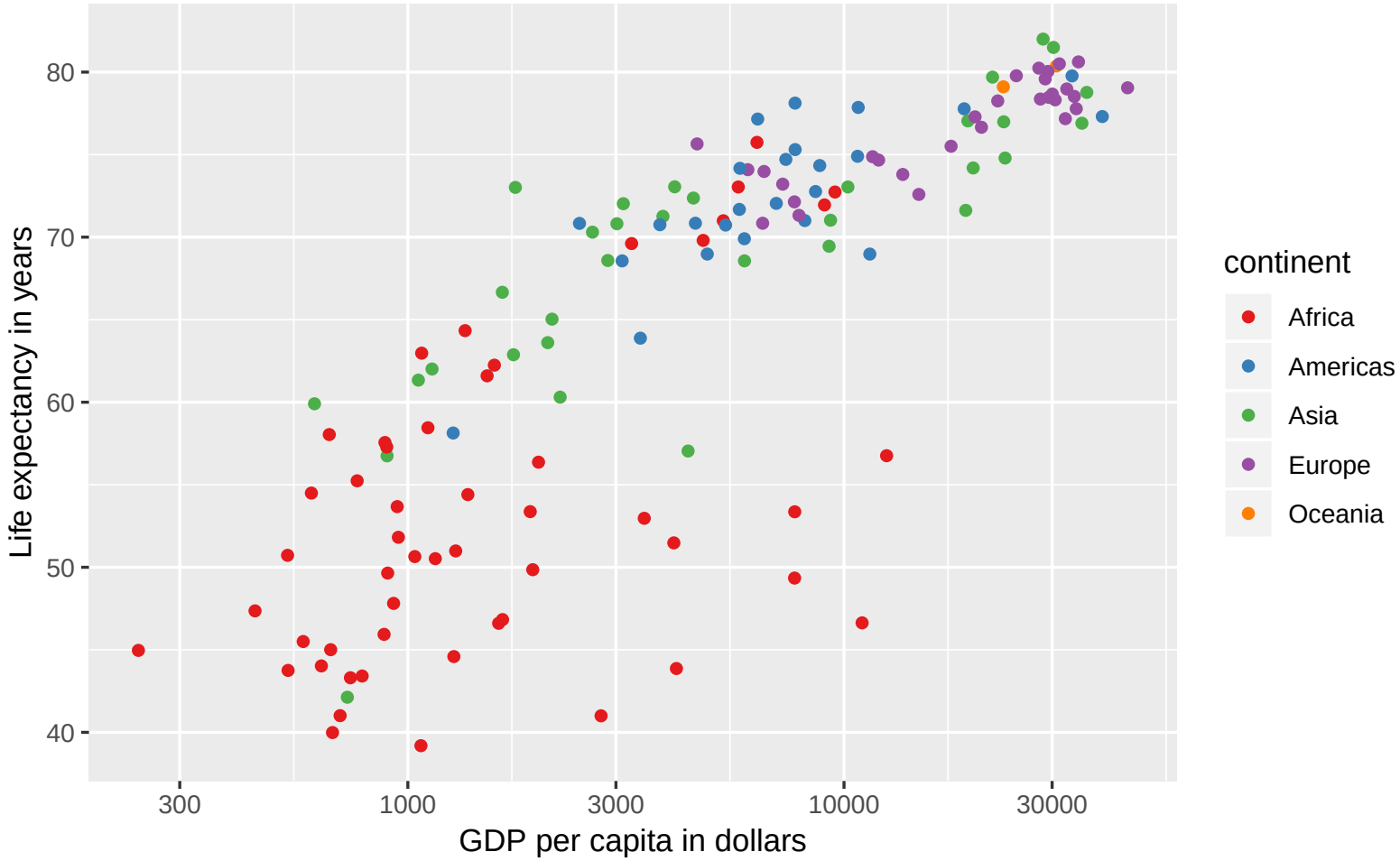


The lesson: you can change most of the sizes in your plot in one line of code by resizing the *device* instead of the elements themselves

If everything is too small (and spread out), make the device *smaller!*



Our data follow a curve, and are mostly crammed against the lower end of the horizontal axis, which is strictly positive – suggest this axis be logged

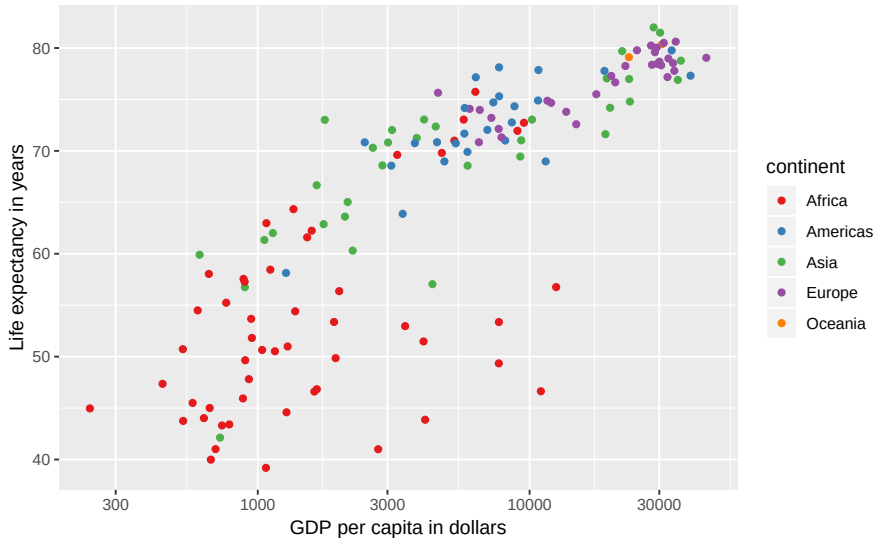


Above shows the results of letting ggplot2 choose the labels of the logged axis

Plot looks linear in the log of GDP per capita, with some outliers (theories?)

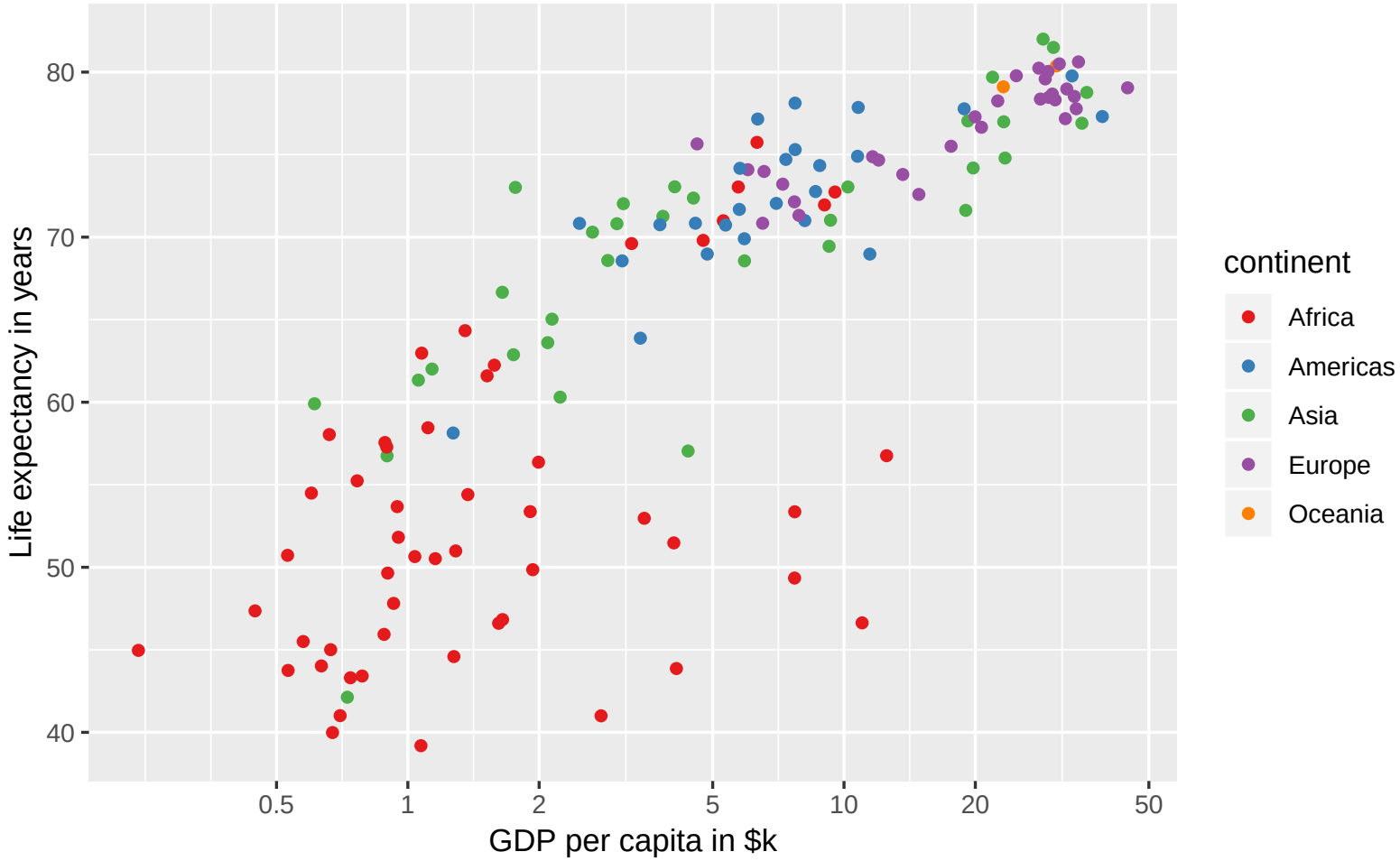
Complete graphics code:

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))
p <- p +
     geom_point() +
     scale_x_log10() +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in dollars",
          y="Life expectancy in years")

width <- 7
ggsave("ggScatterEx3_1.pdf",
       width=width,
       height=width/1.618)
```



`scale_x_log10()` *is one of many options for transforming axes*

But what if we don't like the ticks or labels chosen by `scale_x_log10()`*?*



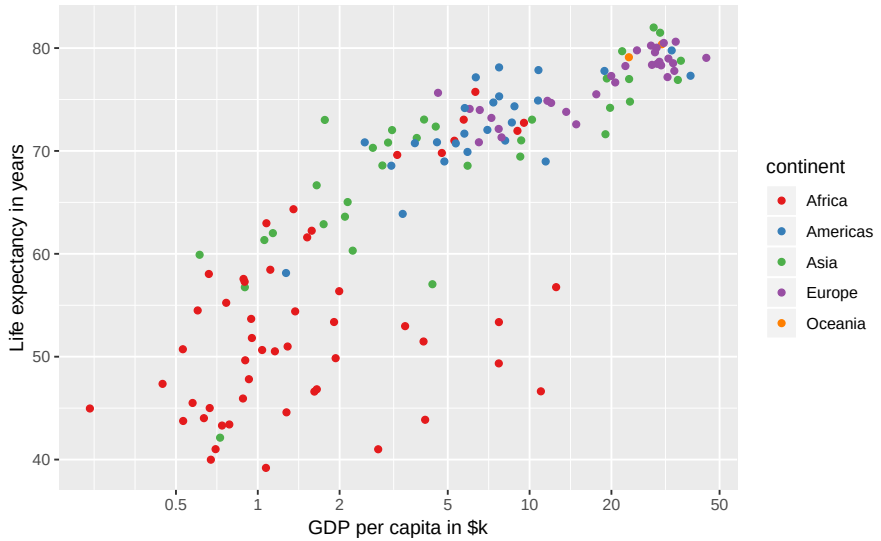
When labeling logged axes, we often want to divide by a unit to avoid long numbers

To do this – or to change the tick locations – we need to customize the axis

Partial code:

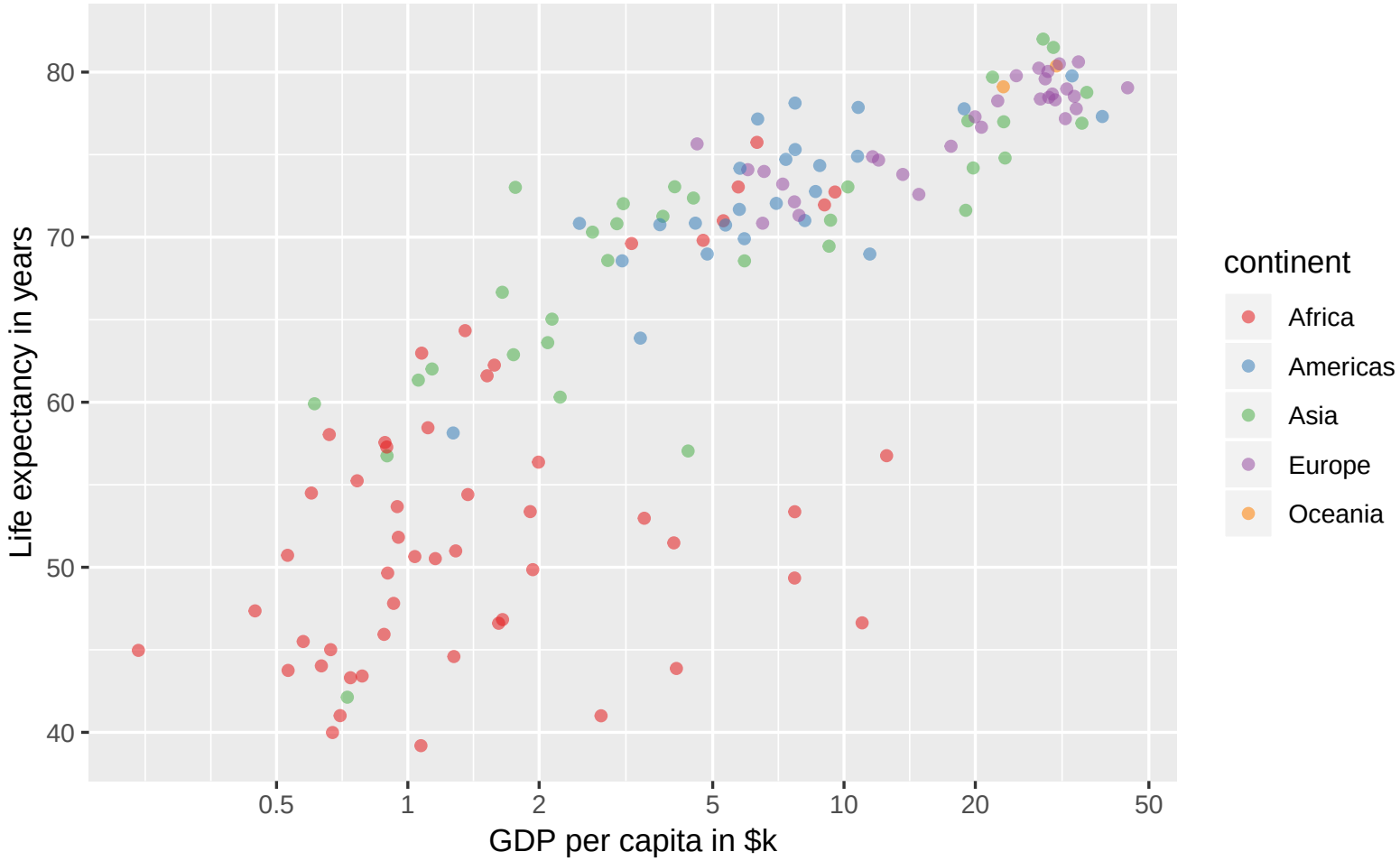
(mapping and saving as above)

```
xbreaks <- c(500, 1000, 2000, 5000,
             10000, 20000, 50000)


p <- p +
     geom_point() +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



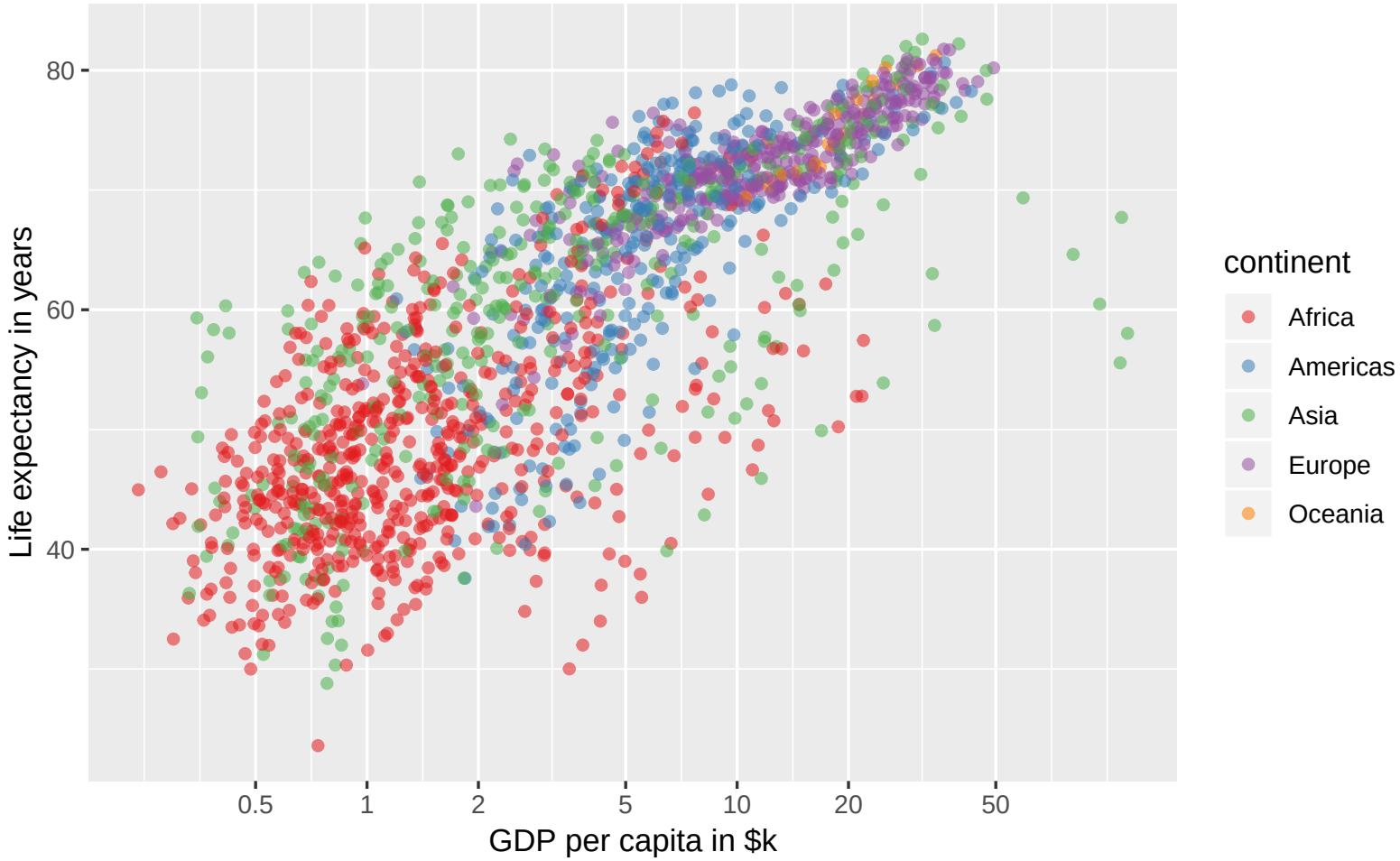
We can pass many arguments to `scale_x_log10()`*, including separate values for tick locations* (`breaks`) *and the labels of those ticks* (`labels`)

Separating these elements allows powerful customization



Even with axes logged, numerous points overlap, making density hard to judge

Allowing partial transparency is an elegant solution



Even with axes logged, numerous points overlap, making density hard to judge

Allowing partial transparency is an elegant solution – especially in large datasets

Partial code:

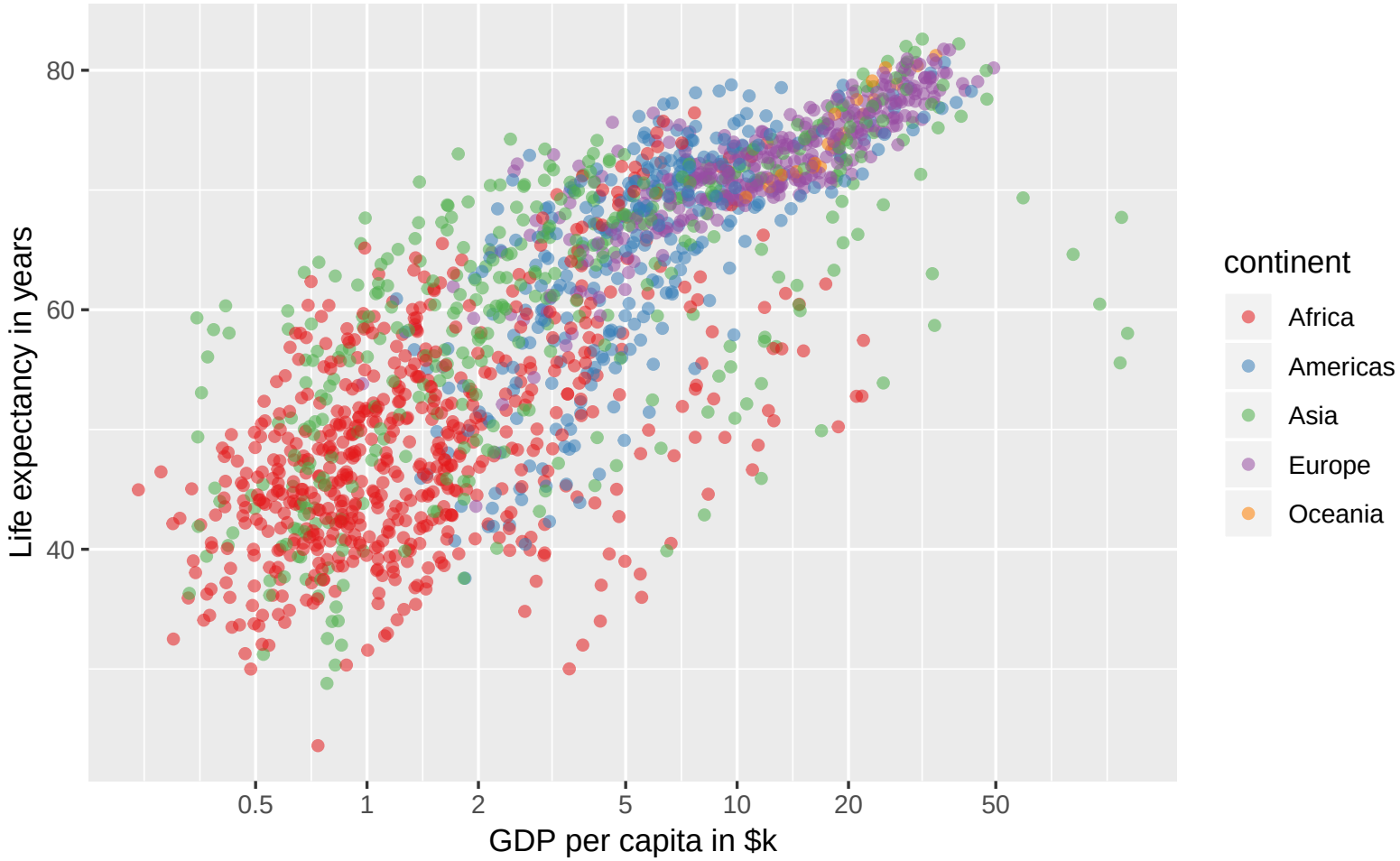
(mapping and saving as above)

```
xbreaks <- c(500, 1000, 2000, 5000,
             10000, 20000, 50000)


p <- p +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```

alpha *stands for transparency, regardless of your graphics package*

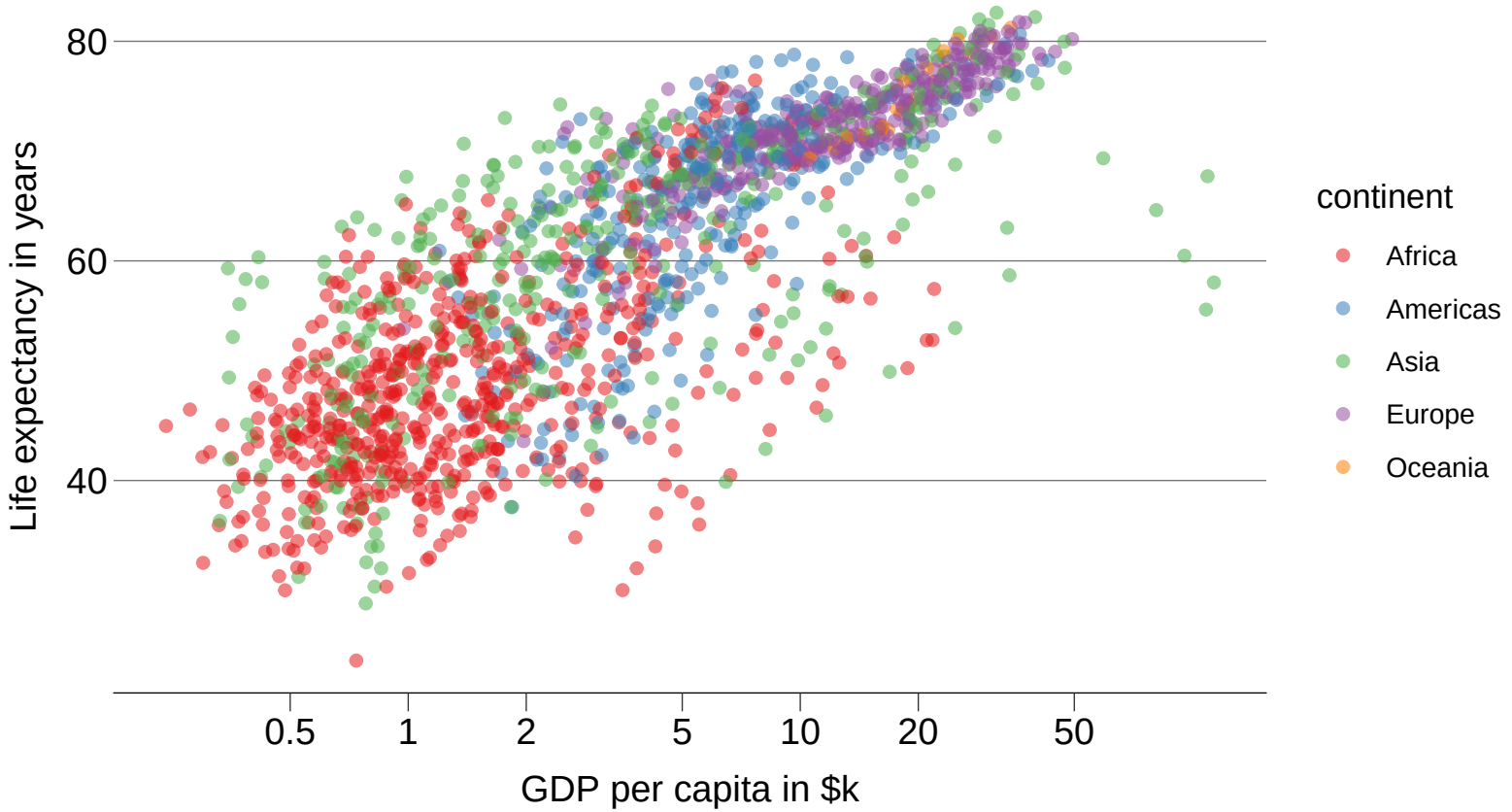
Functions as a fourth color channel

Range from 0 (invisible) to 1 (opaque)

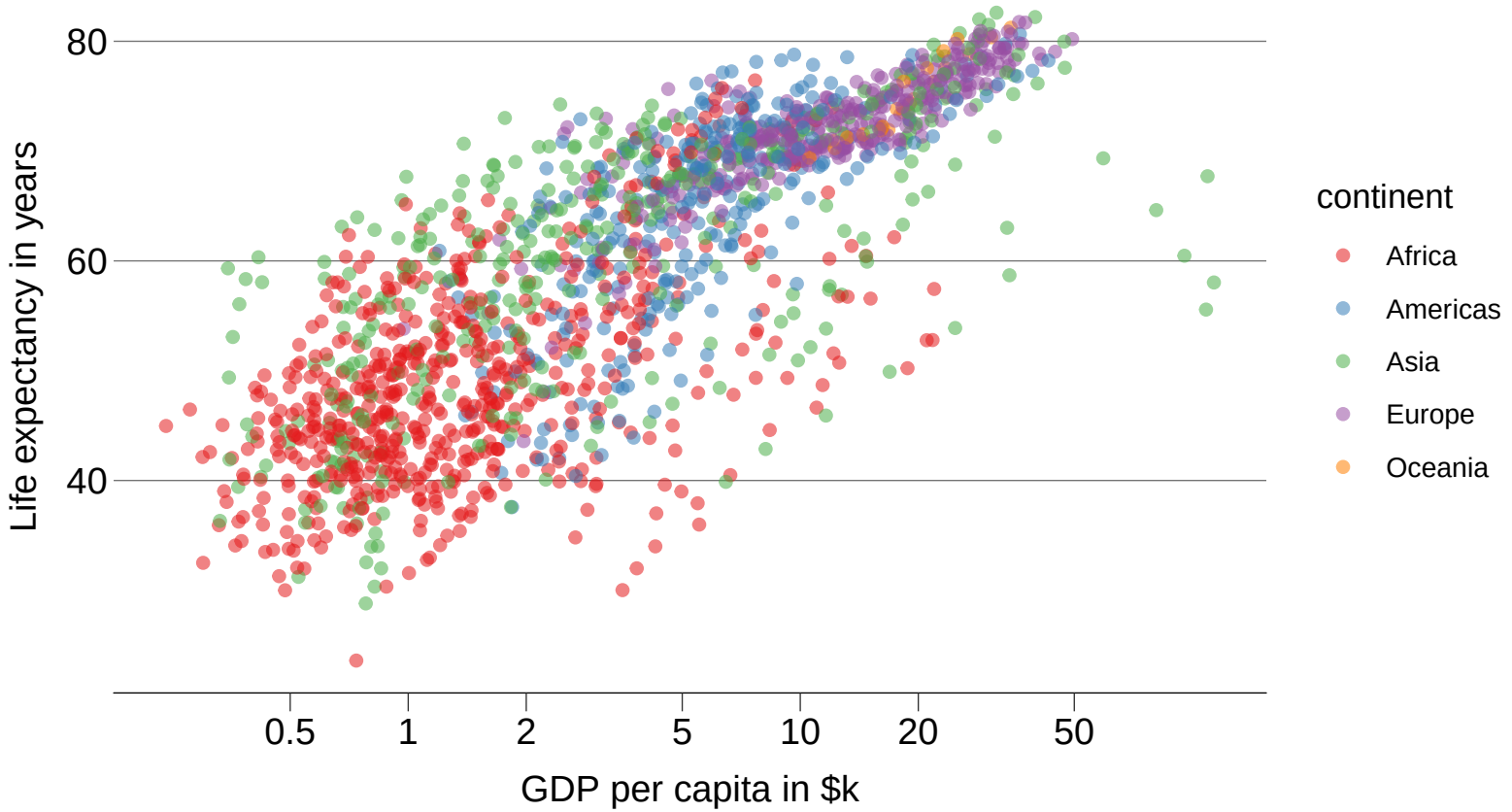


Many of the cognitive and aesthetic issues with this plot are matters of style

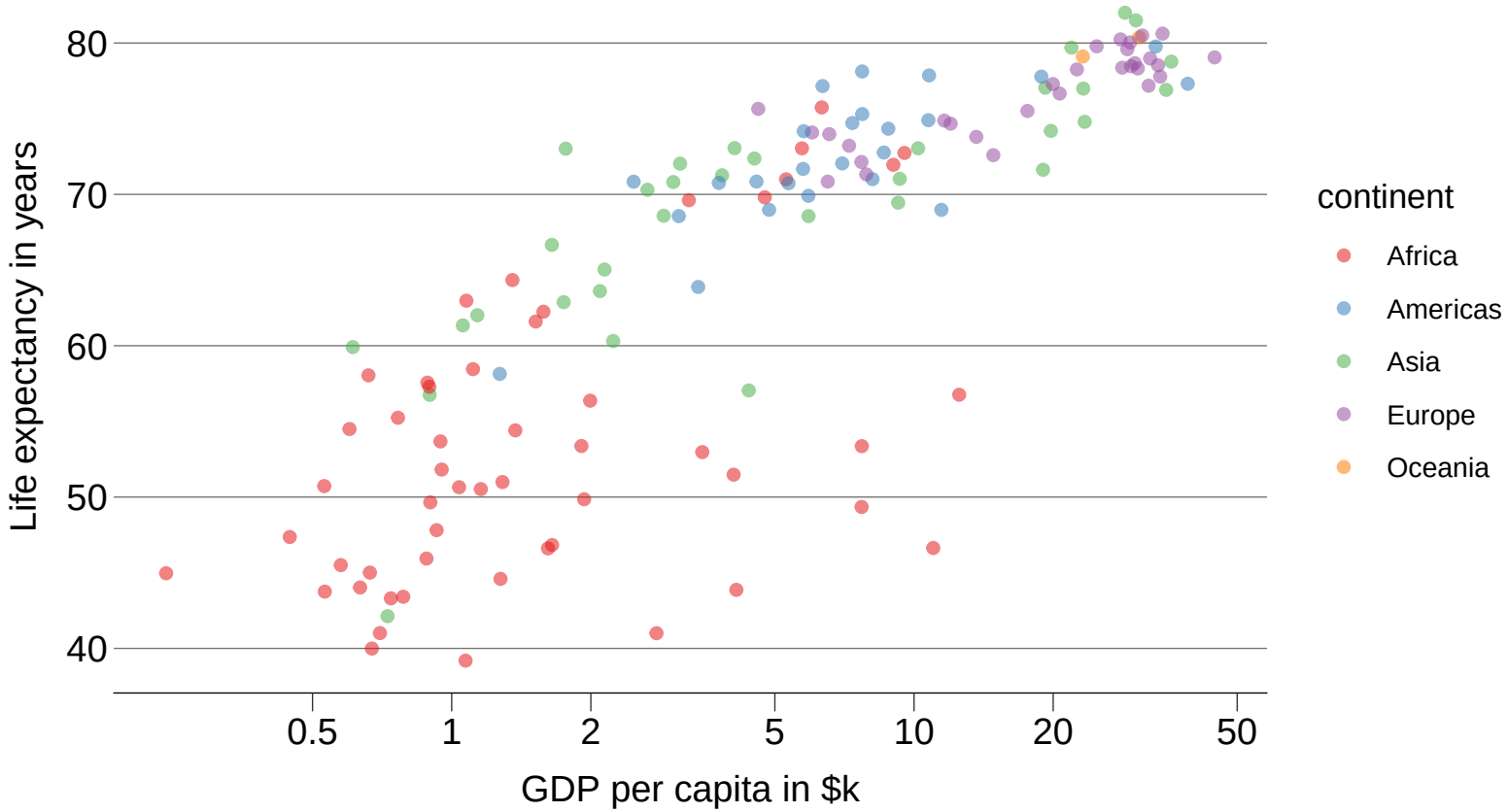
ggplot2 allows for consistent styling to be apply to plots through themes



I've applied my own style sheet here, based on the typical "look" of my own graphs:



I've applied my own style sheet here, based on the typical "look" of my own graphs: white background; horizontal grid; consistent, well-spaced axis text; thinner scaffolding



We can apply the style sheet to any plot, though I've written it to suit a wide scatterplot

ggplot2 themes rely on a special set of commands...

"Golden scatterplot" theme — ggplot2 code (1/3)

Henceforth, all plots in this example will apply this custom theme:

```
## My ggplot theme for a "golden scatterplot"
## Christopher Adolph (with help from Brian Leung)
## faculty.washington.edu/cadolph
goldenScatterCAtheme <- theme(
   ## Removes main plot gray background
   panel.background = element_rect(fill = "white"),

   ## Golden rectangle plotting area (leave out for square)
   aspect.ratio = ((1 + sqrt(5))/2)^(-1),

   ## All axes changes
   axis.ticks.length = unit(0.5, "char"),  # longer ticks
```

"Golden scatterplot" theme — ggplot2 code (2/3)

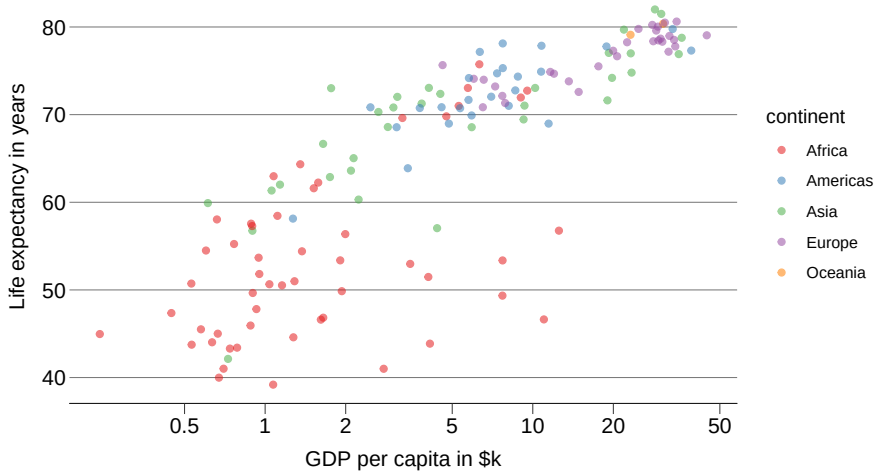
```
## Horizontal axis changes
axis.line.x.top = element_line(size = 0.2),    # thinner axis lines
axis.line.x.bottom = element_line(size = 0.2), # thinner axis lines
axis.ticks.x = element_line(size = 0.2),       # thinner ticks
axis.text.x = element_text(color = "black", size = 12),
              ## match type of axis labels and titles
axis.title.x = element_text(size = 12,
                margin = margin(t = 7.5, r = 0, b = 0, l = 0)),
                ## match type; pad space between title and labels

## Vertical axis changes
axis.ticks.y = element_blank(), # no y axis ticks (gridlines suffice)
axis.text.y = element_text(color = "black", size = 12,
               margin = margin(t = 0, r = -4, b = 0, l = 0)),
               ## match type of axis labels and titles, pad
axis.title.y = element_text(size = 12,
               margin = margin(t = 0, r = 7.5, b = 0, l = 0)),
               ## match type of axis labels and titles, pad
```

"Golden scatterplot" theme ggplot2 code (3/3)

```
  ## Legend
  legend.key = element_rect(fill = NA, color = NA),
               ## Remove unhelpful gray background

  ## Gridlines (in this case, horizontal from left axis only
  panel.grid.major.x = element_blank(),
  panel.grid.major.y = element_line(color = "gray45", size = 0.2),

  ## Faceting (small multiples)
  strip.background = element_blank(),
               ## Remove unhelpful trellis-like shading of titles
  strip.text.x = element_text(size=12),  # Larger facet titles
  strip.text.y = element_blank(),        # No titles for rows of facets
  strip.placement = "outside",           # Place titles outside plot
  panel.spacing.x = unit(1.25, "lines"), # Horizontal space b/w plots
  panel.spacing.y = unit(1, "lines")     # Vertical space b/w plots
)
class(goldenScatterCAtheme)  ## Create this as a class
```

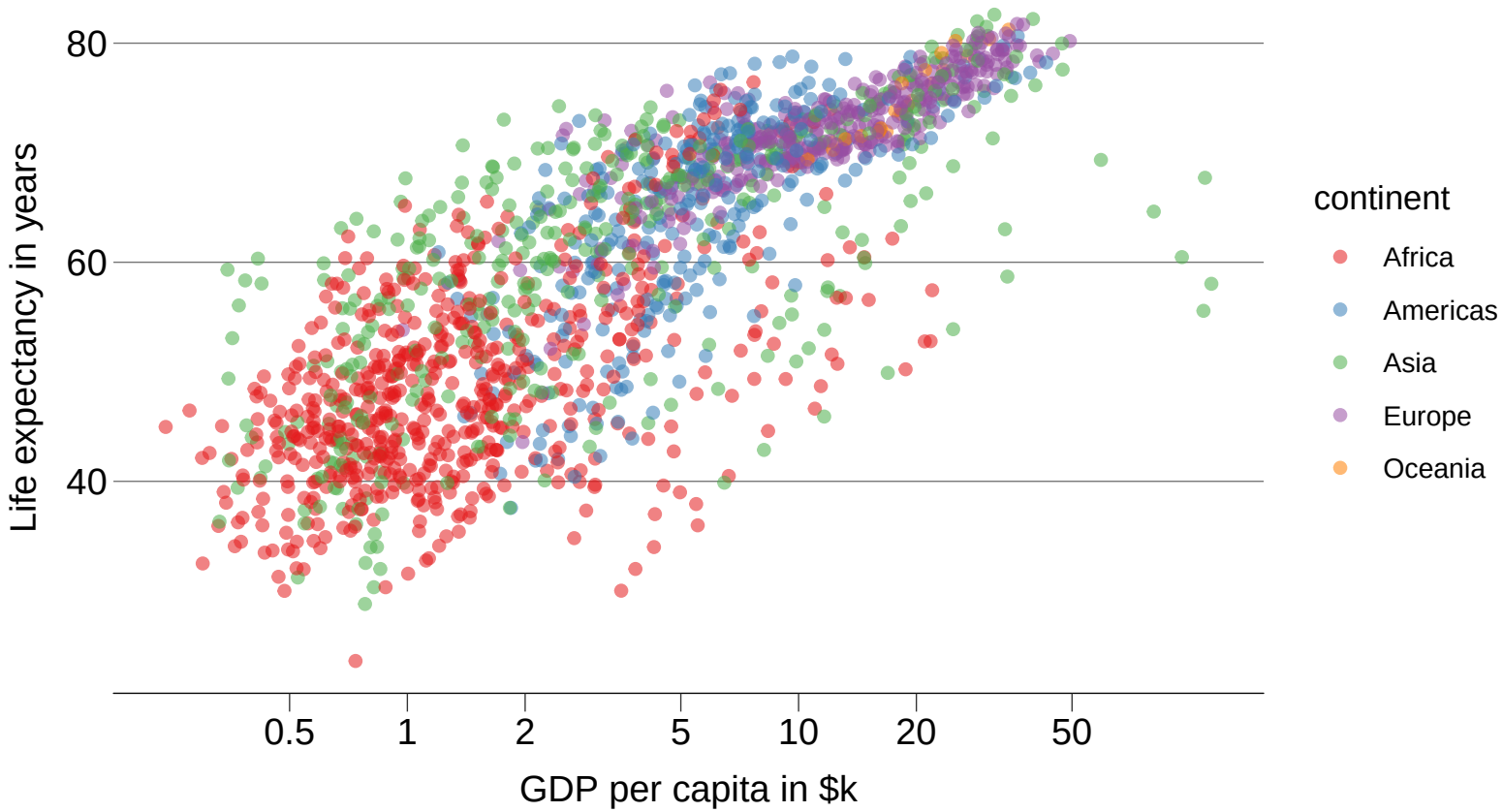
Complete graphics code:

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))


p <- p + goldenScatterCAtheme +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")

width <- 7
ggsave("ggScatterEx4_1.pdf",
       width=width,
       height=width/1.618)
```



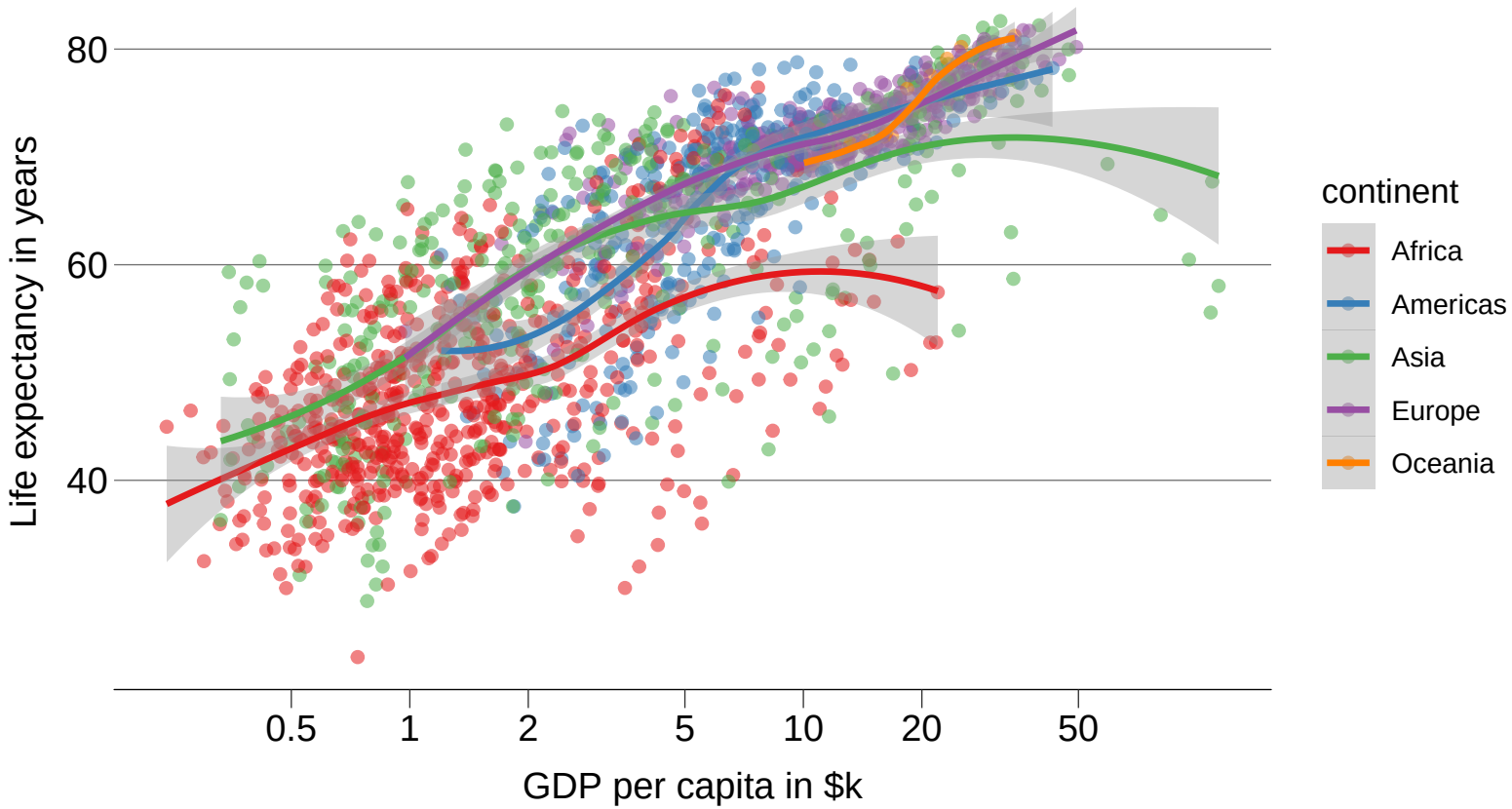
Adding a theme is simple. Probably best to add it before other elements

Could always go back and revise (and re-run) the theme code to make stylistic changes

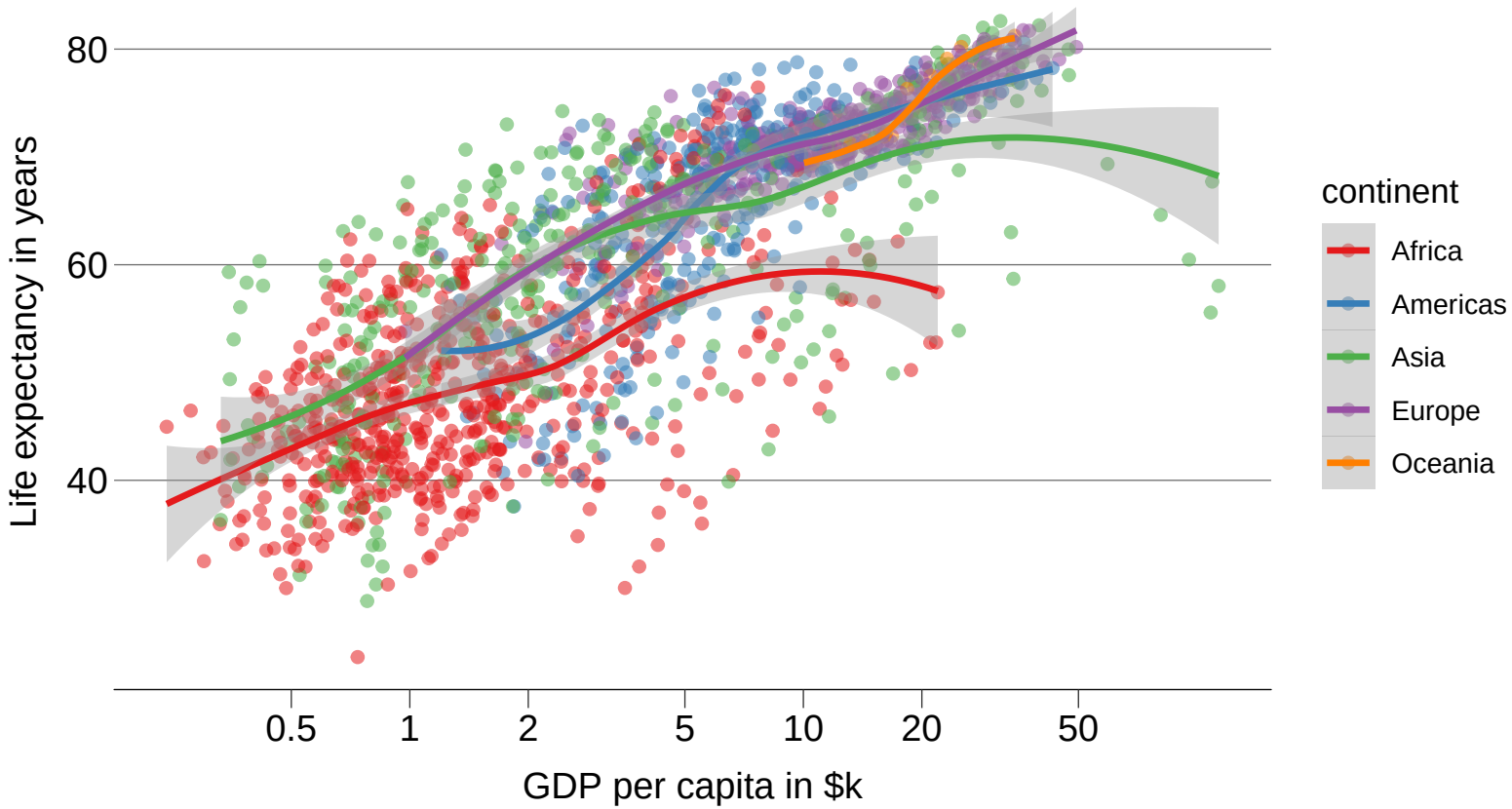


Let's return to the all-years Gapminder data one last time

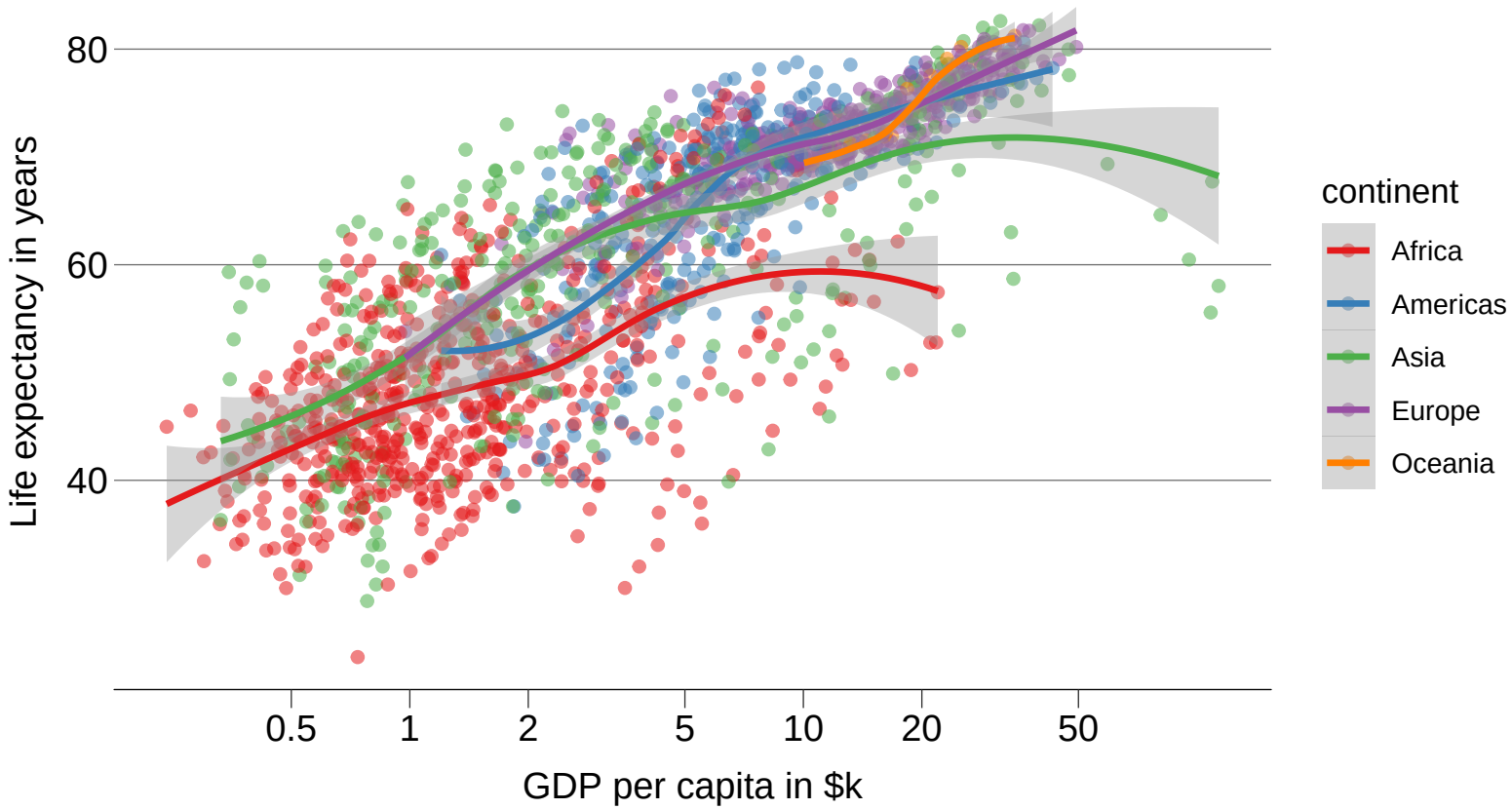
What would happen if we fit a regression line to this entire panel dataset?



Because our plot relies on a mapping to three variables,
ggplot will create separate fits between *x* and *y* for each region (color)



Is this an effective plot? Why or why not?

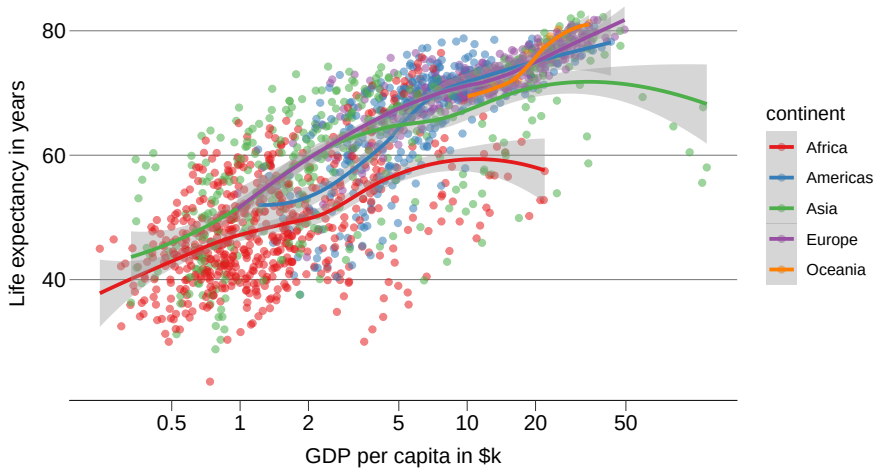


Is this an effective plot? Why or why not?

Garish, unclear, and only works if goal is to show similarity

Partial code (saving as in 4.1):

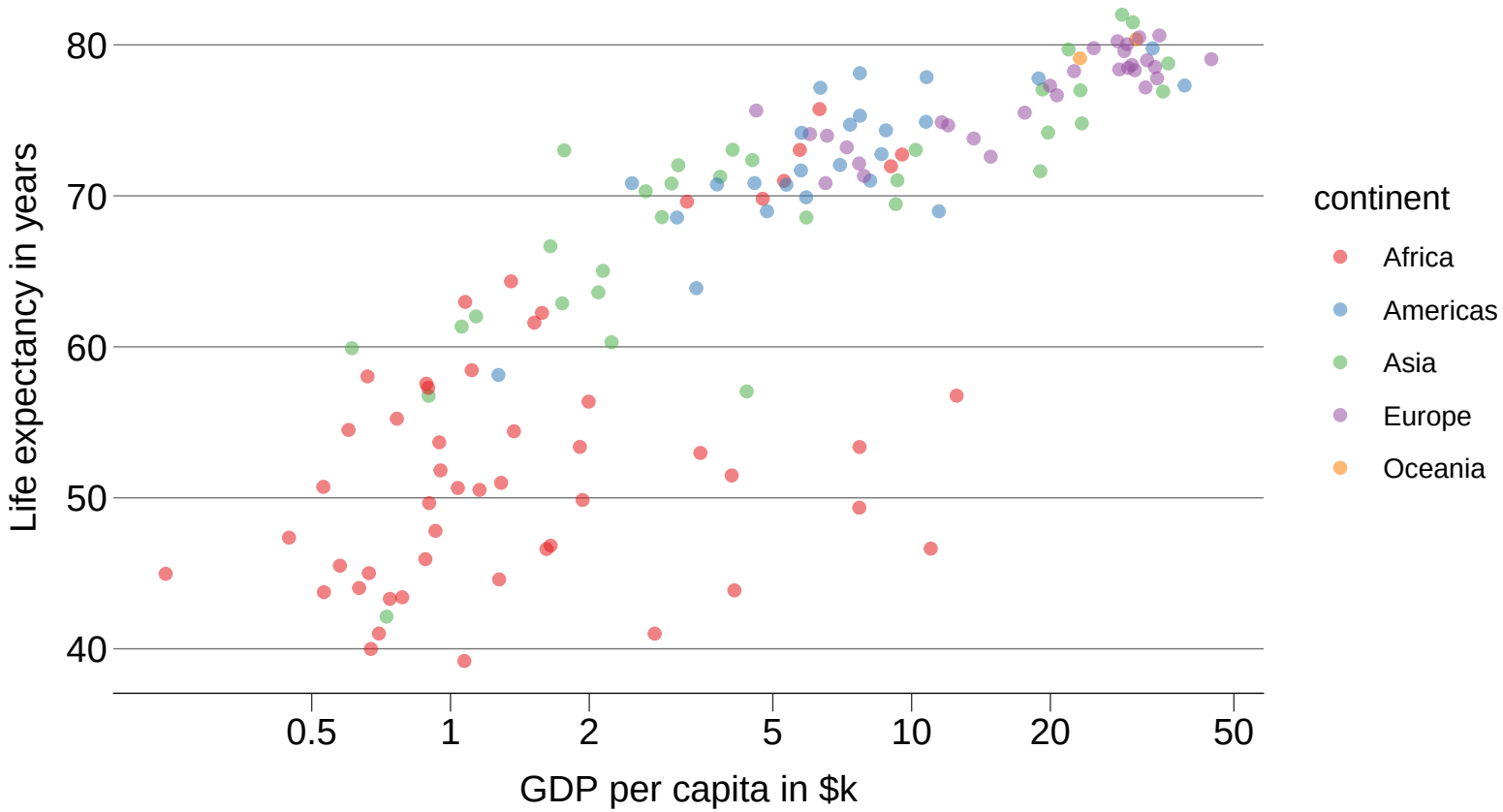
```
p <- ggplot(data=gapminder,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))

p <- p + goldenScatterCAtheme +
     geom_point(alpha=0.55) +
     geom_smooth() +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



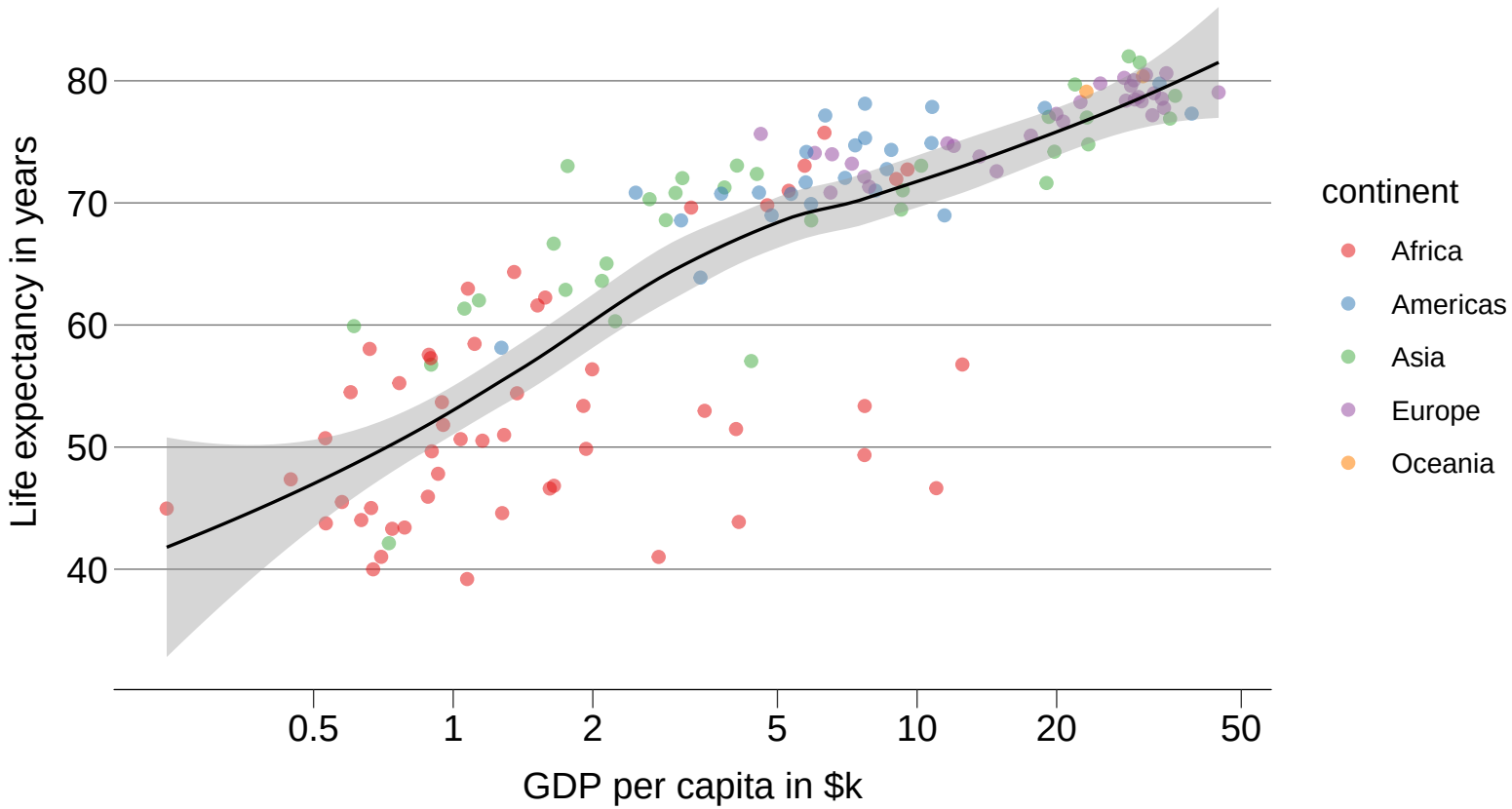
geom_smooth *fits a flexible regression model by default*

Regression lines overlap in order of continent

How hard this would be to parse with a mid-gray background?



Let's return to the 2002 Gapminder data,
rather than try to pool all years into one plot

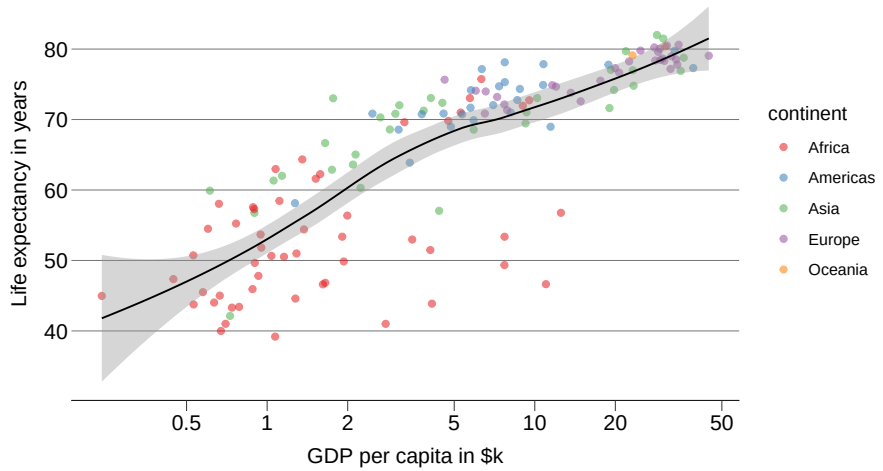


How do we get ggplot to pool all this data into a single regression model

...without deleting the color coding of points by region?

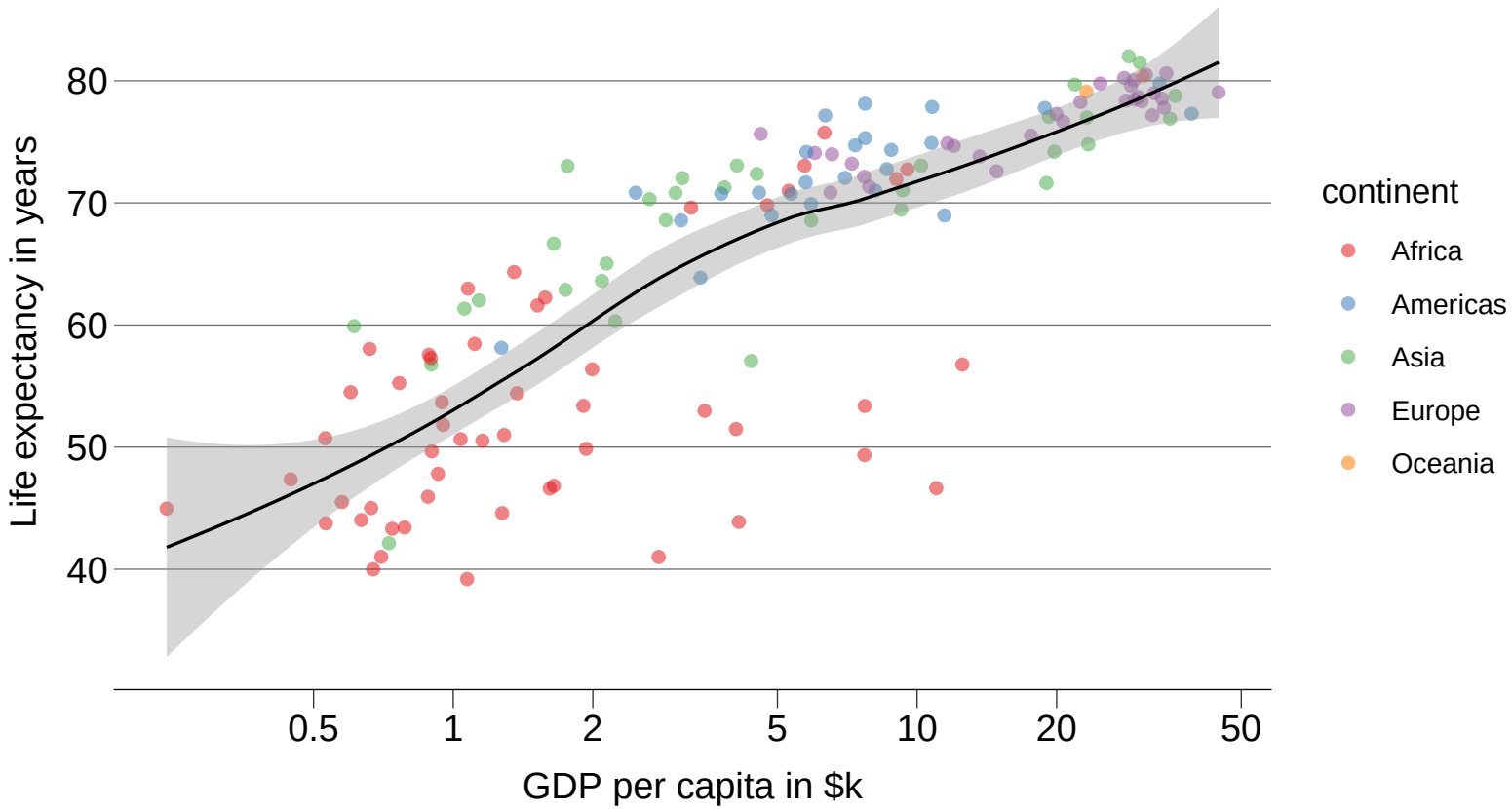
Partial code (saving as in 4.1):

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))


p <- p + goldenScatterCAtheme +
     geom_point(alpha=0.55) +
     geom_smooth(size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



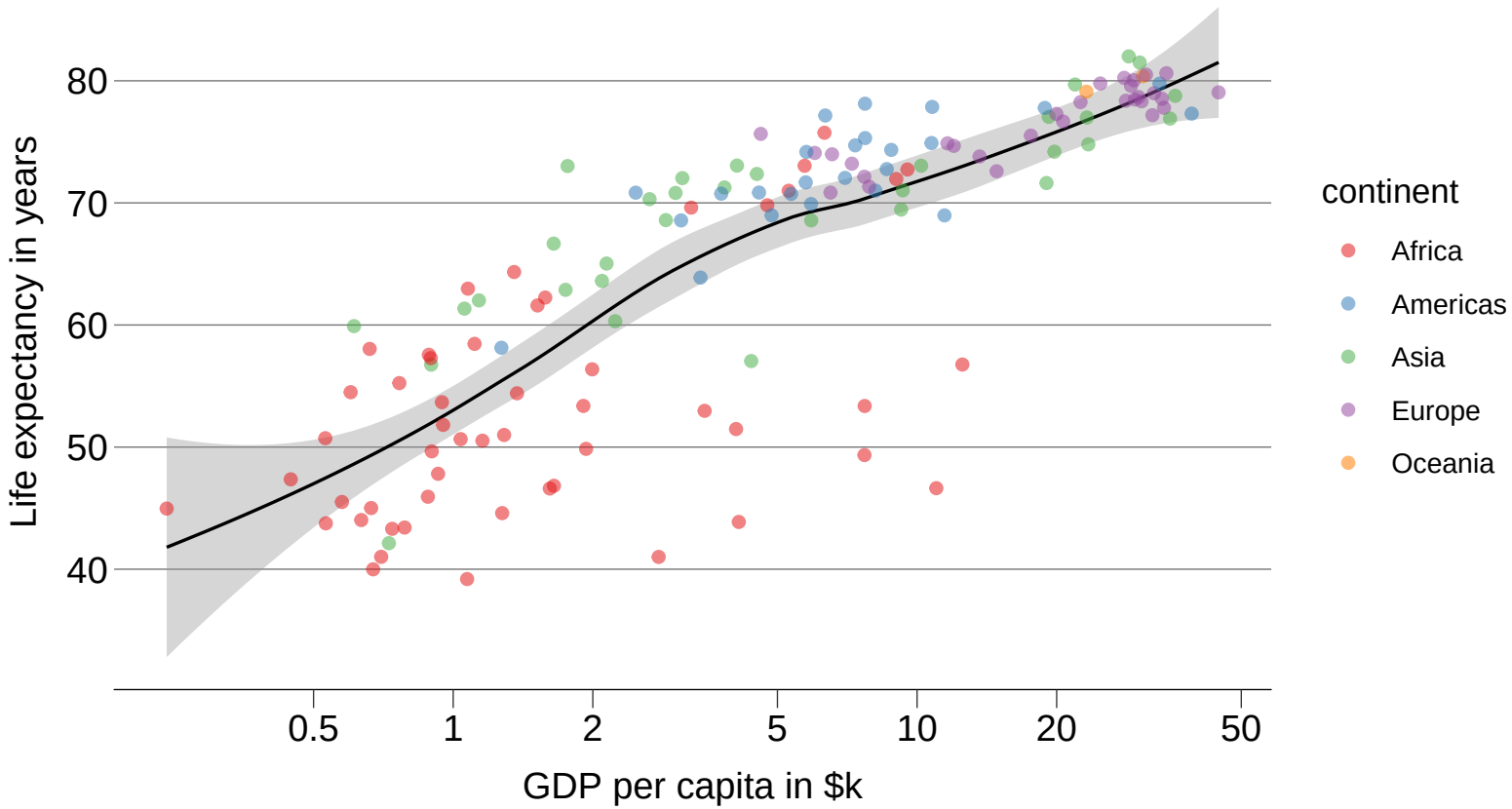
Quite tricky! We need to set an aes() *just for* geom_smooth *to override the usual data mapping*

We also make the line thinner to improve legibility, and make it black, to imply global scope



Subtle details: adding the smoother expanded the *y*-axis range

And the smoother was drawn *on top of the points*, rather than behind them

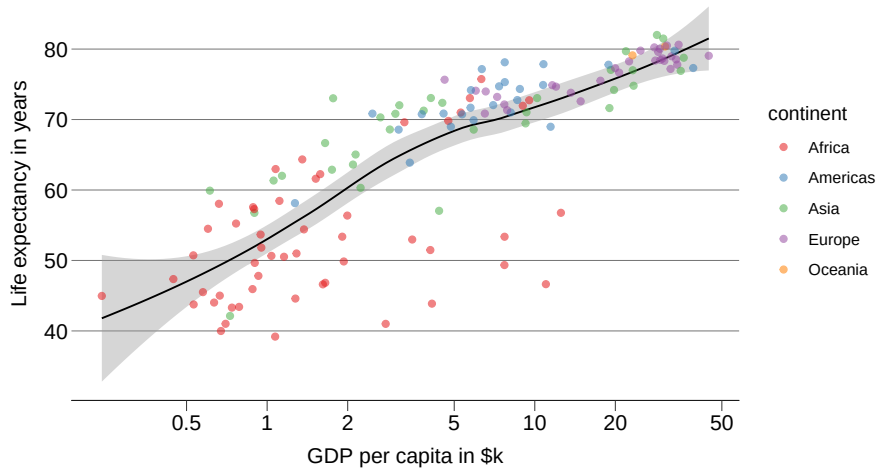


Always better to layer polygons like this confidence interval under points & text

ggplot2 won't layer automatically: it plots in the order geom's are added

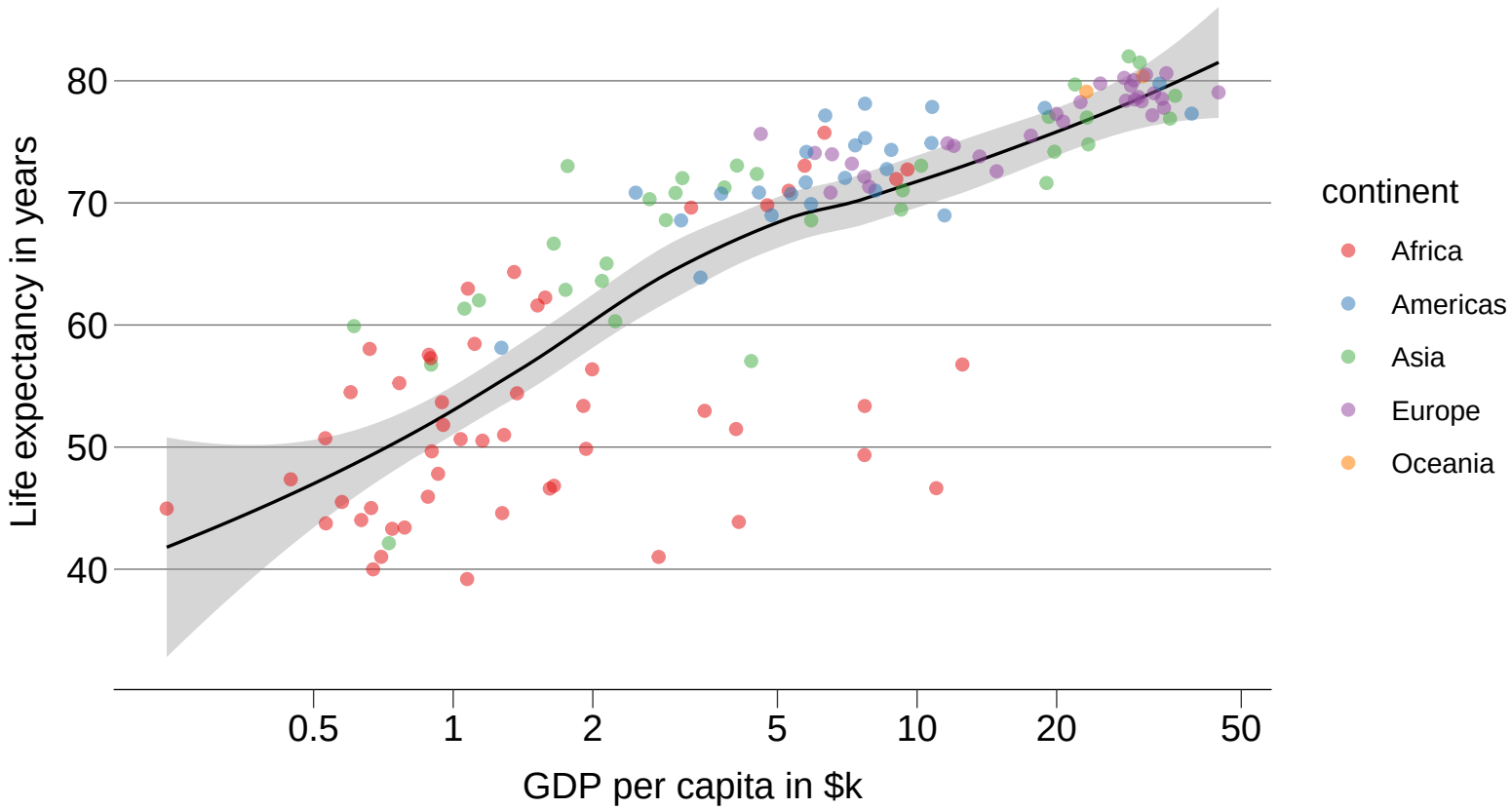
Partial code (saving as in 4.1):

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp,
                        color=continent))


p <- p + goldenScatterCAtheme +
     geom_smooth(size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



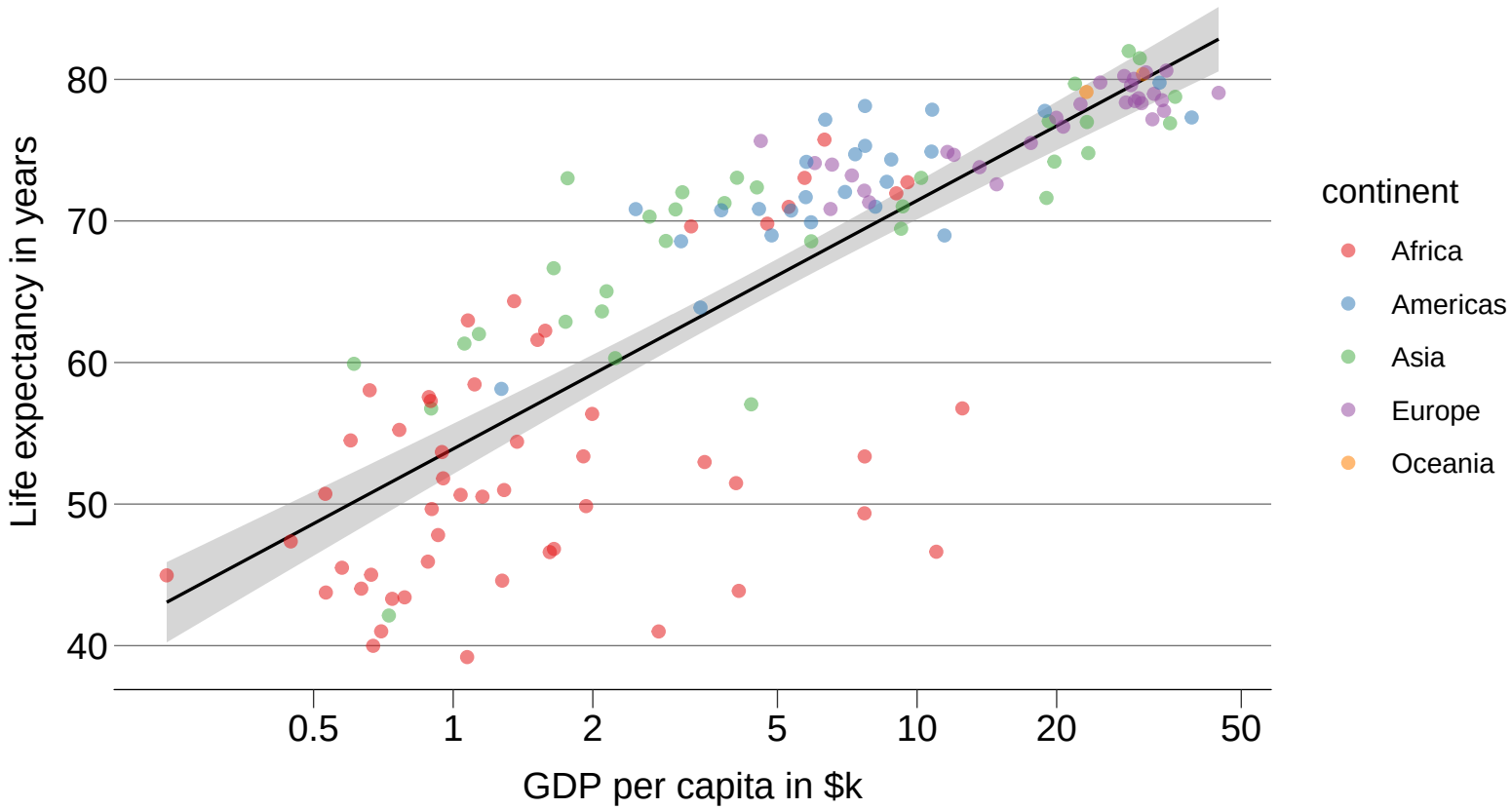
To effectively layer graphical elements, add in geom*'s that make polygons or other background elements, such as reference lines, first*

This is usually the reverse order that we draft code originally



Non-parametric models are great for learning if parametric models are appropriate

More on non-parametric smooths next week

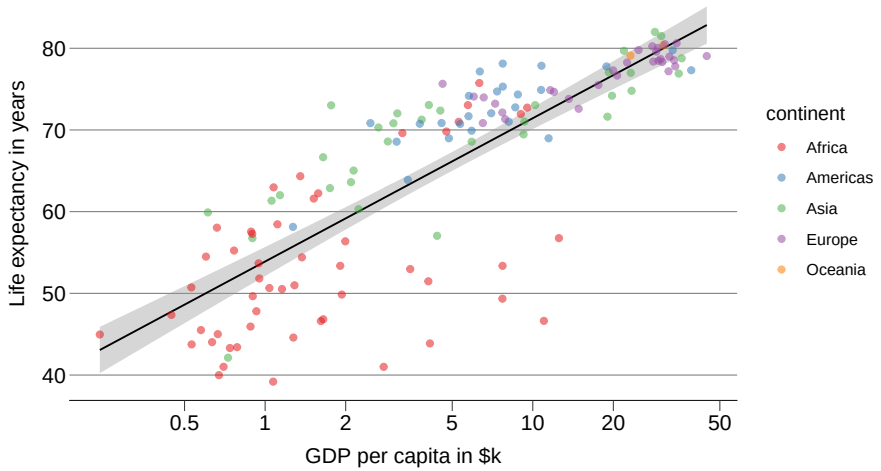


When a linear fit is appropriate, it makes the graph considerably simpler

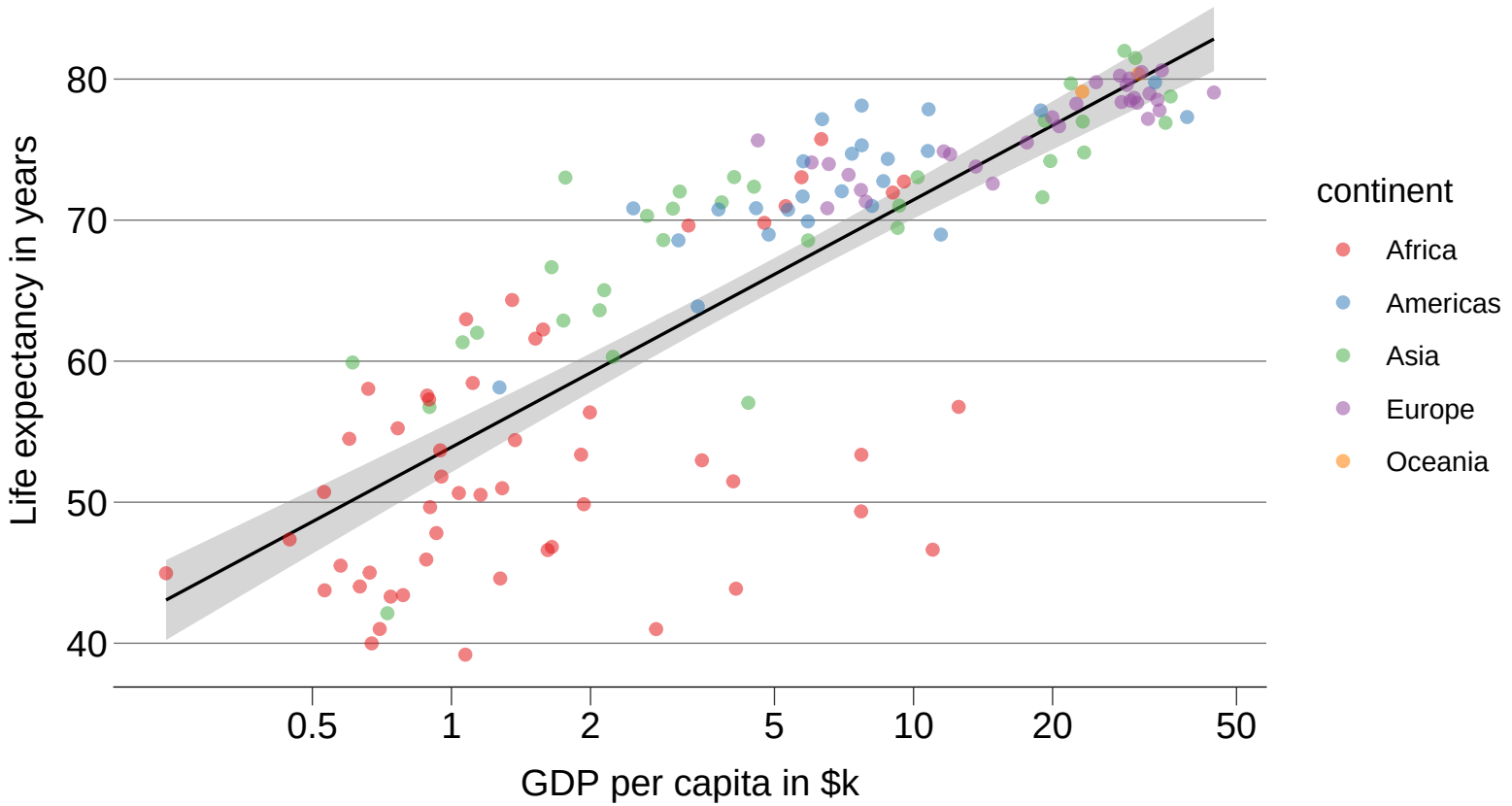
In this case, it's debatable, but worth exploring (and sharing code)

Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=lm,
                 size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```

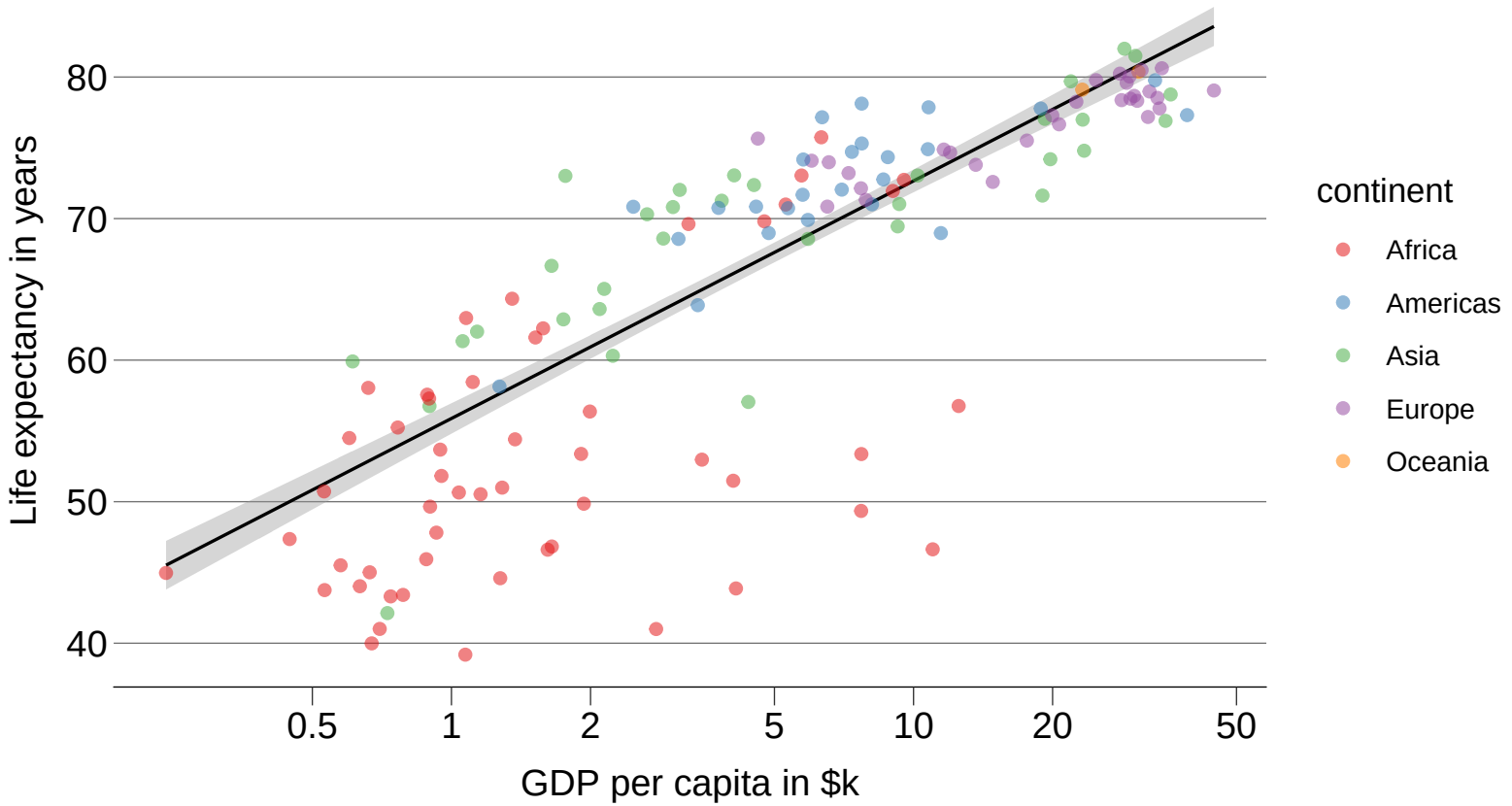


geom_smooth *needs to be told to do a linear fit*



When plots contain outliers – as this one does – it's better to use a robust & resistant fit

Aside: how does "resistant" differ from "robust"?

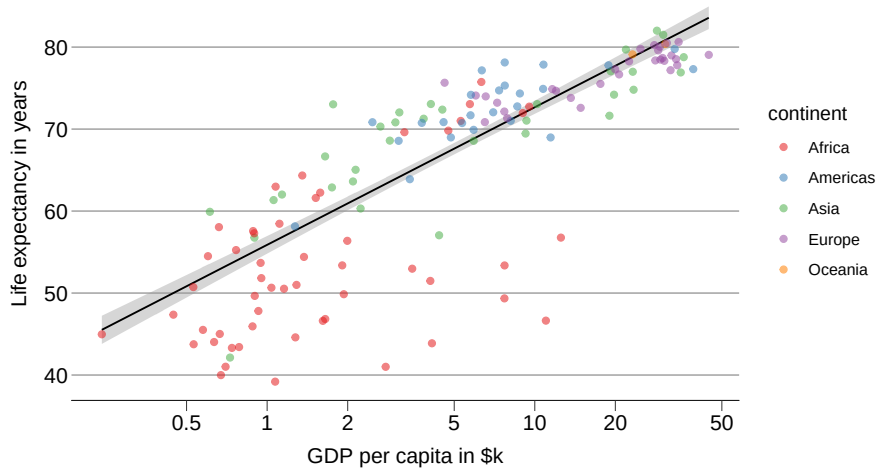


Note the difference a robust and resistant fit makes: closer to a "visual best fit"

Also highlights the presence of outliers – what could explain them?

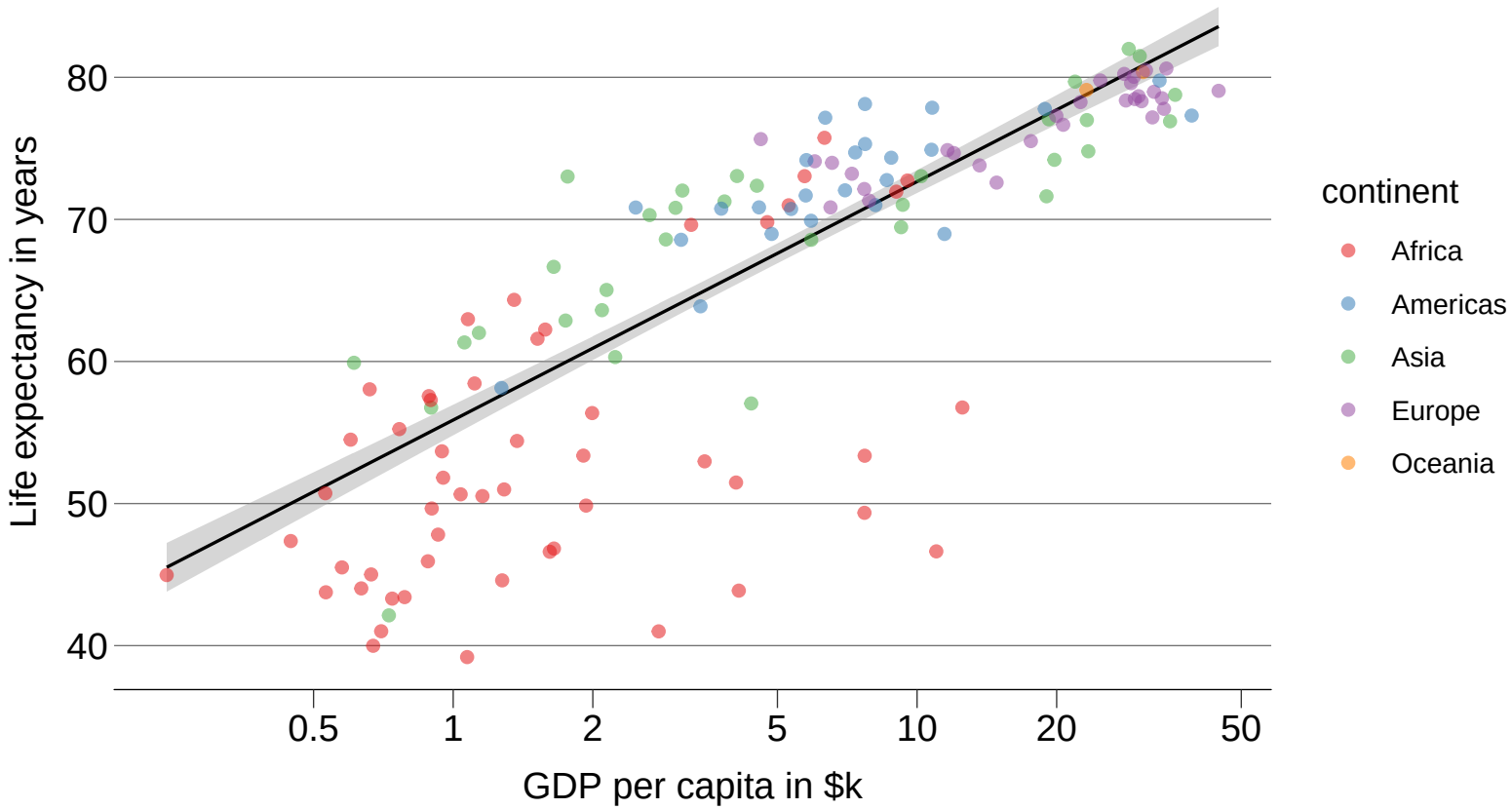
Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



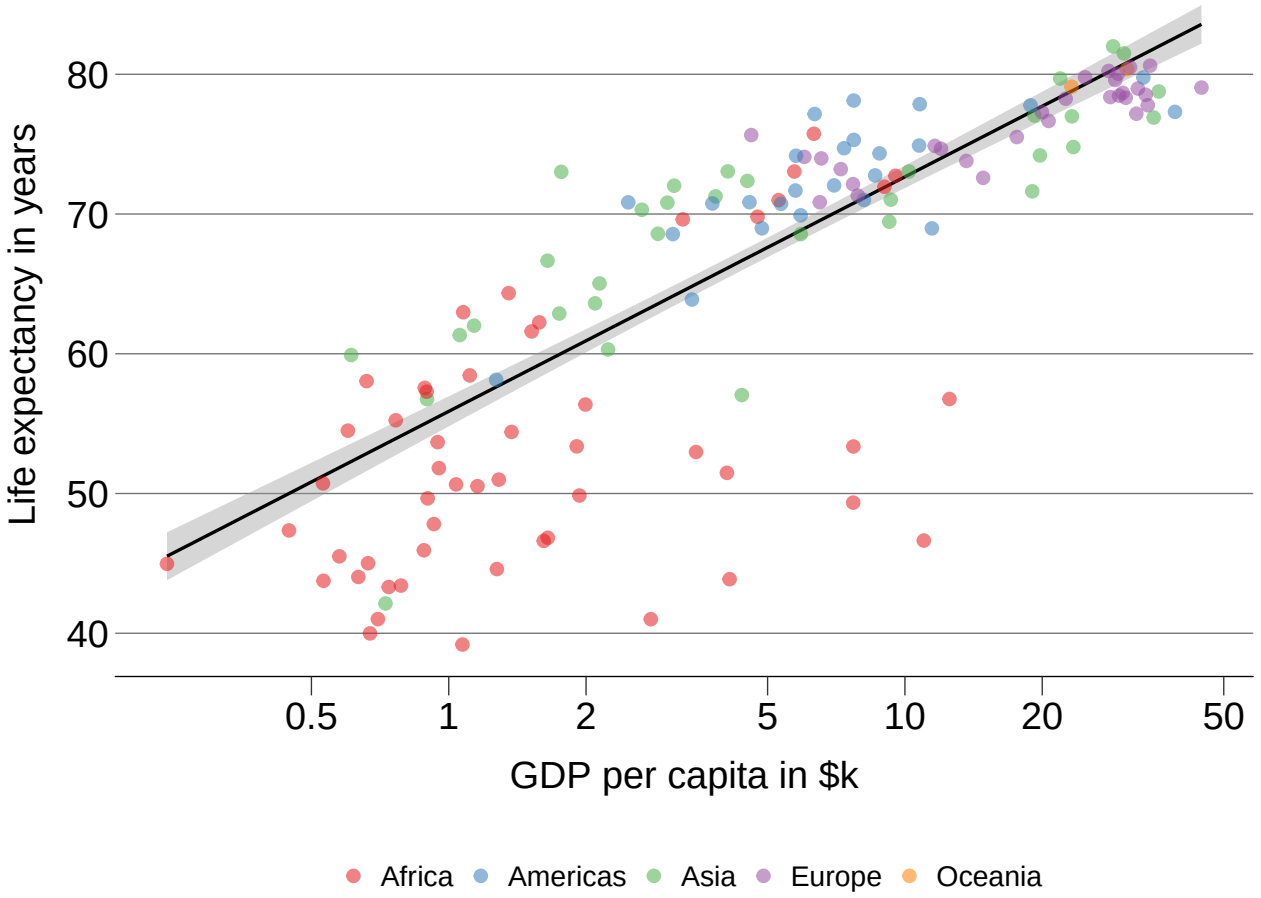
We can use a wide variety of methods to make fits in `geom_smooth`

Here, we need to pass an argument down to `rlm()` *to request the robust and resistant fit*



The legend wastes a lot of space on the right – usually a bad spot for small legends

Let's move it to the bottom

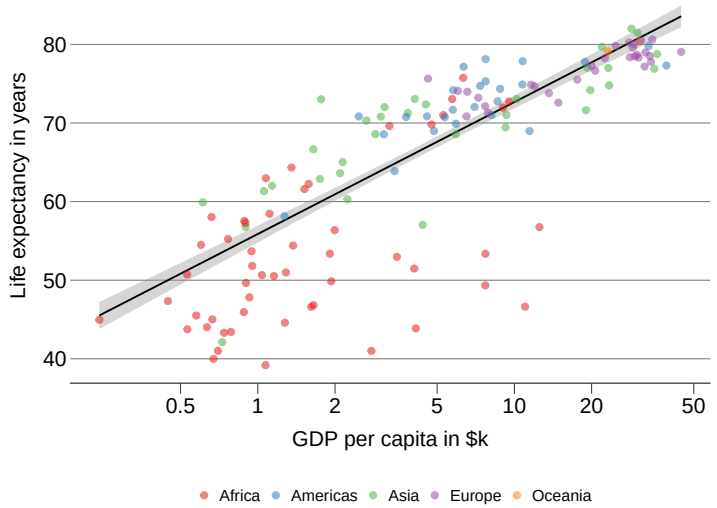


Let's move it to the bottom as a simple (but not perfect) fix to wasted space

We've also removed the legend title, which is unnecessary with self-explanatory groups

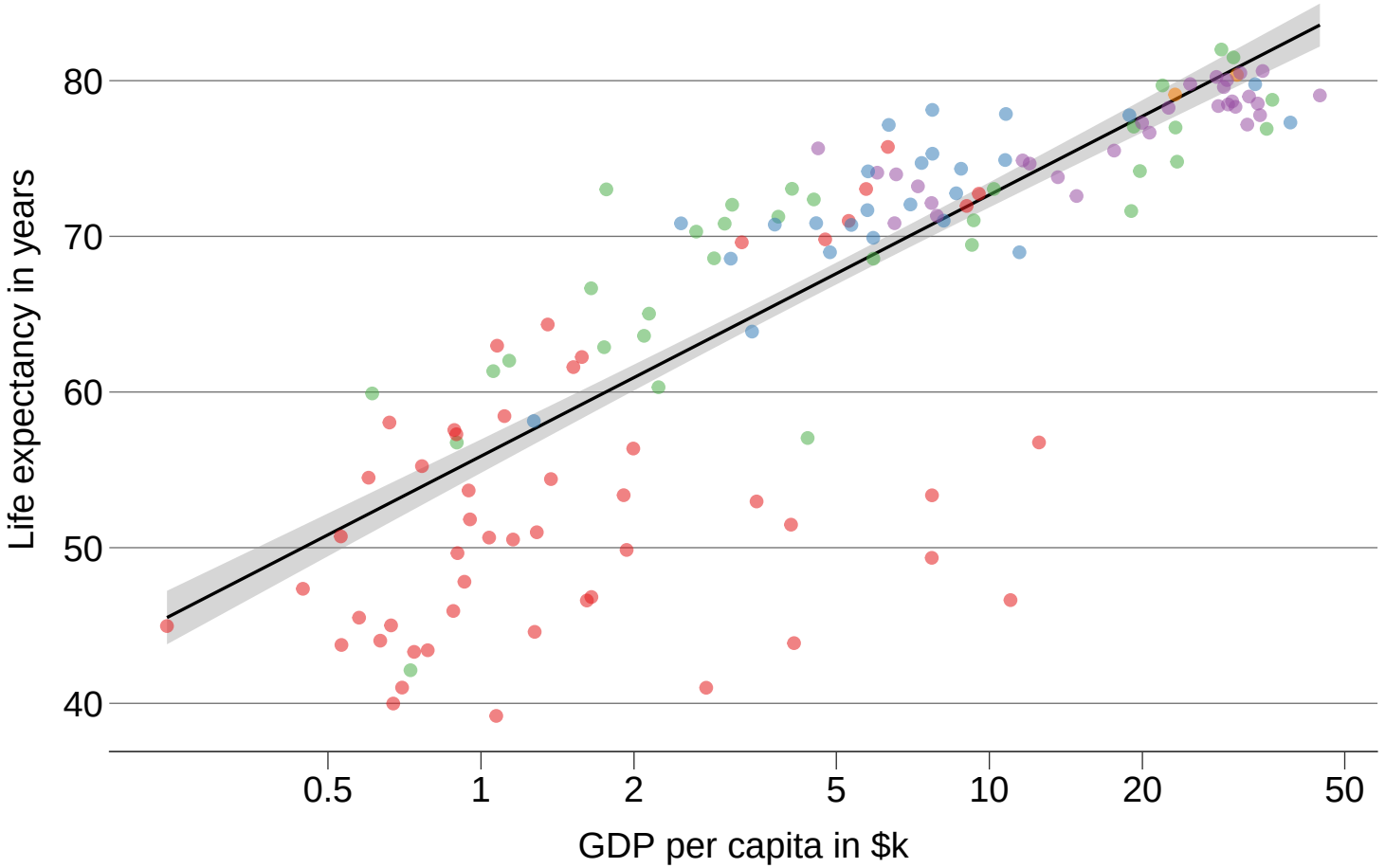
Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years") +
     theme(legend.title = element_blank(),
           legend.position = "bottom")
```



We can add case-specific theme modifications through the `theme` *element*

Note that `element_blank()` ***removes*** *an element*

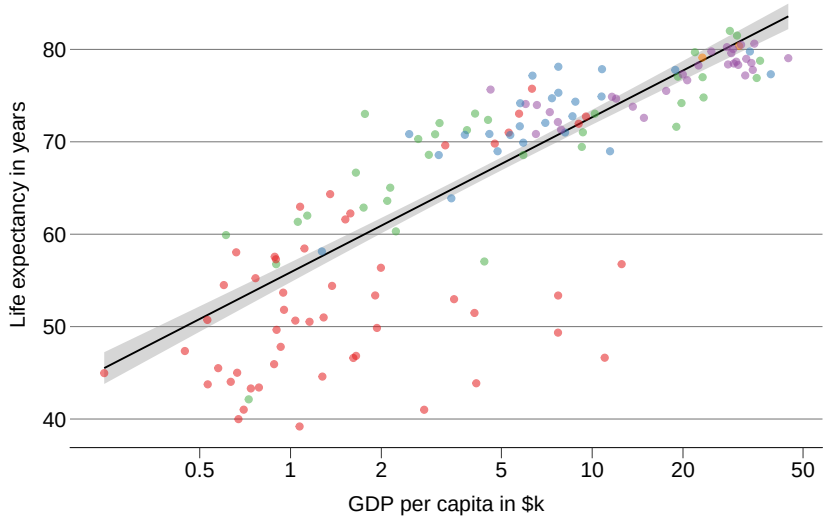


What if we wanted to remove the legend entirely?

Often we do – it's usually better to label things directly!

Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years") +
     theme(legend.position = "none")
```

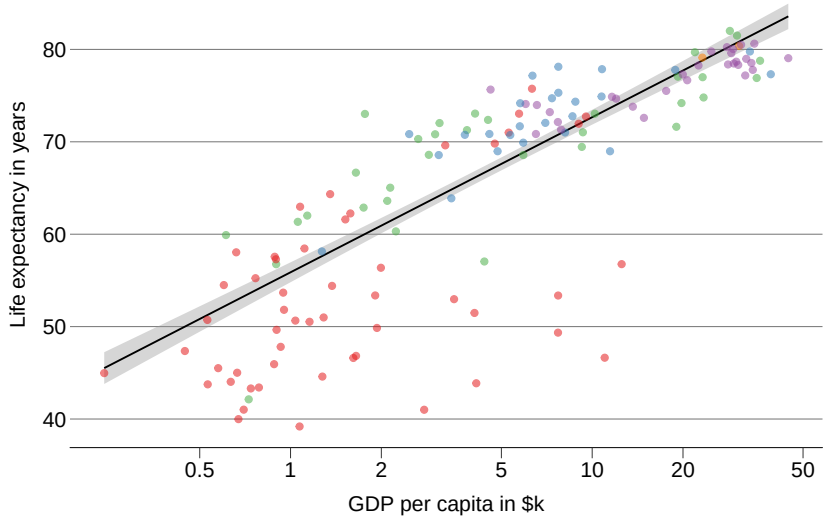


Two ways to do it

Method 1: Using a theme argument

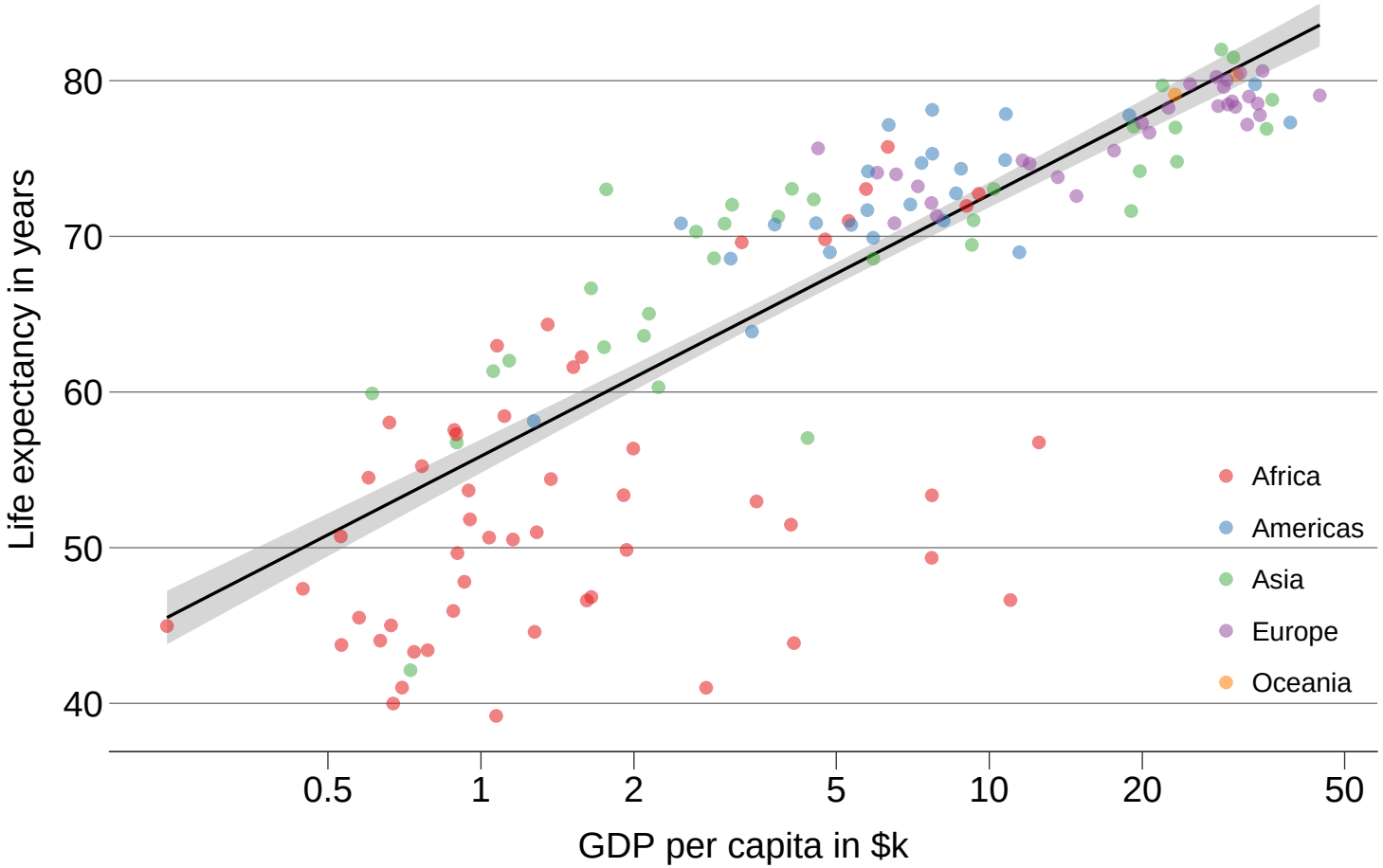
Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5,
                 inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1",
                        guide=FALSE) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years")
```



Two ways to do it

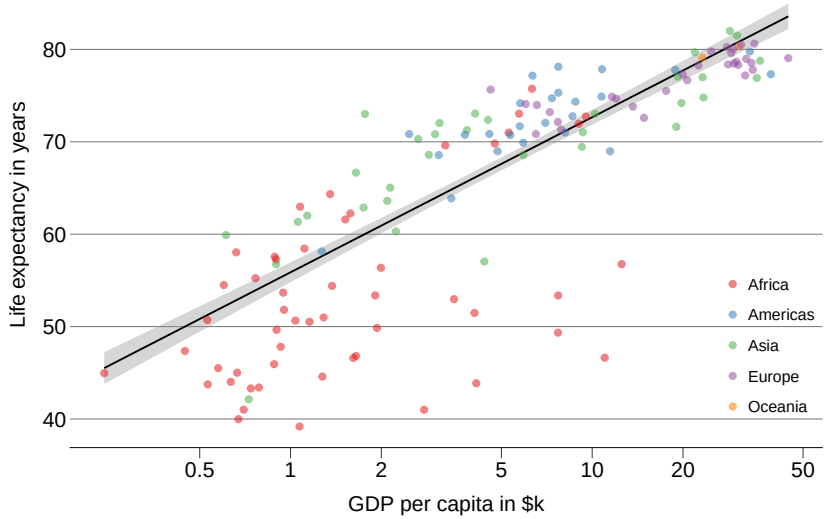
Method 2: Through the `guide` *input to the appropriate* `scale_` *argument*



Direct labeling isn't an option for these points,
so let's save space by moving the legend into the plot

Partial code (map & save as in 4.1):

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5,
                   inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years") +
     theme(legend.title = element_blank(),
           legend.position = c(0.93, 0.2),
           legend.background =
       element_rect(fill=alpha("white", 0)))
```

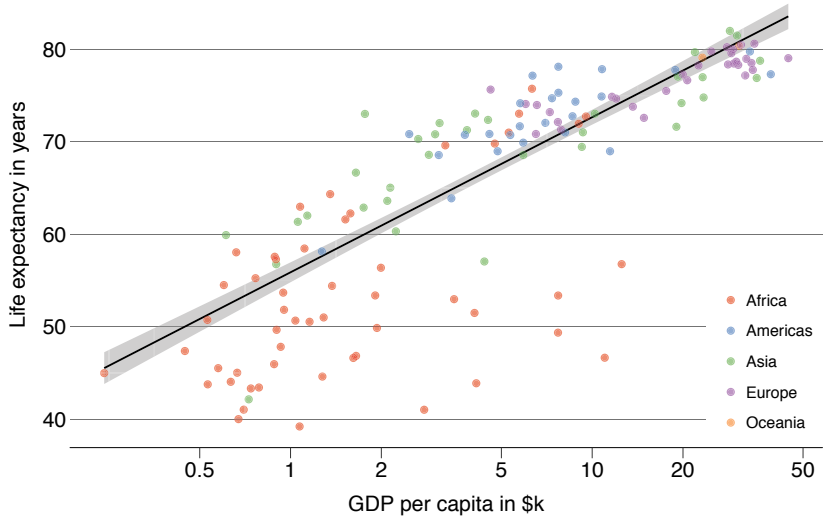


`legend.position()` *can take plot locations instead of plot sides*

You'll need to test several locations to avoid overlap of data and/or spilling out of the plot

Partial code (. . . indicates omitted code):

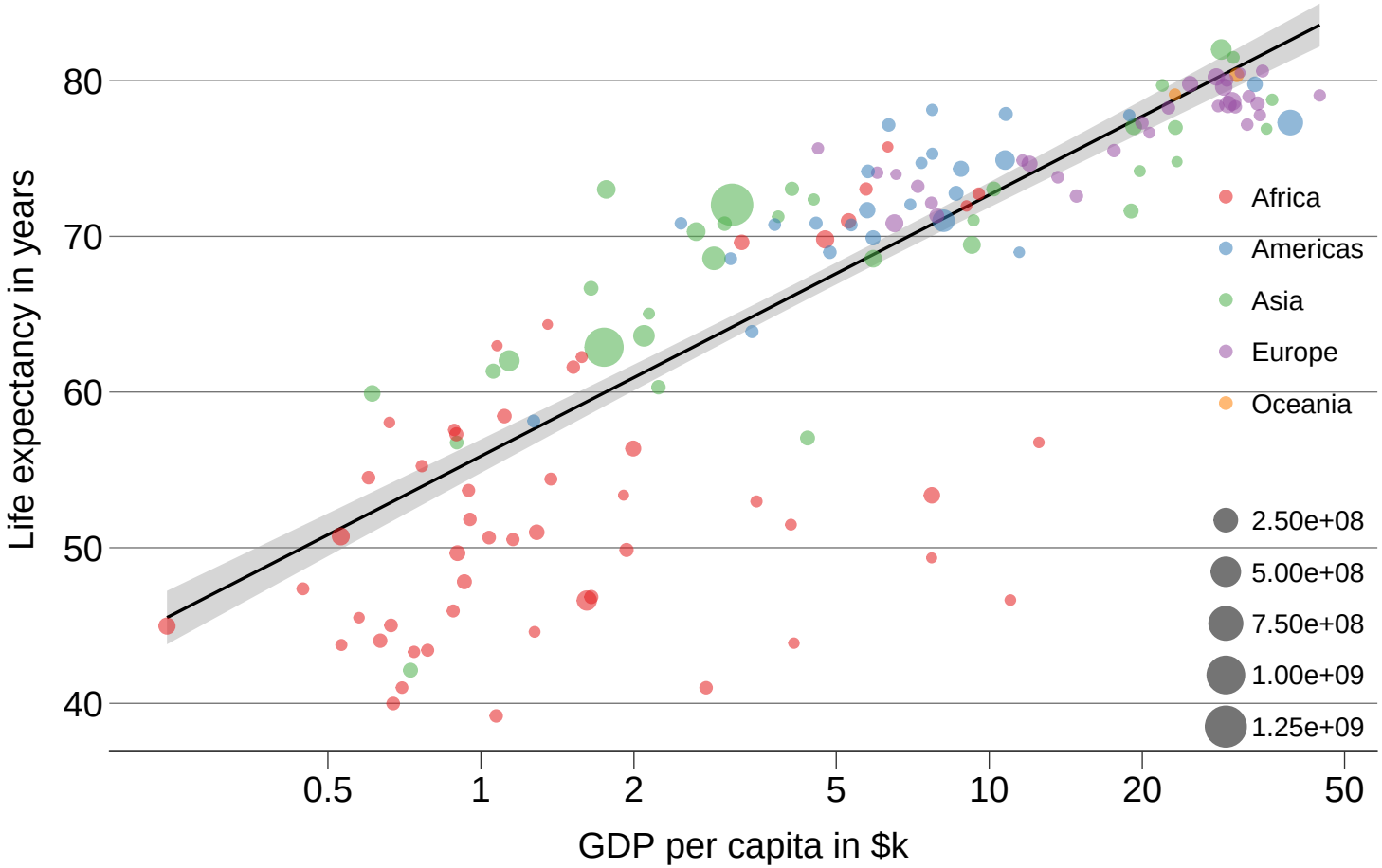
```
p <- p + goldenScatterCAtheme +
     ...
     labs(x="GDP per capita in $k",
          y="Life expectancy in years") +
     theme(legend.title = element_blank())

width <- 7
ggsave(filename="ggScatterEx6_3a.pdf",
       plot = reposition_legend(p, "bottom right"),
       width=width,
       height=width/1.618)
```



Alternative approach using

`lemon::reposition_legend()`

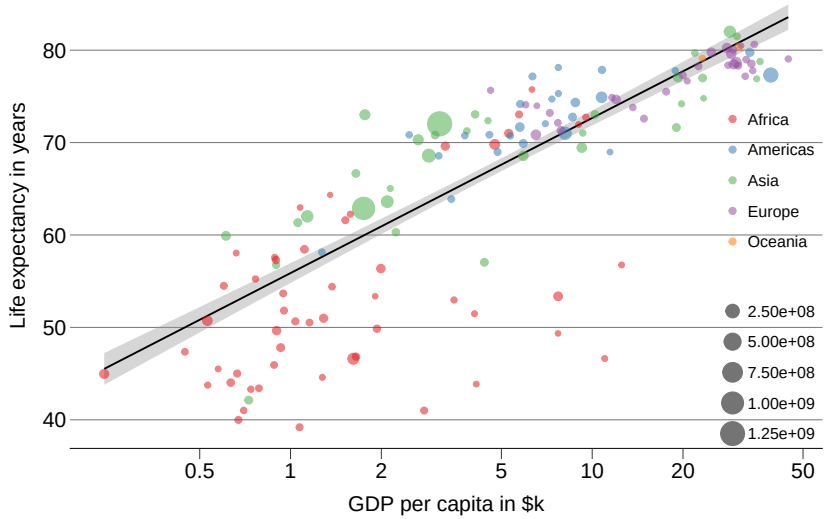
Seem to insert a blank page in the PDF



Let's make the plot more functional by adding a size dimension to the points

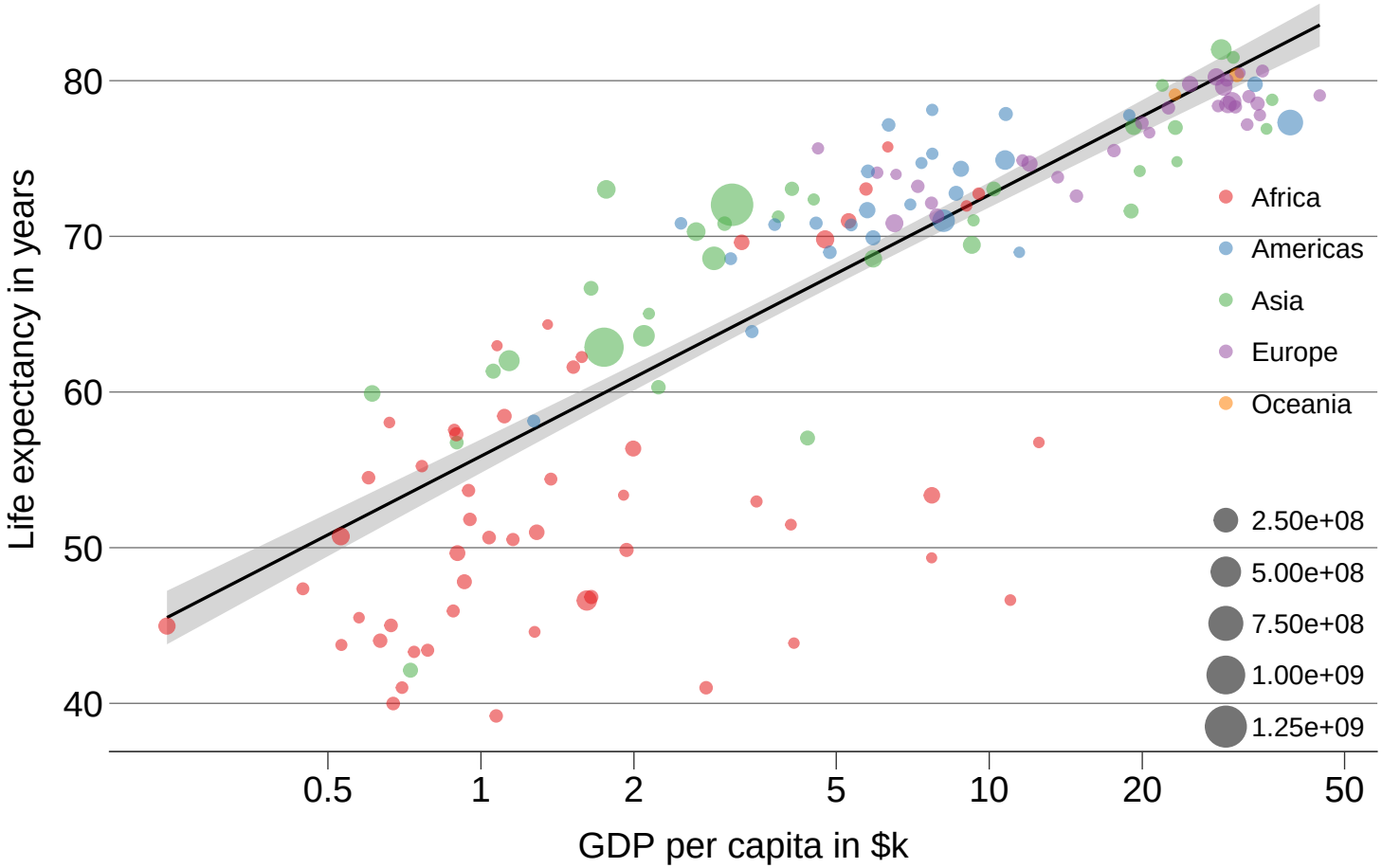
We now have a bubble plot where bubble size indicates population

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
             y=lifeExp, color=continent))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years") +
     theme(legend.title = element_blank(),
           legend.position = c(0.925, 0.37),
           legend.background =
        element_rect(fill=alpha("white", 0)))
```



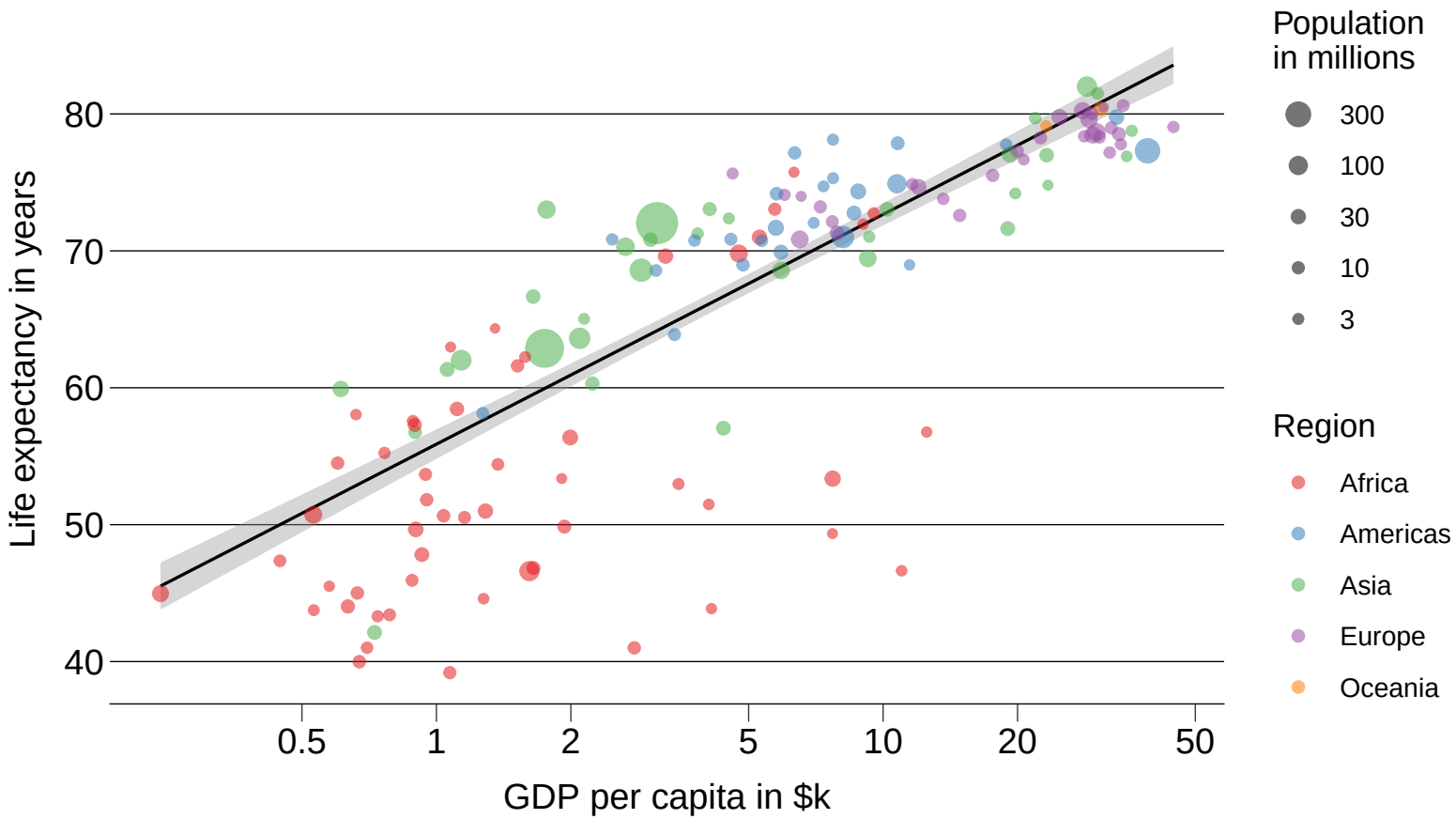
Bubble plots: allow `geom_point` *to have a new, specific aesthetic mapping of its size to the population variable in the dataset*

Default is (appropriately) to scale by area, not radius



Legend is now too crowded – maybe split it into each corner; hard to do in ggplot

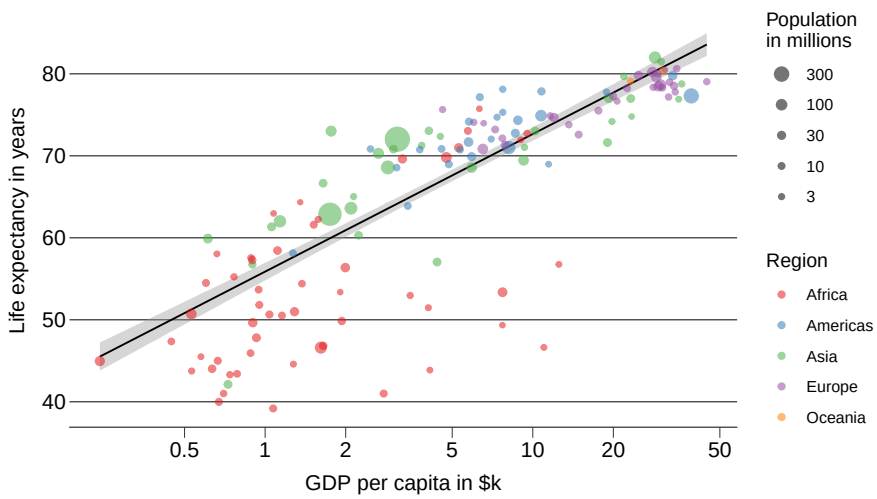
Legend is badly labelled – *the most common issue I see in student* ggplots



Sensible labels: readable legend titles, custom (and clear) numerical categories

Don't leave unnecessary scientific notation or odd numerical cutoffs in plots!

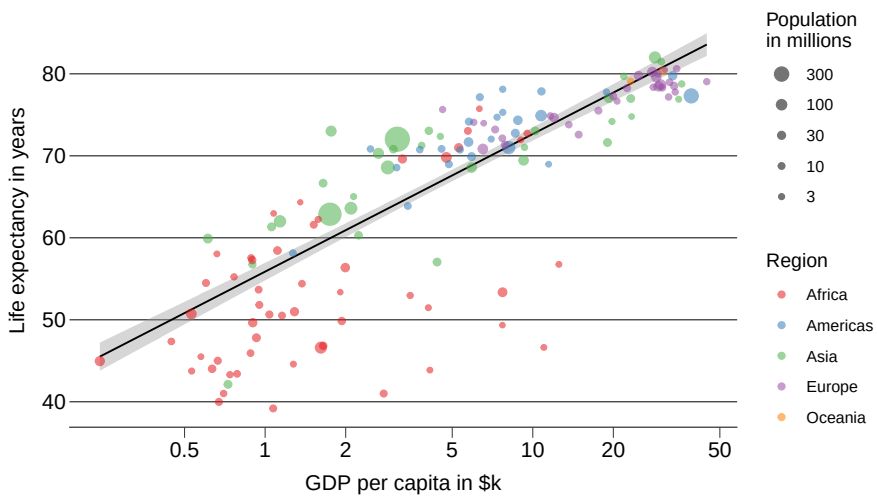
```
popBreaks <- rev(c(3, 10, 30, 100, 300)*1000000)

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                    labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions")
```



Several tricky bits:

1. `scale_size` *used to set custom population legend breaks and labels – separately, because the data are in units and the labels in millions*

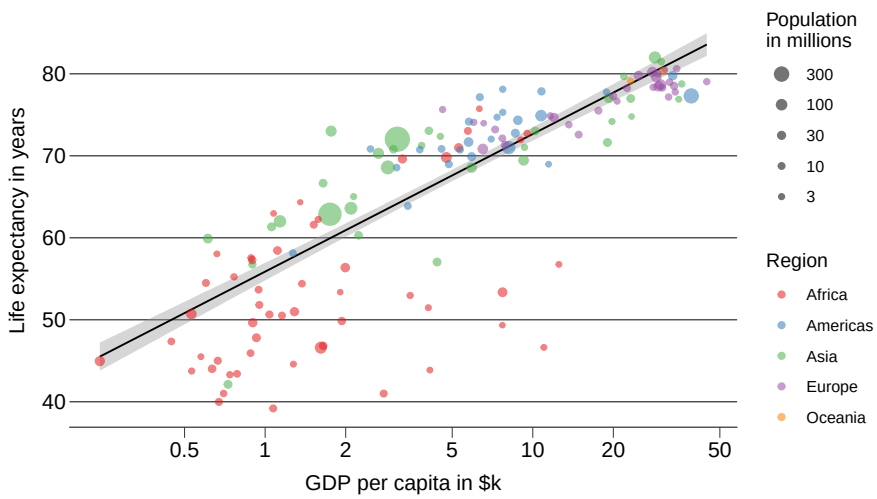
```
popBreaks <- rev(c(3, 10, 30, 100, 300)*1000000)

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                    labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions")
```



Several tricky bits:

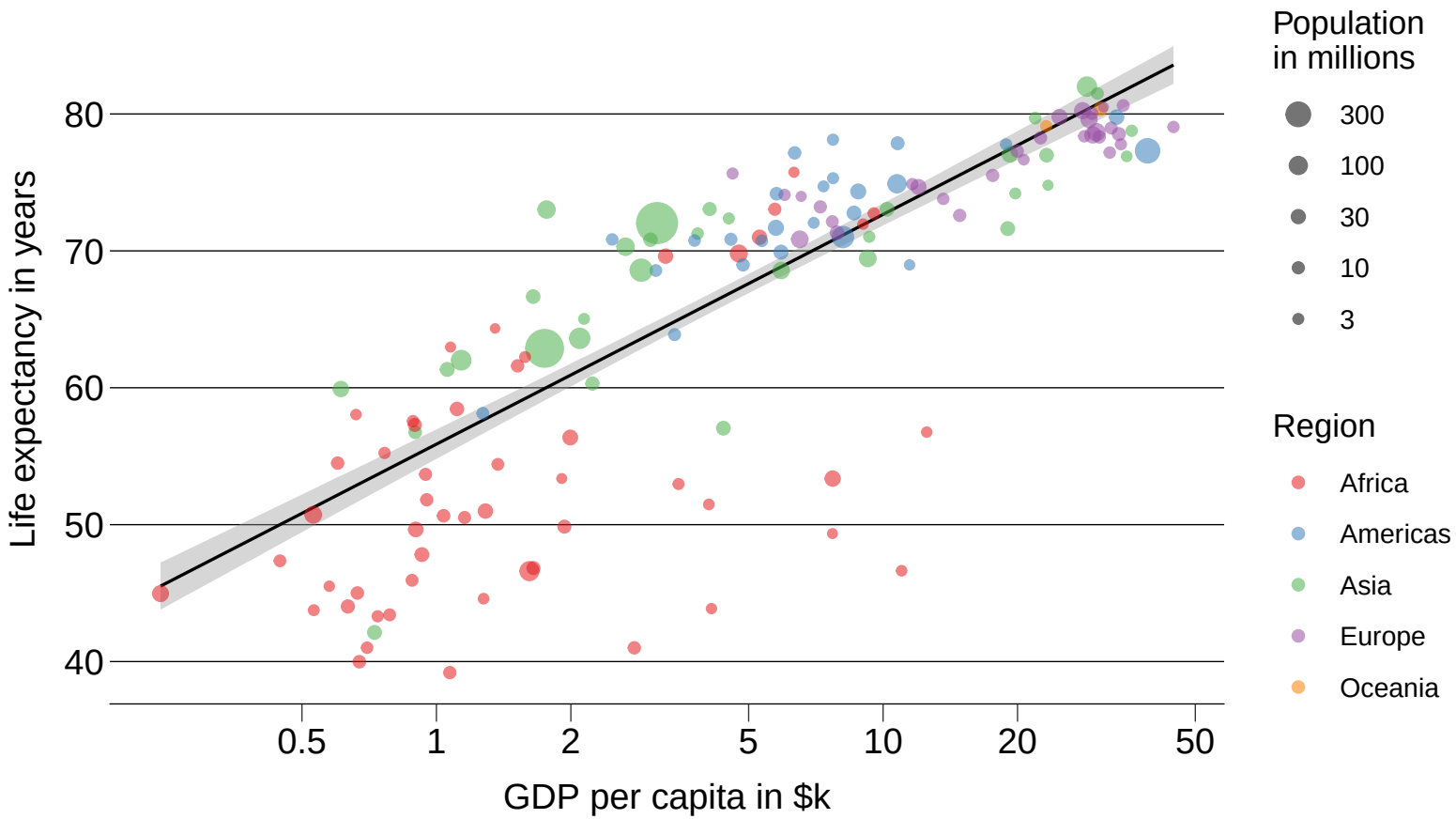
2. I've made my own breaks as an object, and reversed them to reverse the legend order

```
popBreaks <- rev(c(3, 10, 30, 100, 300)*1000000)

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                    labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions")
```



Several tricky bits:

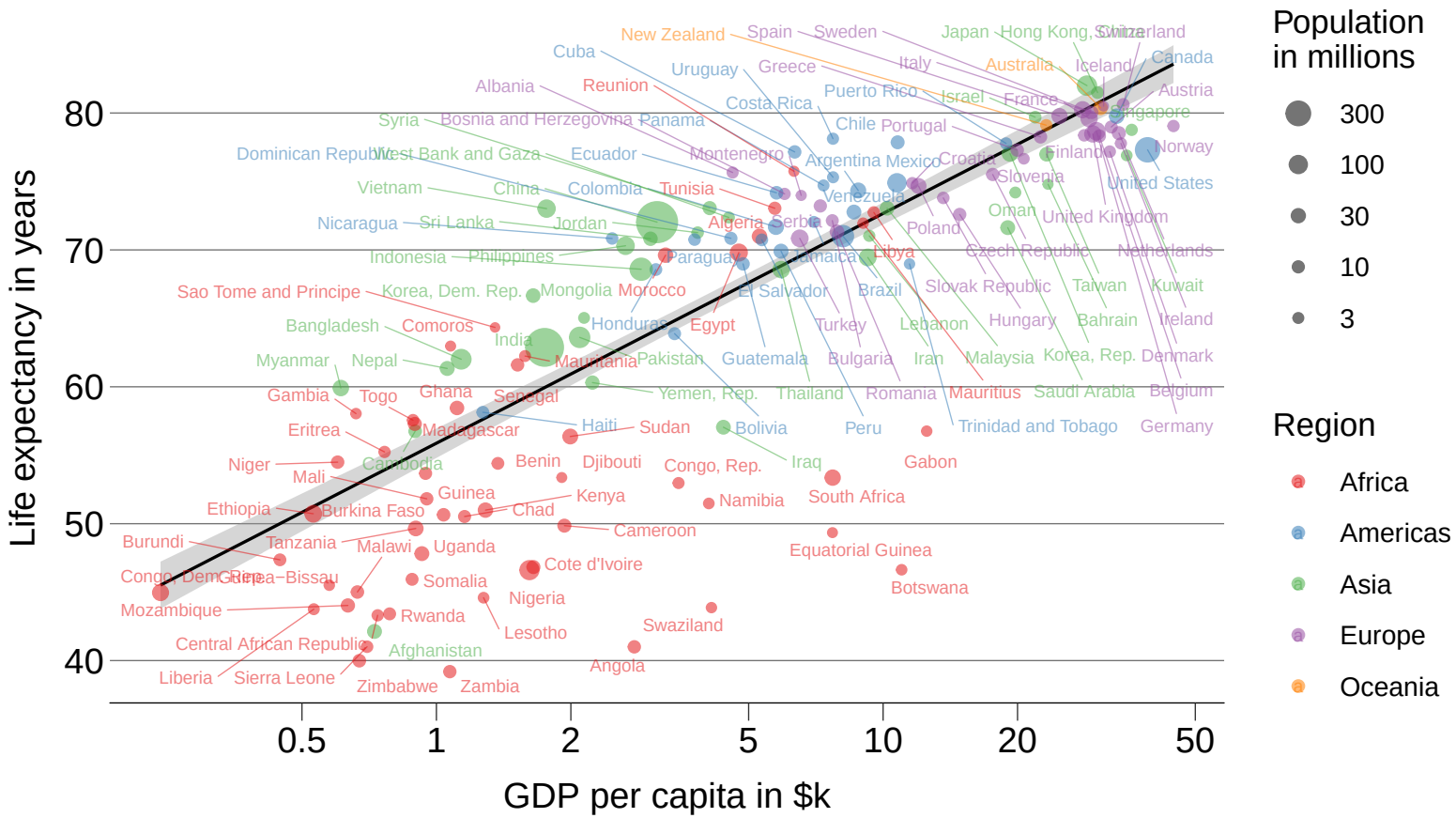
3. In all R plots, `\n` *indicates a line break – note the absence of spaces!*



It's hard to learn about specific cases without labels

We could add labels with `geom_text`, but they'd overlap the points and each other

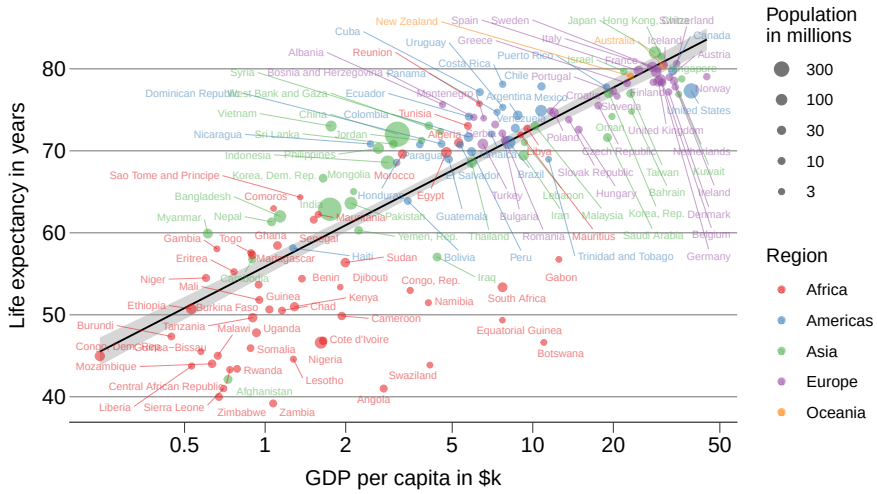
Build a scatterplot using `ggplot2` *7.3 Labeling points*



The add-on function `ggrepel::geom_text_repel` provides a clever solution

But this case is just too much for any complete text labeling scheme

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     geom_text_repel(aes(label=country),
                       size=2, alpha=0.55,
                       segment.size=0.2) +
     scale_x_log10(breaks=xbreaks,
                    labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions")
```

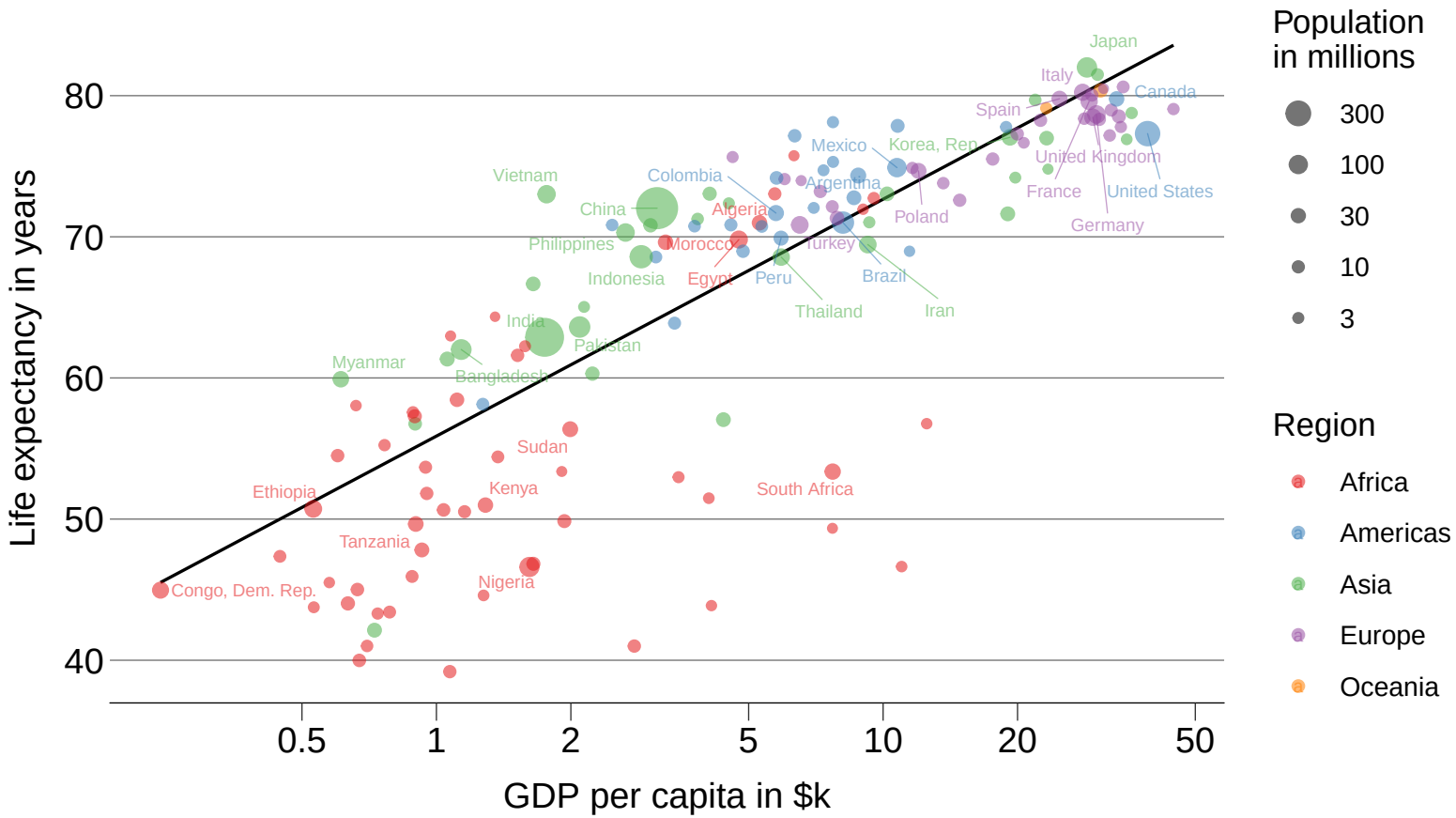


Lots of options to set in `geom_text_repel`

1. `size` *controls the text size: use to optimize the collision/size trade-off*

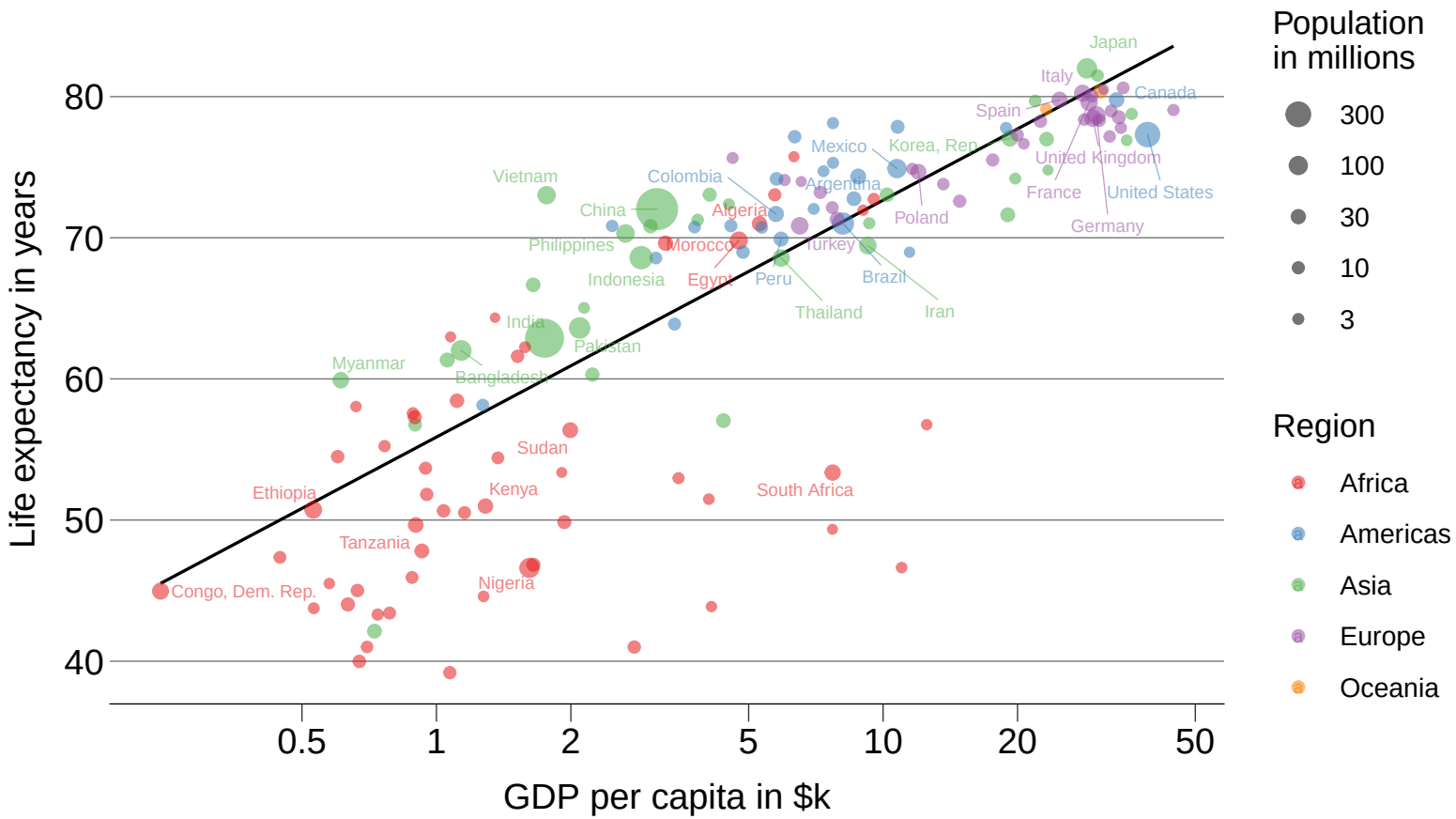
2. `segment.size` *controls line thickness; default is distractingly thick*

3. *Set* `alpha` *to match points*



Unless data are sparse, it's rarely ideal to label every point: label interesting ones

Interesting could mean "outliers" or cases that have something specific in common

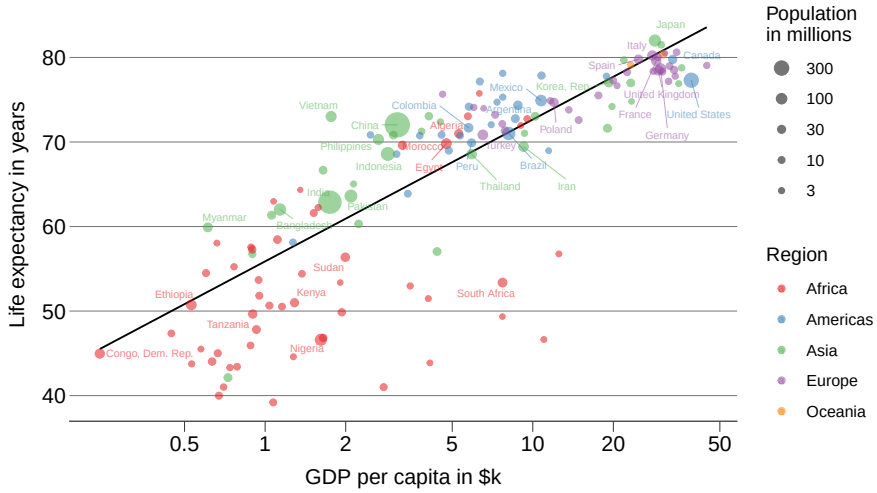


As a demonstration, let's label the top 25% of countries by population

But really, your creativity is the only limit here

```
ctyLabs <- as.character(gap2002$country)
ctyLabs[gap2002$pop<quantile(gap2002$pop, probs=0.75)] <- ""
```

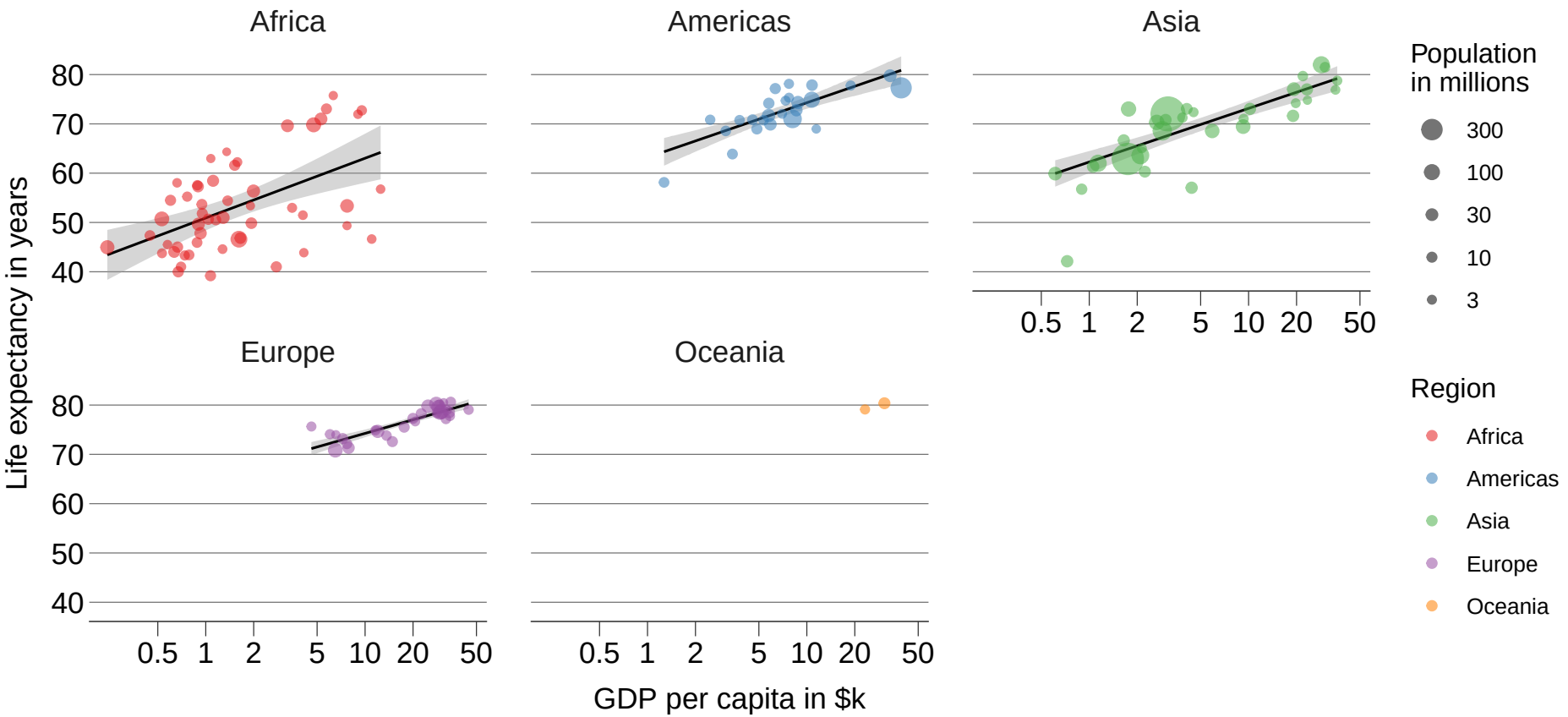
```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                   size=0.5, inherit.aes=FALSE,
                   aes(x=gdpPercap, y=lifeExp),
                   color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     geom_text_repel(aes(label=ctyLabs),
                       size=2, alpha=0.55,
                       segment.size=0.2) +
     scale_x_log10(breaks=xbreaks,
                    labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions")
```



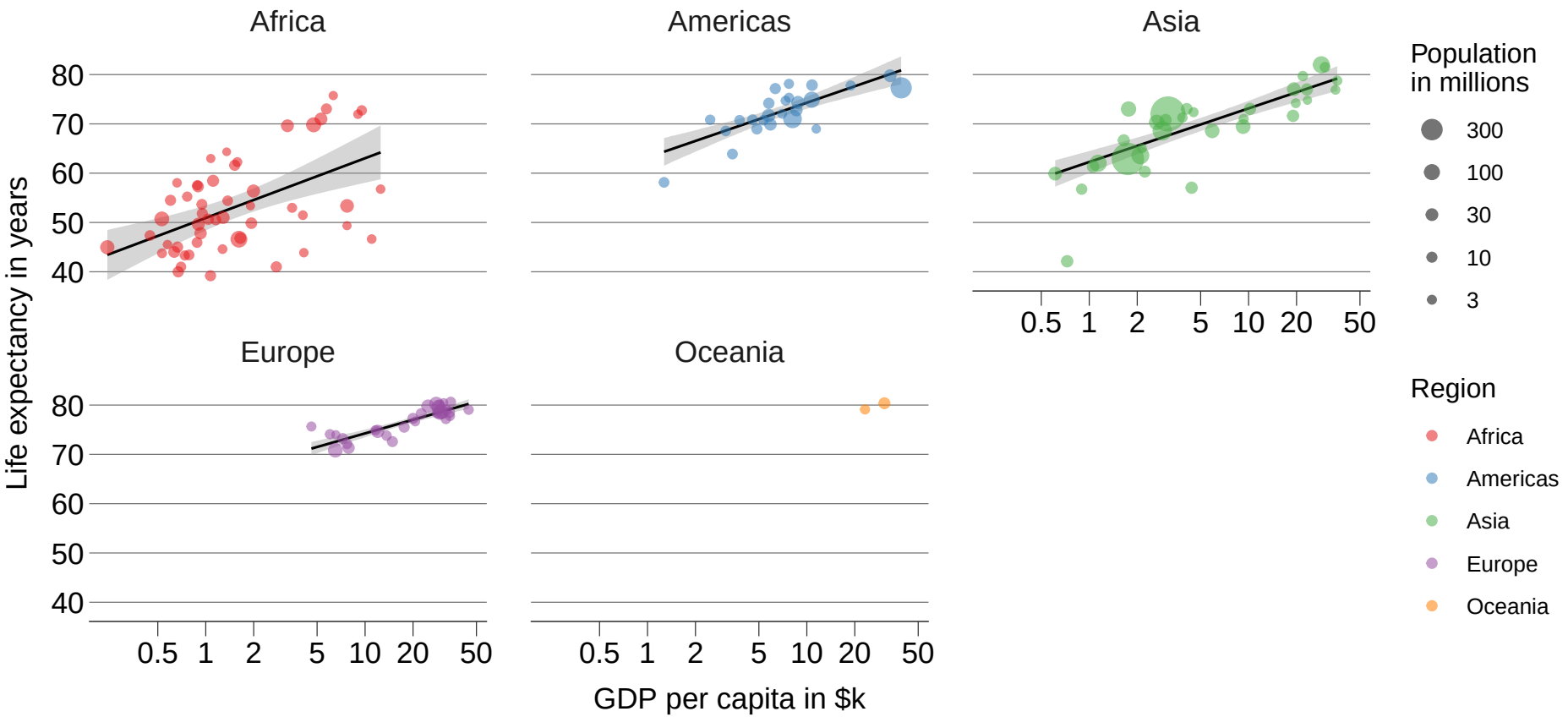
The main change here is outside the graphics code

ctyLabs *is a custom label vector that is blank when countries are small*

Note that this is an object from outside our main dataframe

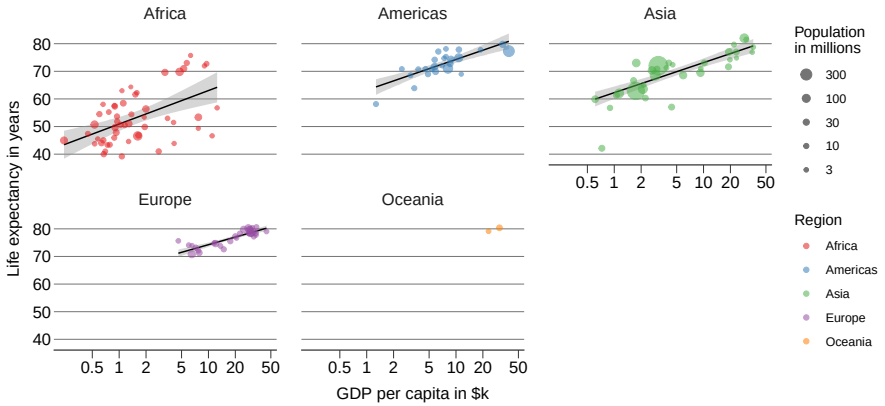


Small multiples are a major theme of the class, and ggplot makes it easy to get started (though customization can be tricky)



The key functions are `facet_wrap()`, which sorts plots on one dimension and the more rigid `facet_grid()`, which sorts on two

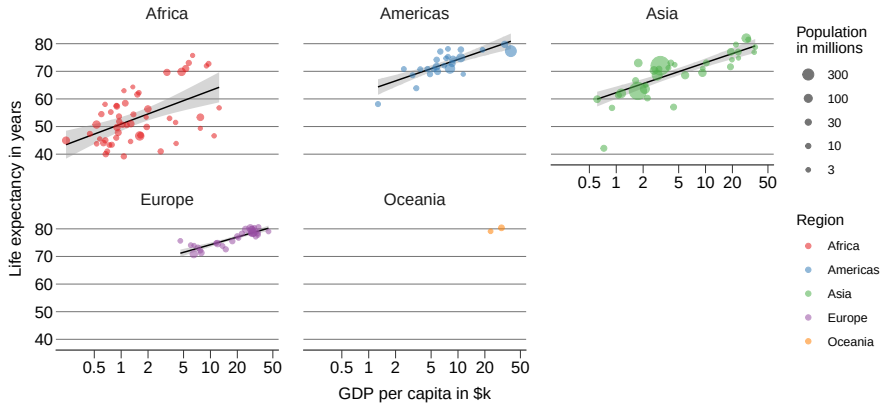
```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions") +
     facet_wrap(~continent)

width <- 9
ggsave("ggScatterEx8_1.pdf",
       width=width, height=width/1.618)
```



Getting started with small multiples can be as easy as telling `facet_wrap` *to break the data down by continent*

Now the data from each continent gets is own graph

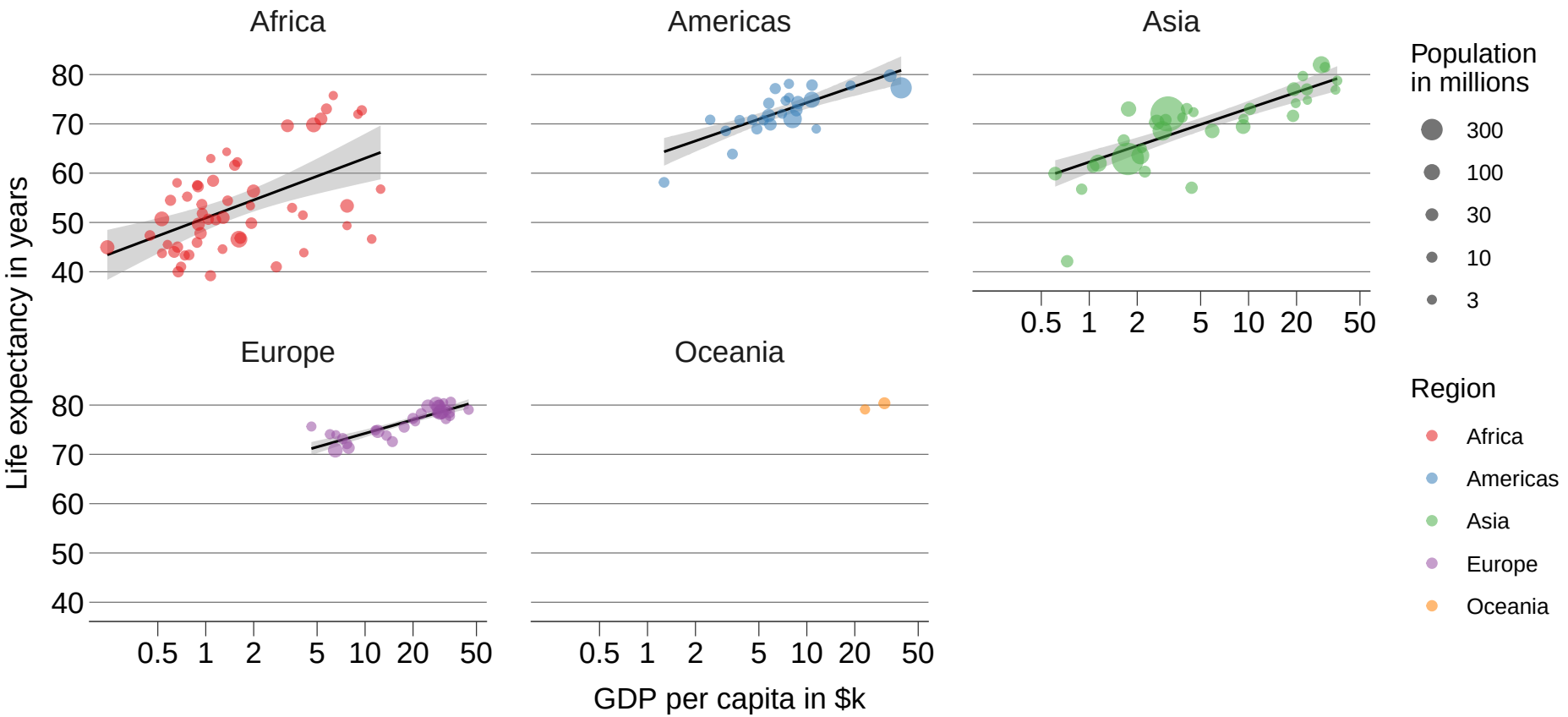
```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.55, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_brewer(palette = "Set1") +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          col="Region",
          size="Population\nin millions") +
     facet_wrap(~continent)

width <- 9
ggsave("ggScatterEx8_1.pdf",
       width=width, height=width/1.618)
```



Effective use of this command depends on dataframe structure: you need a categorical variable that subsets your observations across small multiples

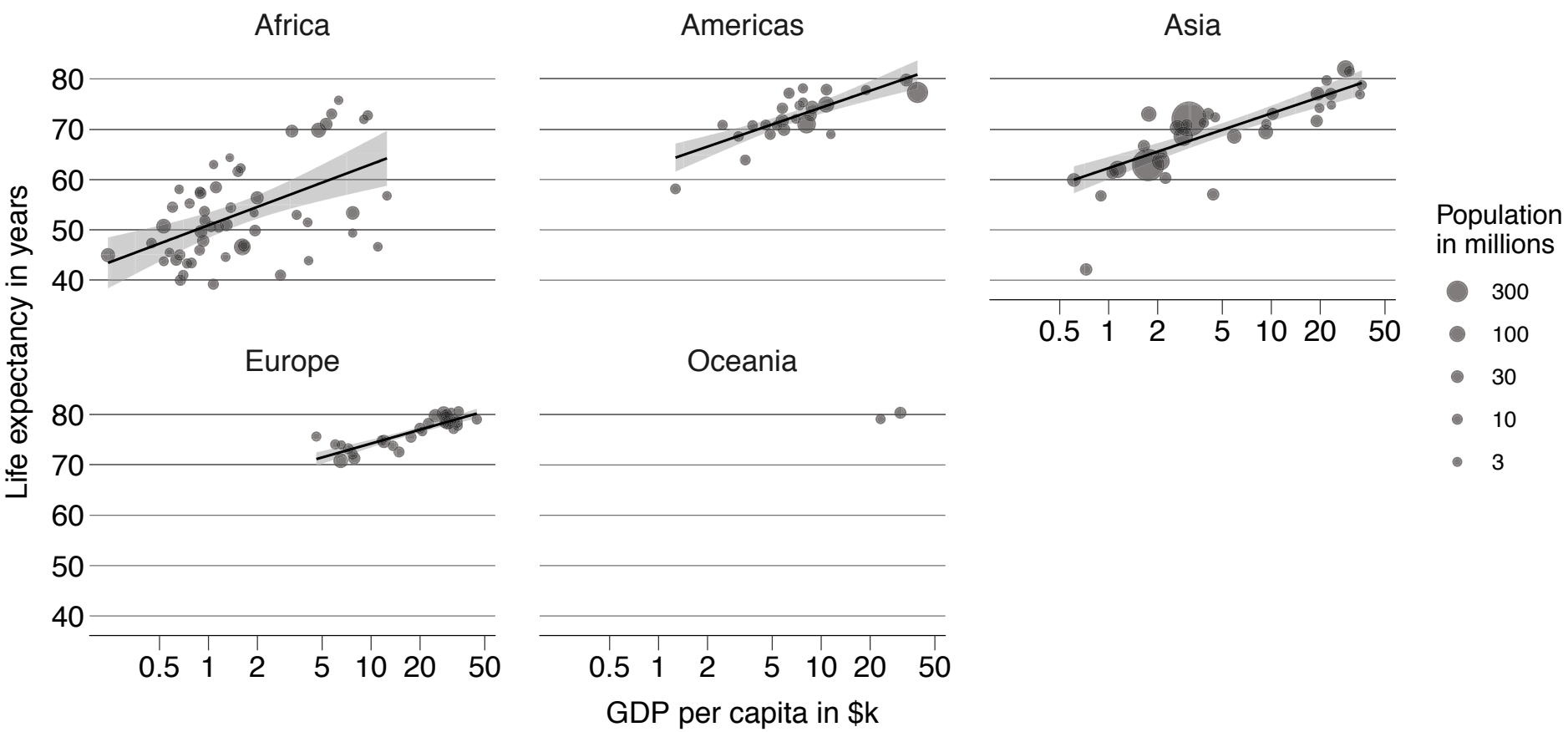
Note also that the width of the PDF is now 8 instead of 7 (why?)

Build a scatterplot using `ggplot2` — *8.2 Remove unused color*



While it's easy to get started with small multiples, there's still much to do

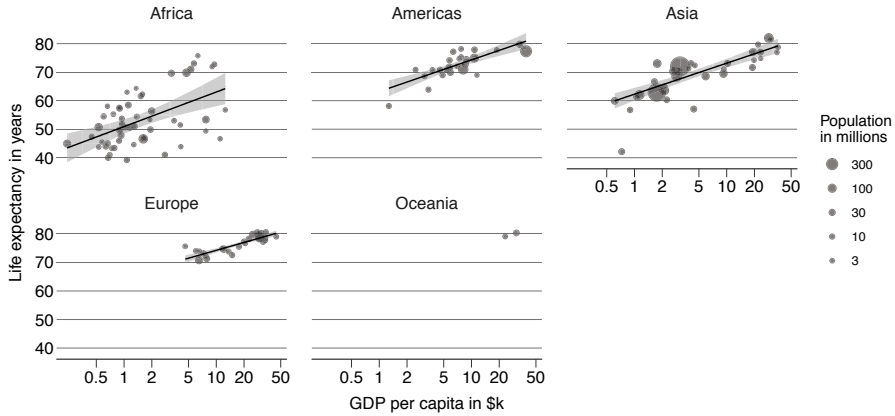
For a start, the colors are now superfluous, and should be saved for a better purpose



An unnecessary riot of color distracts from the comparisons at hand

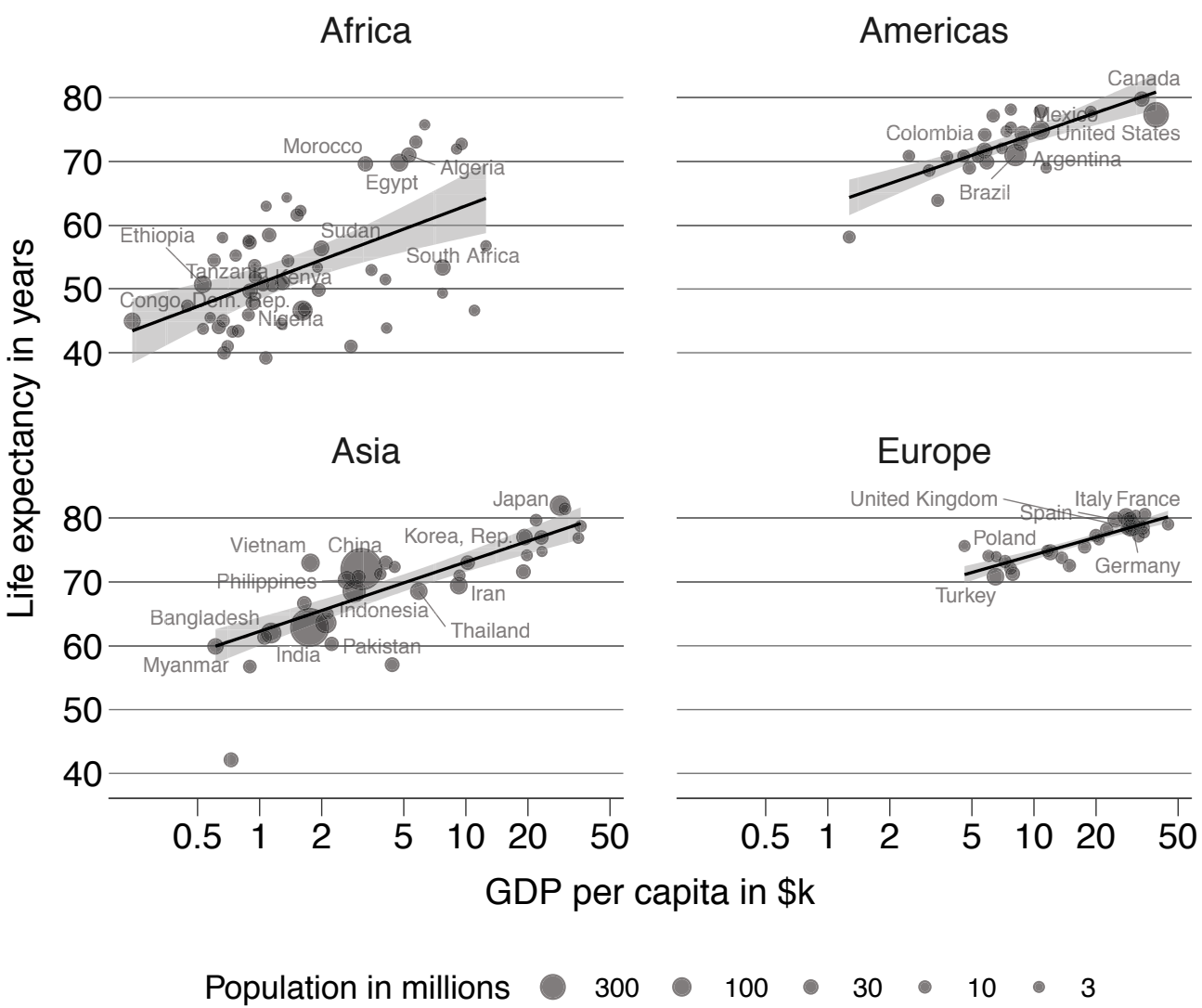
Now we have a cleaner plot and the potential to use color to show another variable

```
p <- ggplot(data=gap2002,
            mapping=aes(x=gdpPercap,
                        y=lifeExp))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.40, aes(size=pop)) +
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_size(breaks = popBreaks,
                labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k",
          y="Life expectancy in years",
          size="Population\nin millions") +
     facet_wrap(~continent)
```



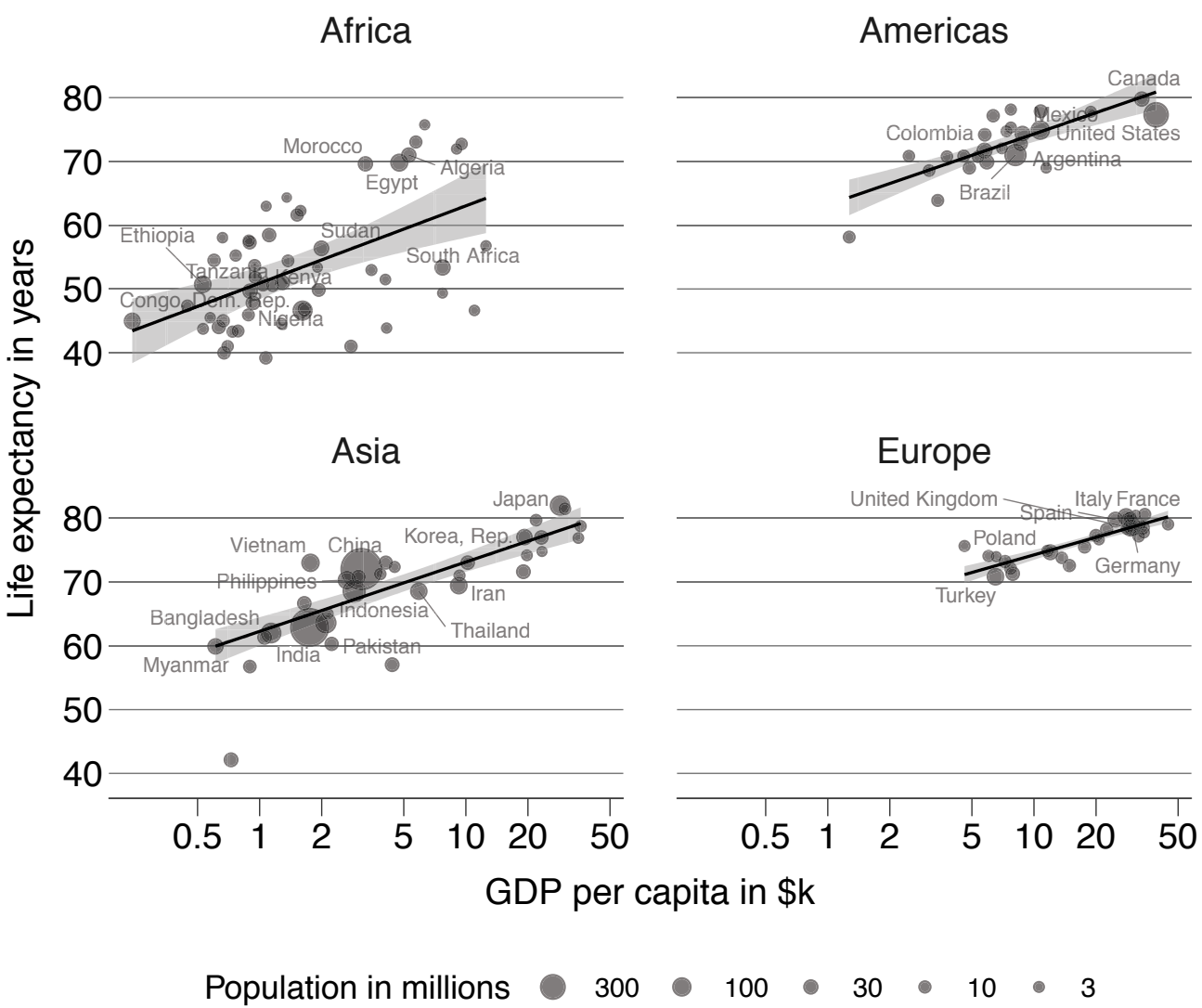
We simply remove the mapping of color to continents, and the color scale customization

Once again, the right-side scale looks like a waste of space – perhaps we could move it to the empty "sixth plot"?



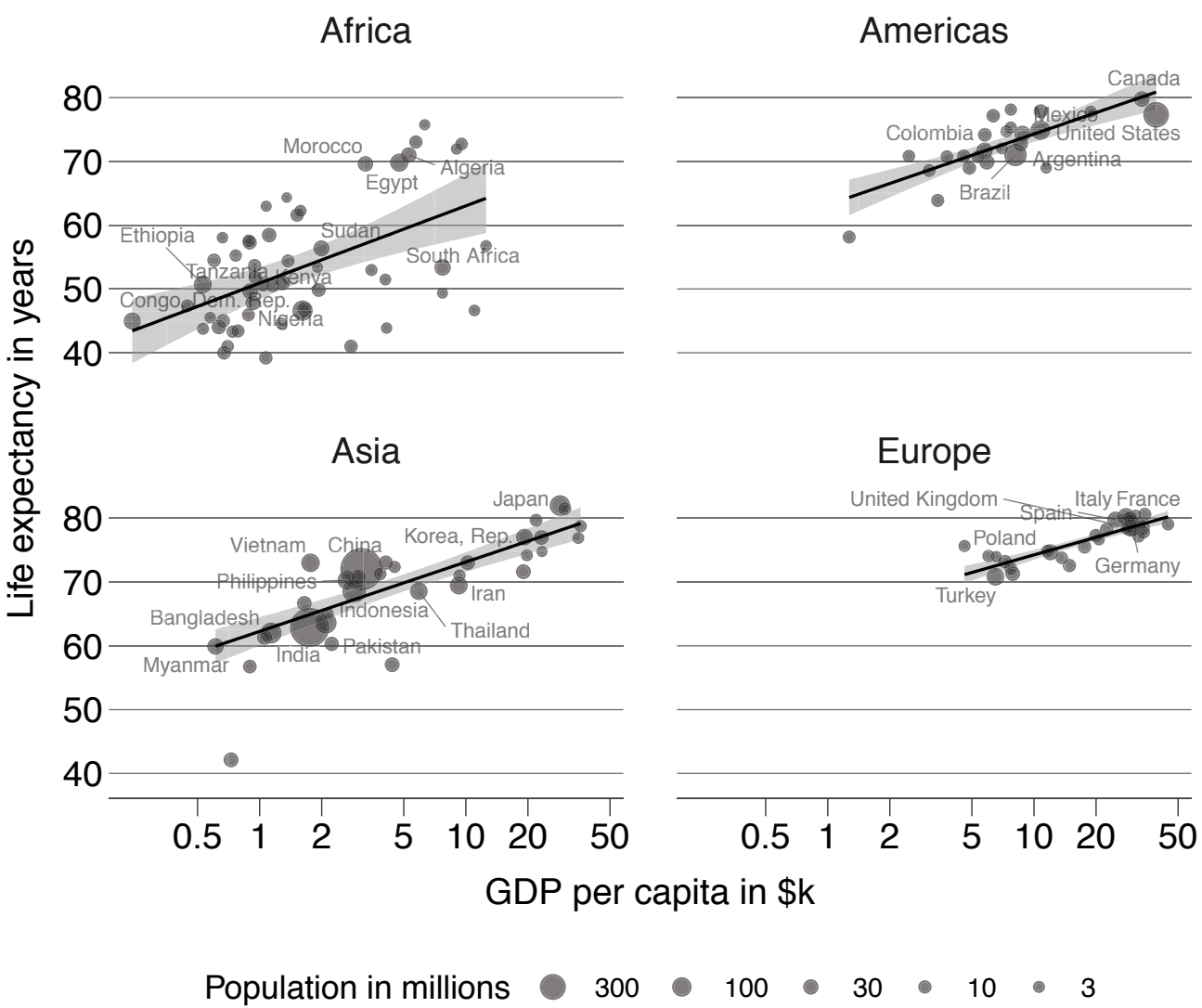
The data for Oceania are too few to assess a trend – just Australia and New Zealand – so let's delete that plot

Now we have a simple 2x2 grid, so it makes more sense to put the legend below the plot



The panels are sorted alphabetically – if we were telling a story, this is probably not the order we'd choose

Fixing this in `ggplot` is hard: the panels must be sorted by the order of the factor variable `continent`, so we'd have to change that factor in the dataframe to match our preferred order

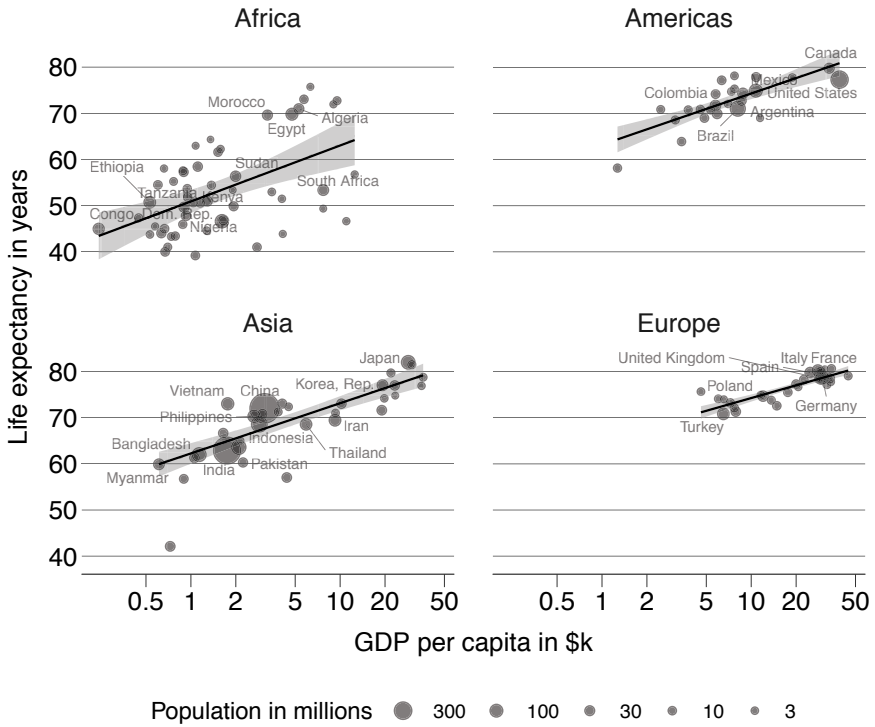


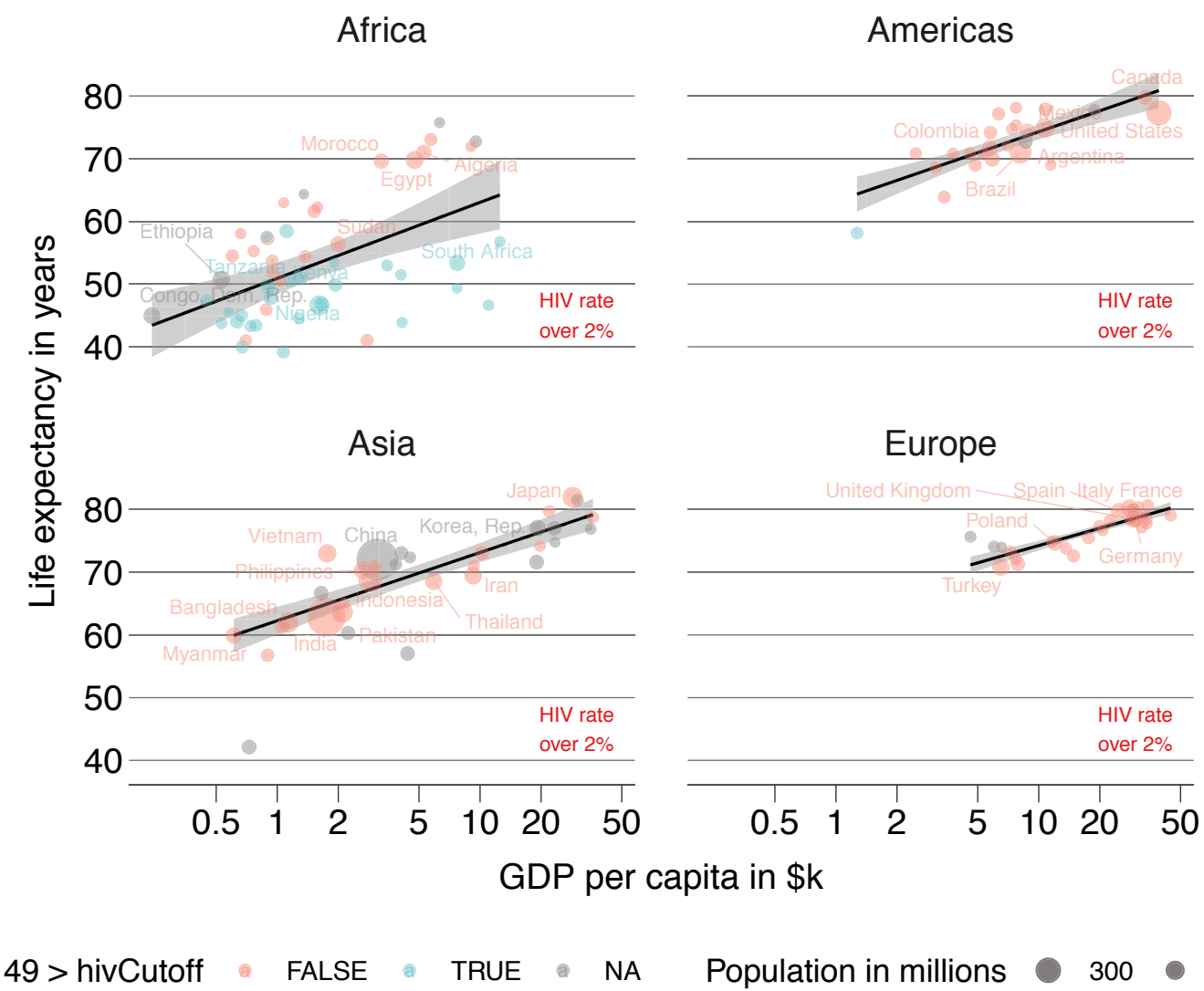
Finally, note that the individual panels are looking a lot more linear (hard to see when the data were overlapping)

Instead of nonlinearity, we see higher variability in Africa

What might explain that variability?

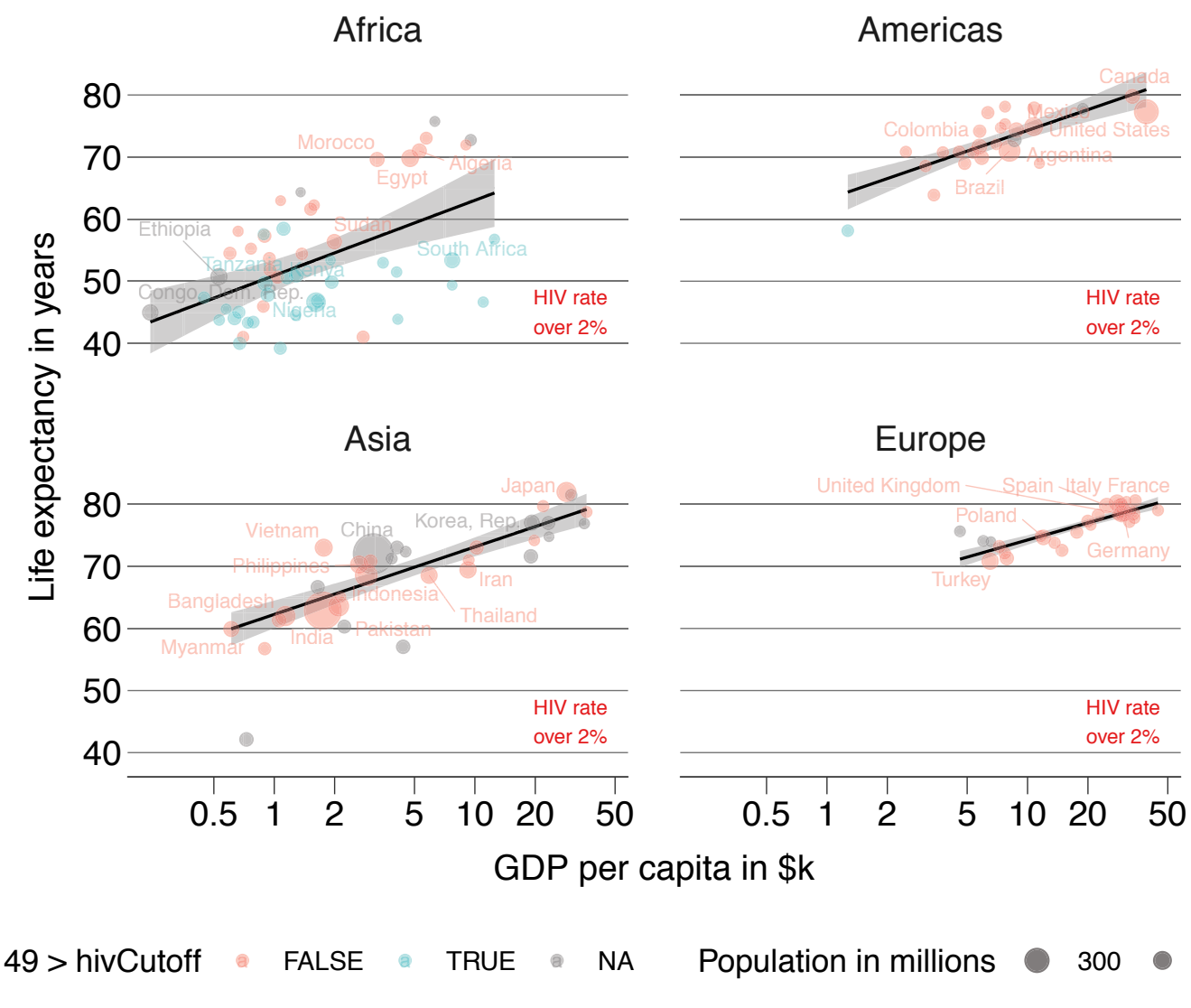
```
gap2002noO <- gap2002[gap2002$continent!="Oceania",]

ctyLabs0 <- as.character(gap2002noO$country)
ctyLabs0[gap2002noO$pop <
   quantile(gap2002noO$pop, probs=0.75)] <- ""

p <- ggplot(data=gap2002noO,
        mapping=aes(x=gdpPercap, y=lifeExp))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp), color="black") +
     geom_point(alpha=0.40, aes(size=pop)) +
     geom_text_repel(aes(label=ctyLabs0), size=2.5, alpha=0.40,
                     segment.size=0.2) +
     scale_x_log10(breaks=xbreaks, labels=xbreaks/1000) +
     scale_size(breaks = popBreaks, labels = popBreaks/1000000) +
     labs(x="GDP per capita in $k", y="Life expectancy in years",
          size="Population in millions") +
     facet_wrap(~continent) + theme(legend.position="bottom")
```





Let's highlight countries with high rates of HIV infection, which may explain the outliers in Africa

I chose the 2% cutoff after looking at the density plot of HIV prevalence rates and experimenting with different values

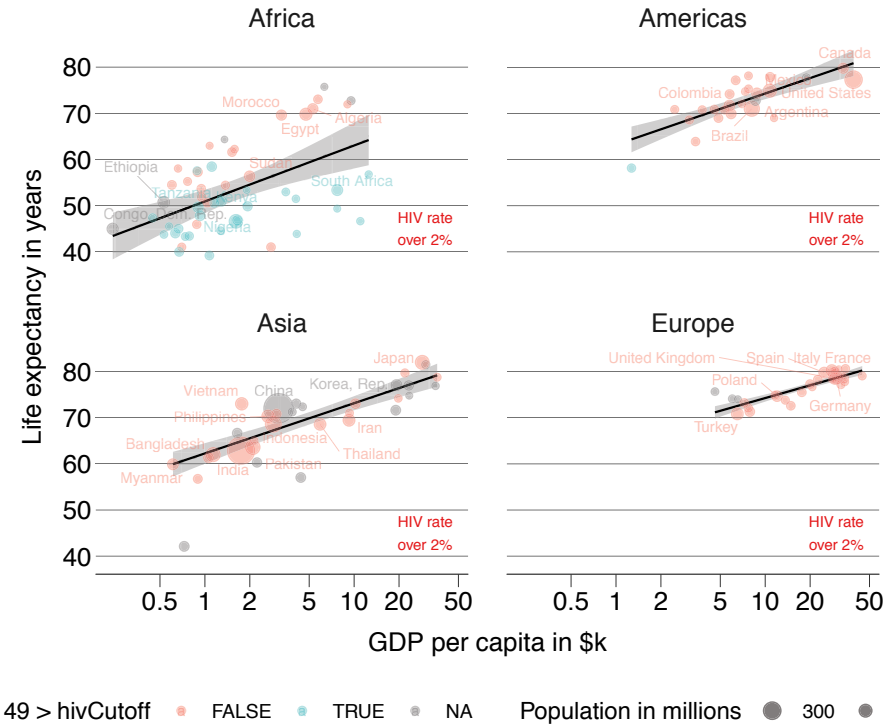


Initial indications are that many of the outliers are high HIV rate countries, but the crazy default color scheme obscures this

There's also a missing value category (and color), which we probably want to omit instead

Finally, what's happening with the legend title?

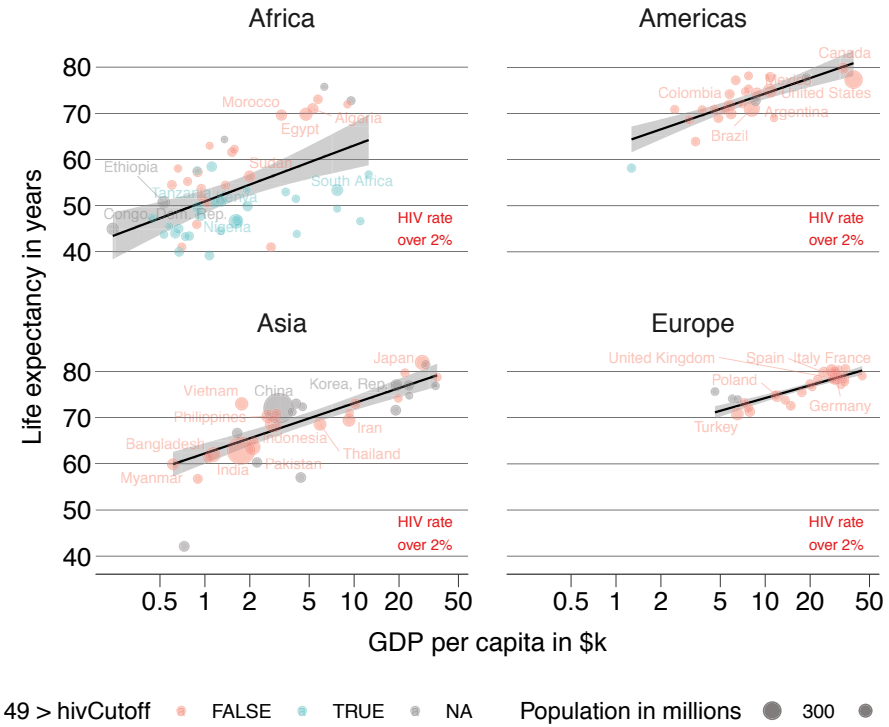
```
hivCutoff <- 2

p <- ggplot(data=gap2002noO,
            mapping=aes(x=gdpPercap, y=lifeExp))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.40, aes(size=pop,
         col=hivPrevalence15to49>hivCutoff)) +
     geom_text_repel(aes(label=ctyLabs0,
         col=hivPrevalence15to49>hivCutoff),
         size=2.5, alpha=0.40, segment.size=0.2) +
     ...
     annotate(geom="text", x=30000, y=45,
              label=paste0("HIV rate\nover ",
                           hivCutoff, "%"),
              col=red, size=2.5)
```



We inserted `hivPrevalence15to49 > hivCutoff` *as the point color mapping, and this became the legend title*

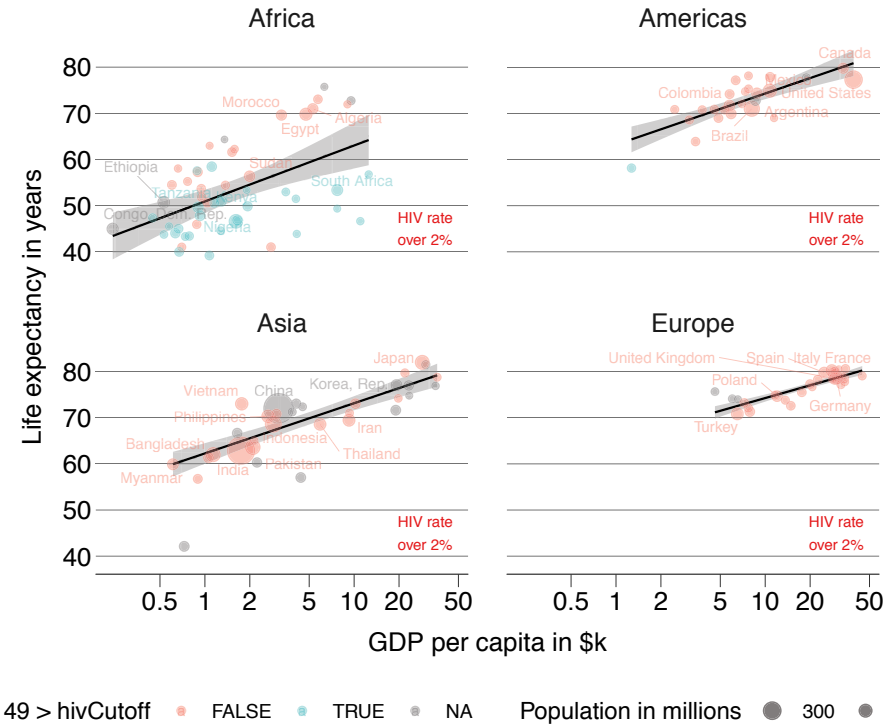
(Note, in passing, the use of an easily modified object to set the cutoff)

```
hivCutoff <- 2

p <- ggplot(data=gap2002noO,
            mapping=aes(x=gdpPercap, y=lifeExp))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.40, aes(size=pop,
         col=hivPrevalence15to49>hivCutoff)) +
     geom_text_repel(aes(label=ctyLabs0,
         col=hivPrevalence15to49>hivCutoff),
         size=2.5, alpha=0.40, segment.size=0.2) +
     ...
     annotate(geom="text", x=30000, y=45,
              label=paste0("HIV rate\nover ",
                           hivCutoff, "%"),
              col=red, size=2.5)
```

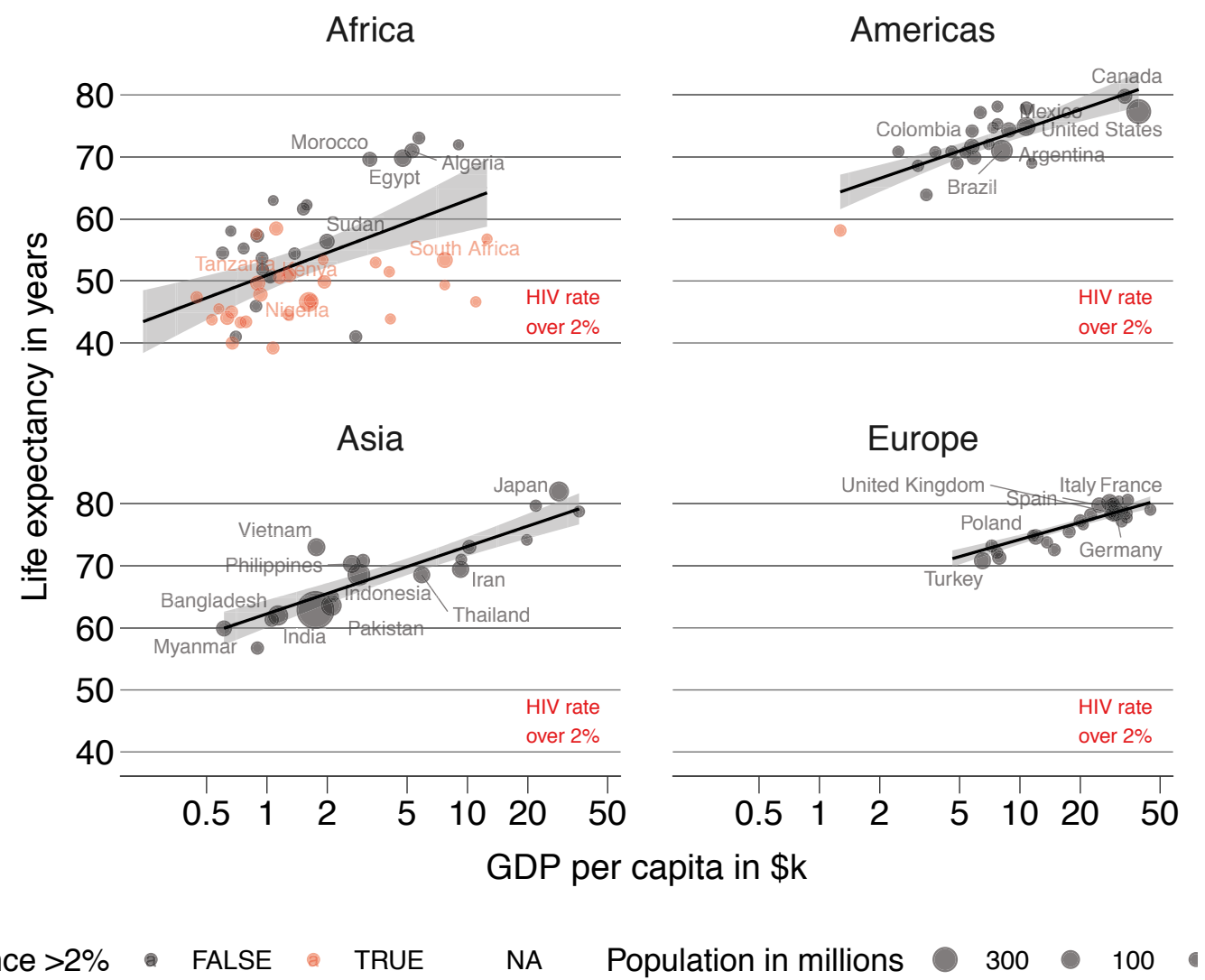


We've also introduced annotate(), *by which we can add specific text labels or other graphic elements idiosyncratically into the plot*

```
hivCutoff <- 2

p <- ggplot(data=gap2002noO,
            mapping=aes(x=gdpPercap, y=lifeExp))

p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5, inherit.aes=FALSE,
                 aes(x=gdpPercap, y=lifeExp),
                 color="black") +
     geom_point(alpha=0.40, aes(size=pop,
         col=hivPrevalence15to49>hivCutoff)) +
     geom_text_repel(aes(label=ctyLabs0,
         col=hivPrevalence15to49>hivCutoff),
         size=2.5, alpha=0.40, segment.size=0.2) +
     ...
     annotate(geom="text", x=30000, y=45,
              label=paste0("HIV rate\nover ",
                           hivCutoff, "%"),
              col=red, size=2.5)
```



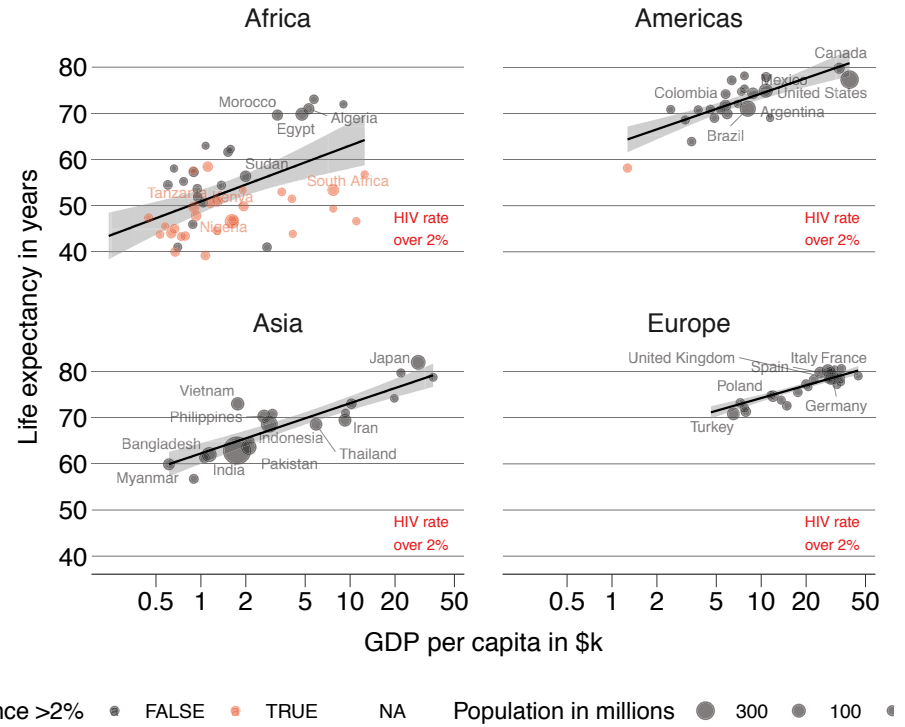
Unfortunately, this feature cannot be targeted at a single panel, so appears in the same place in every facet



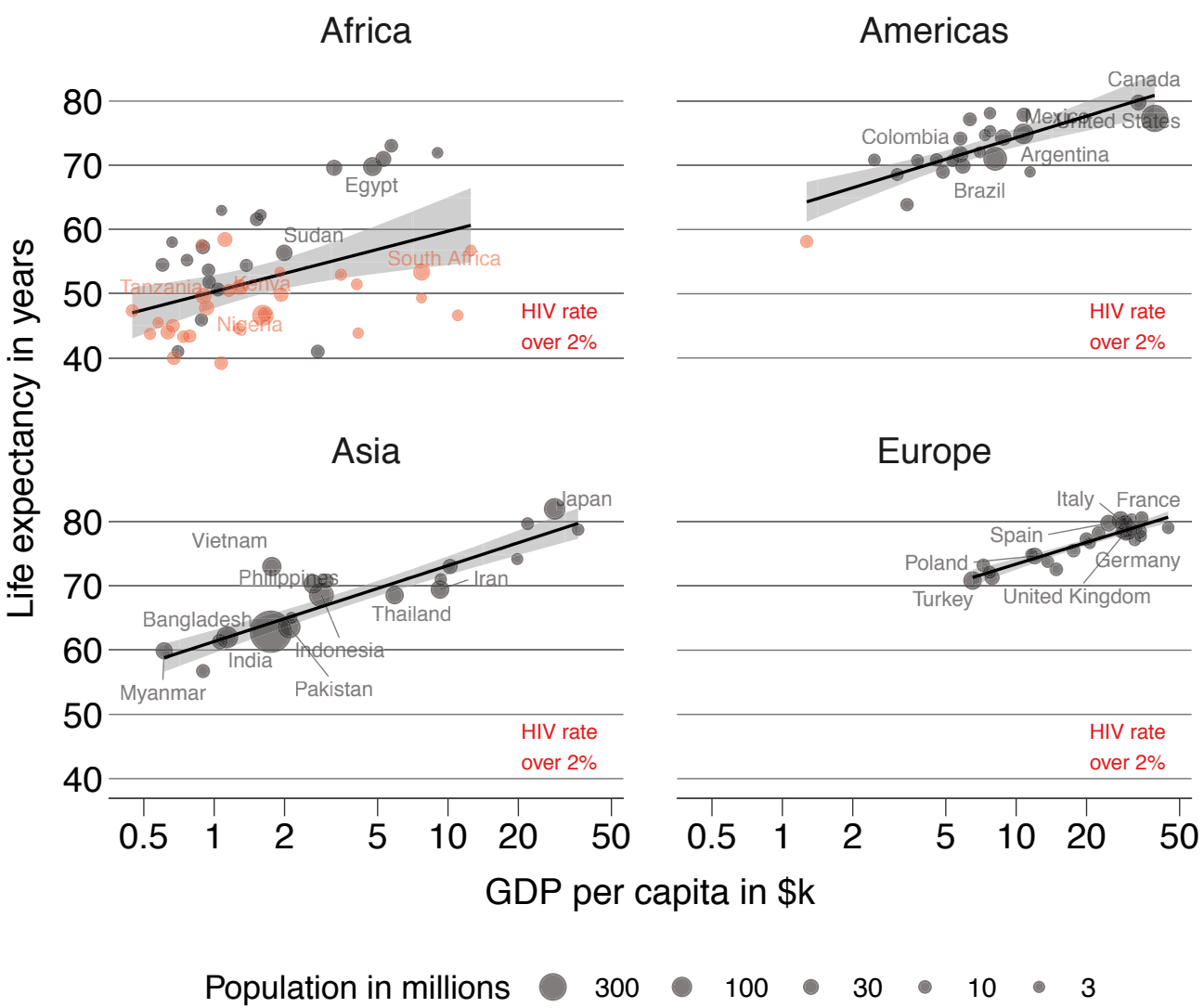
Let's fix the colors so that the HIV outliers pop: red for high HIV cases, and black for other cases

But there's another problem: the missing cases are being plotted "invisibly", so they still affect the fit lines!

```
p <- p + goldenScatterCAtheme +
    geom_smooth(method=MASS::rlm,
                method.args=list(method="MM"),
                size=0.5, inherit.aes=FALSE,
                aes(x=gdpPercap, y=lifeExp),
                color="black") +
    geom_point(alpha=0.40, aes(size=pop,
        col=hivPrevalence15to49>hivCutoff)) +
    geom_text_repel(aes(label=ctyLabs0,
        col=hivPrevalence15to49>hivCutoff),
        size=2.5, alpha=0.40, segment.size=0.2)
    scale_x_log10(breaks=xbreaks,
                  labels=xbreaks/1000) +
    scale_color_manual(values=c(black, red)) +
    scale_size(breaks = popBreaks,
               labels = popBreaks/1000000) +
    scale_color_manual(values=c(black, red)) +
    ...
    annotate(geom="text", x=30000, y=45,
             label=paste0("HIV rate\nover ",
                          hivCutoff, "%"),
             col=red, size=2.5)
```



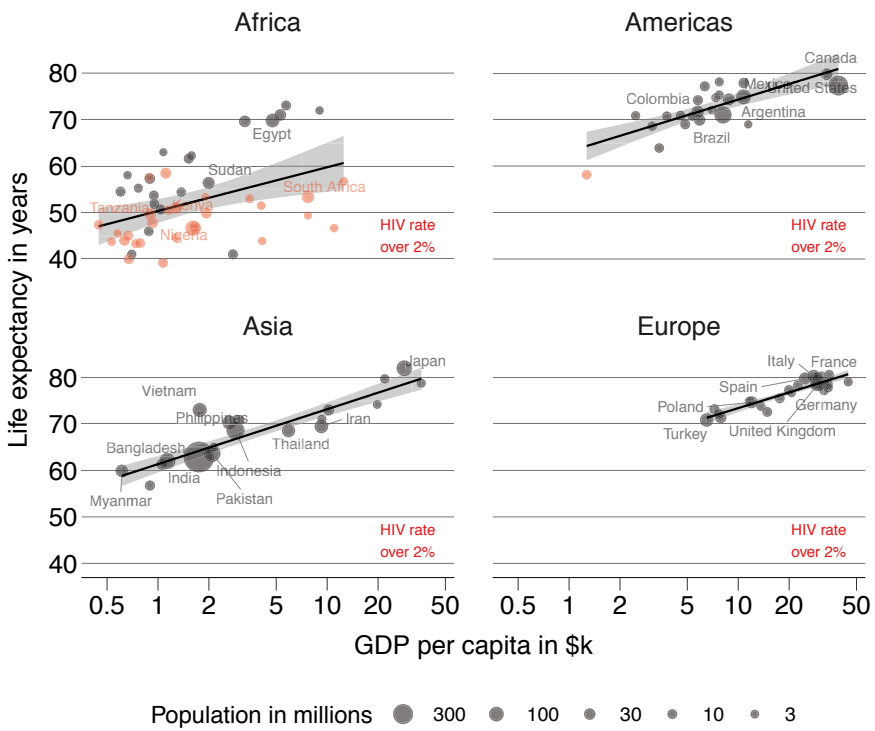
Setting `scale_color_manual` *overrides the usual color scale with whatever colors we want for each category*



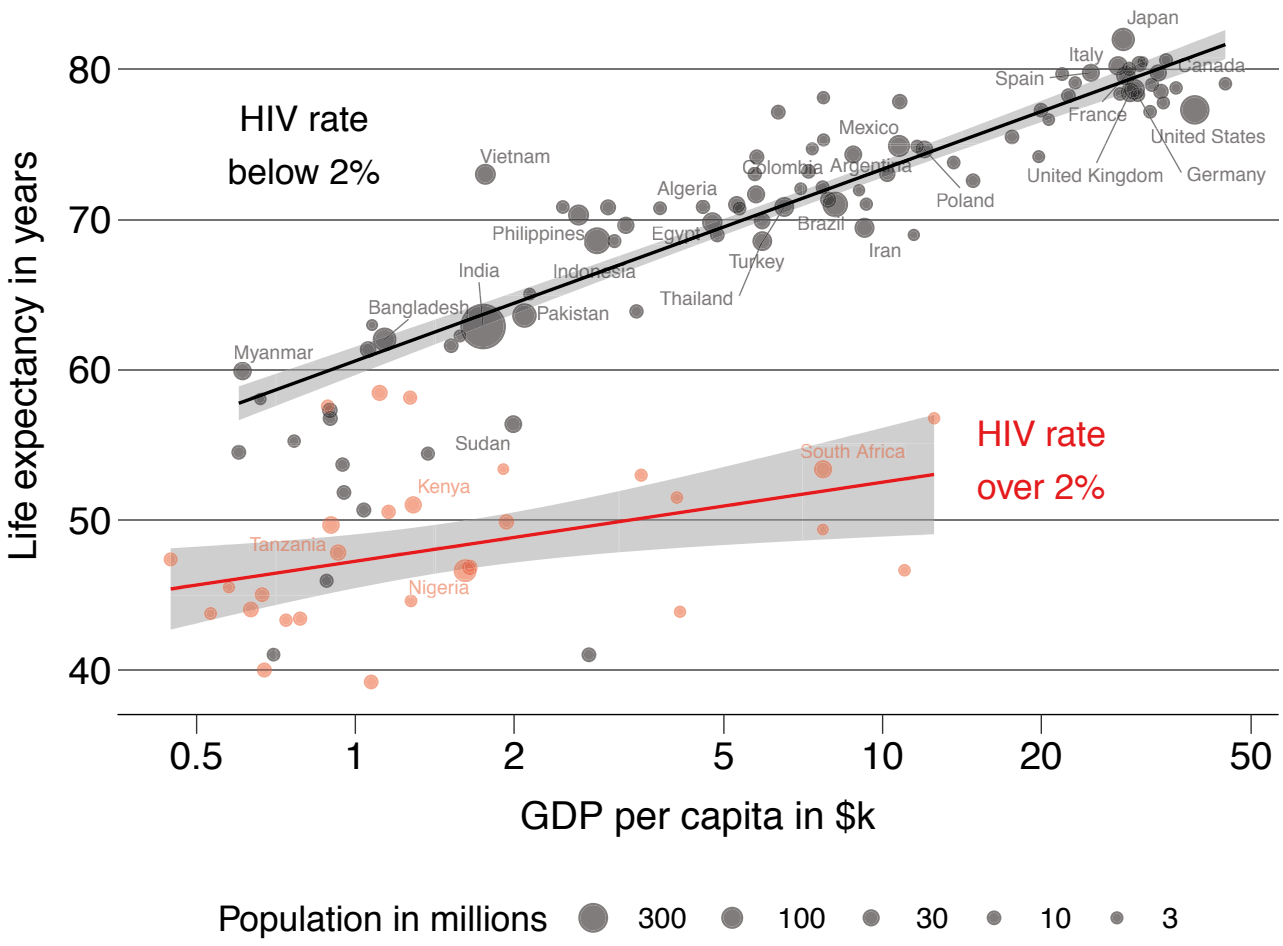
Let's make sure the cases that are missing HIV data get deleted before we start plotting

After making a dataframe without these cases, we see the plot at left

```
library(simcf)  ## for extractdata(), from faculty.washington.edu/cadolph/software
mdata1 <- extractdata(lifeExp~gdpPercap + continent + hivPrevalence15to49,
                      data=gap2002, extra=~country+pop, na.rm=TRUE)
mdata1 <- mdata1[mdata1$continent!="Oceania",]
ctyLabs1 <- as.character(mdata1$country)
ctyLabs1[mdata1$pop<quantile(mdata1$pop,
                      probs=0.75)] <- ""

p <- ggplot(data=mdata1,
            mapping=aes(x=gdpPercap, y=lifeExp))
p <- p + goldenScatterCAtheme +
    geom_smooth(method=MASS::rlm,
                method.args=list(method="MM"),
                size=0.5, inherit.aes=FALSE,
                aes(x=gdpPercap, y=lifeExp),
                color="black") +
    geom_point(alpha=0.40, aes(size=pop,
        col=hivPrevalence15to49>hivCutoff)) +
    geom_text_repel(aes(label=ctyLabs1,
        col=hivPrevalence15to49>hivCutoff),
        size=2.5, alpha=0.40, segment.size=0.2) +
    ...
```



simcf::extractdata() helps us delete on just the variables we need for the plot, and not any missing cases of extraneous variables (available from my website)

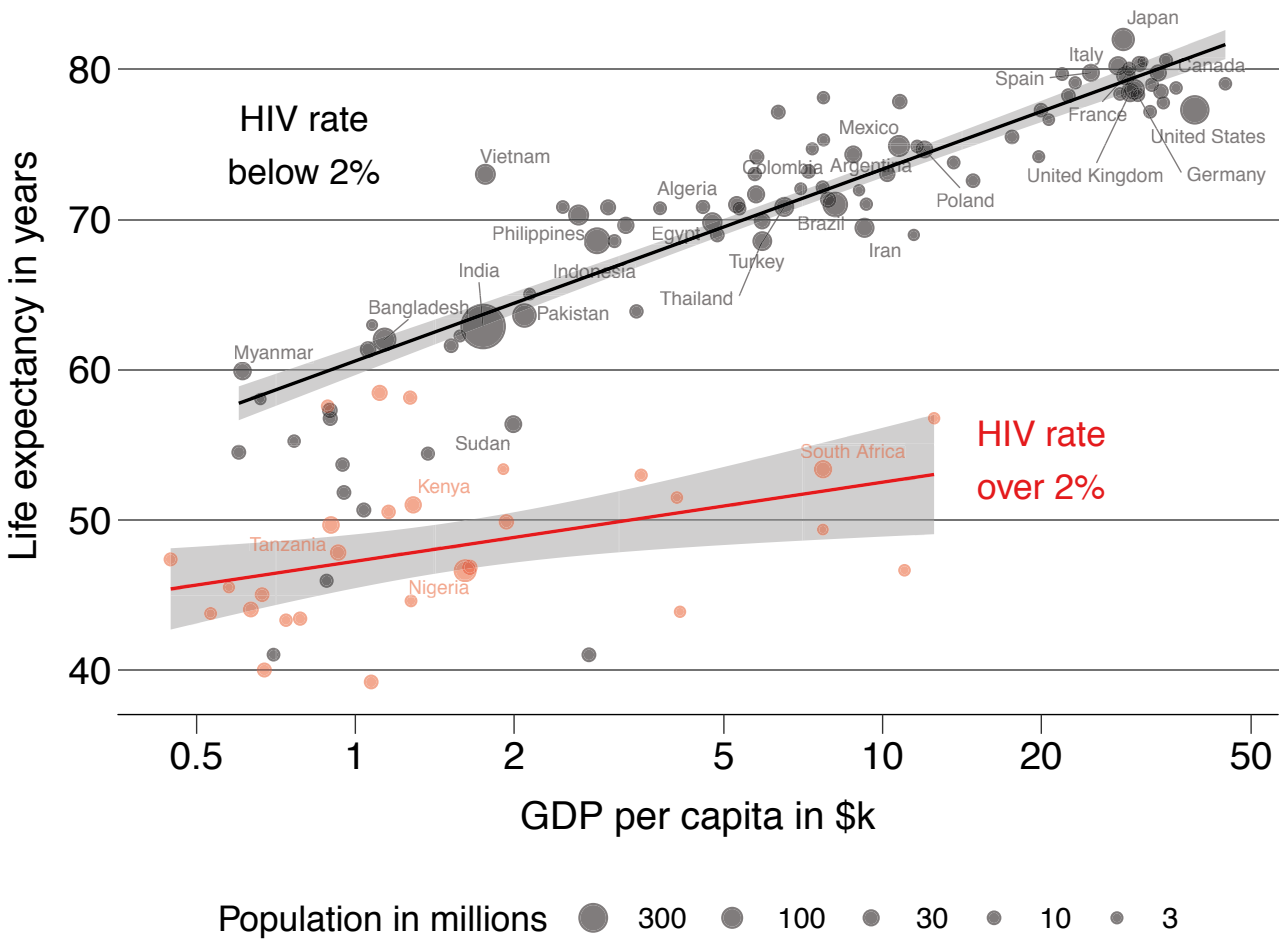


What if HIV is the key secondary variable here, rather than region?

Let's pool the data across regions, but separate (and plot separate robust fits) by HIV infection rate

Because the data separate so well, a single facet and direct labels now suffice

Build a scatterplot using `ggplot2` *10.1 Zoom in on the story*

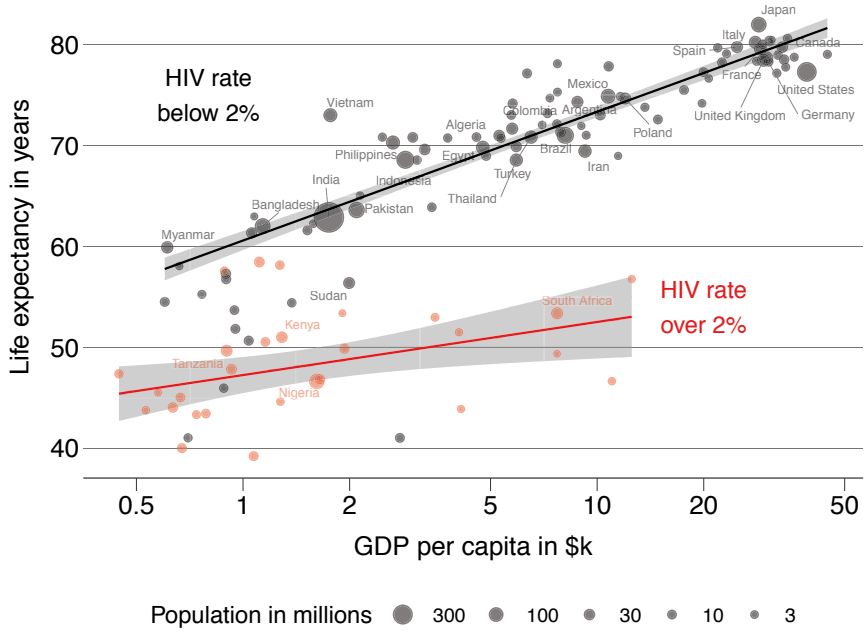


Conditional on HIV rate, life expectancy looks more (log)-linear in GDP per capita than before

Results are pushing us in the direction of a more complete model

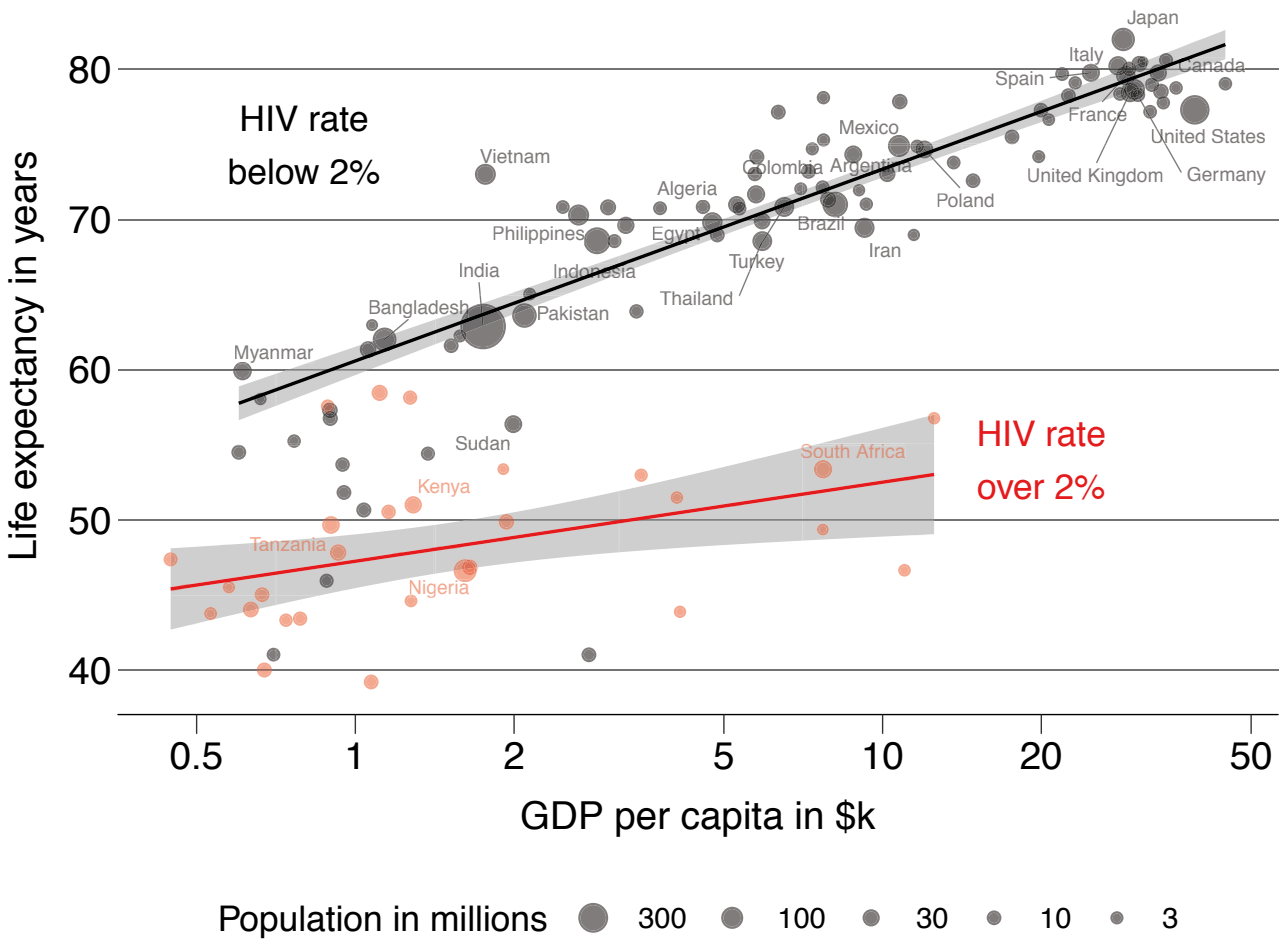
Often, the point of exploratory data analysis is to reveal how we should begin the mode-building process

```
mdata2 <- extractdata(lifeExp~gdpPercap + hivPrevalence15to49,
                      data=gap2002, extra=~country+pop, na.rm=TRUE)
ctyLabs2 <- as.character(mdata2$country)
ctyLabs2[mdata2$pop<quantile(mdata2$pop,
                       probs=0.75)] <- ""
p <- ggplot(data=mdata2,
            mapping=aes(x=gdpPercap, y=lifeExp,
             color=hivPrevalence15to49>hivCutoff))
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5) +
     geom_point(alpha=0.40, aes(size=pop)) +
     geom_text_repel(aes(label=ctyLabs2), size=2,
                     alpha=0.40, segment.size=0.2) +
     ...
     annotate(geom="text", x=20000, y=54,
              label=paste0("HIV rate\nover ",
                           hivCutoff, "%"),
              col=red, size=4) +
     annotate(geom="text", x=800, y=75,
              label=paste0("HIV rate\nbelow ", hivCutoff, "%"), col=black, size=4)
```



Note new dataframe (put Australia & NZ back in)

Note new annotation

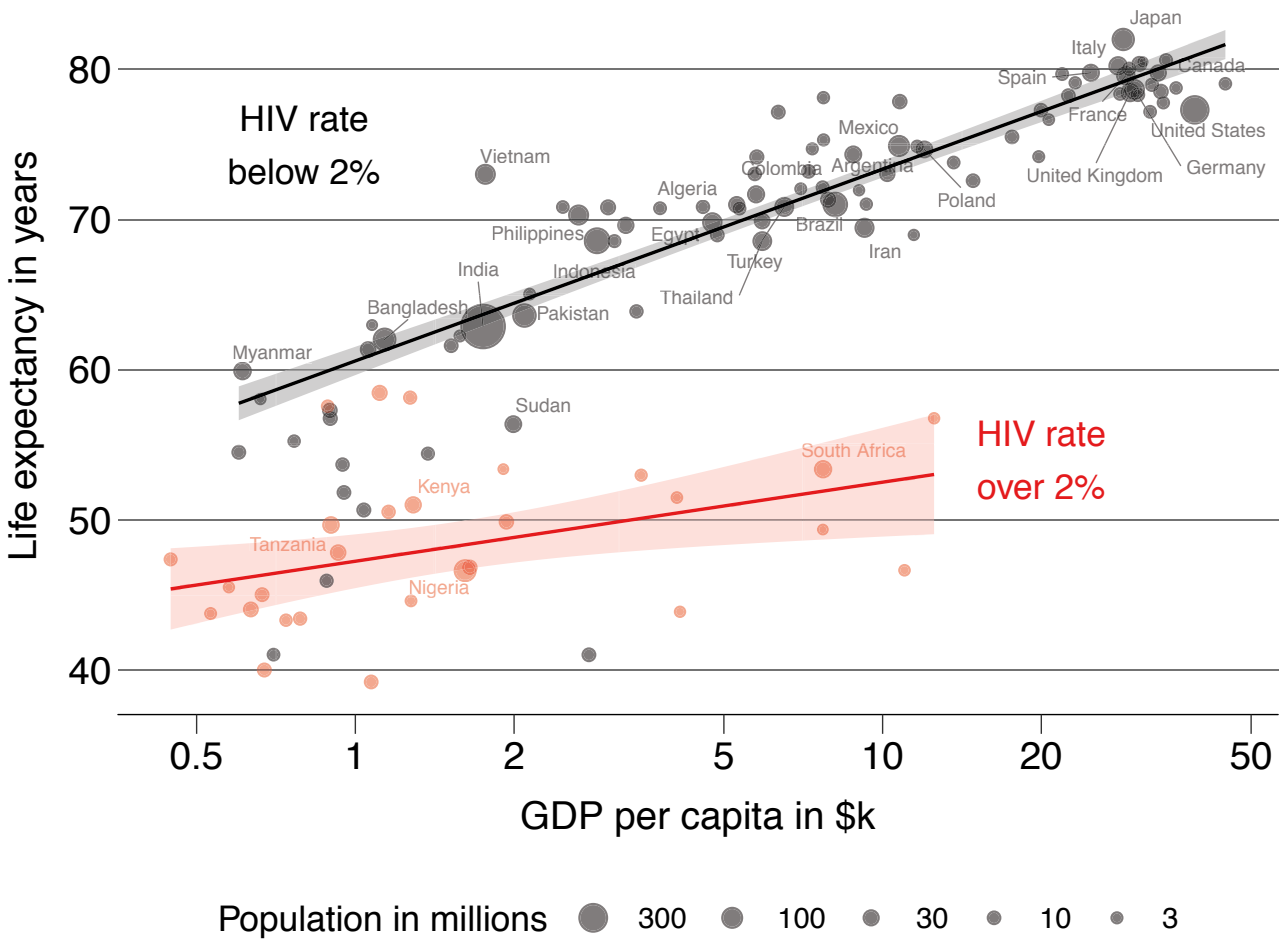


One last fix: CI "ribbon" should match color of fit line

Helps overall readability, especially with multiple fits

Surprisingly tricky to do...

Build a scatterplot using `ggplot2` *10.2 Color cleanup*



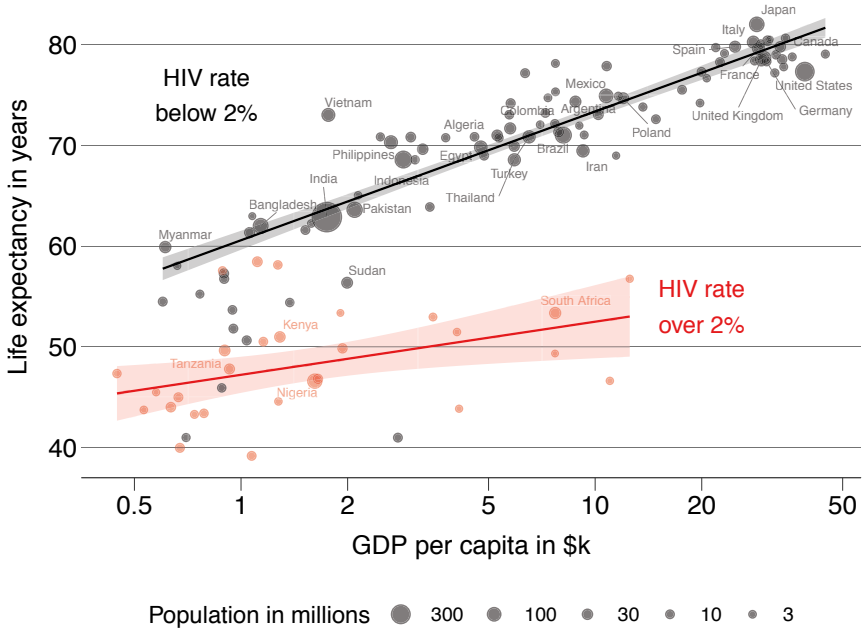
Note that the text labels jump a bit, and in generally aren't always clear: `ggrepel` isn't perfect

I touch up issues like this in Adobe Illustrator prior to publication

Final thoughts?

Anything else you'd add or change?

```
p <- p + goldenScatterCAtheme +
     geom_smooth(method=MASS::rlm,
                 method.args=list(method="MM"),
                 size=0.5,
        aes(fill=hivPrevalence15to49>hivCutoff))
     geom_point(alpha=0.40, aes(size=pop)) +
     geom_text_repel(aes(label=ctyLabs2), size=2,
                     alpha=0.40, segment.size=0.2)
     scale_x_log10(breaks=xbreaks,
                   labels=xbreaks/1000) +
     scale_color_manual(values=c(black, red),
                        guide="none") +
     scale_fill_manual(values=c(gray, lightred),
                       guide="none") +
     ...
```

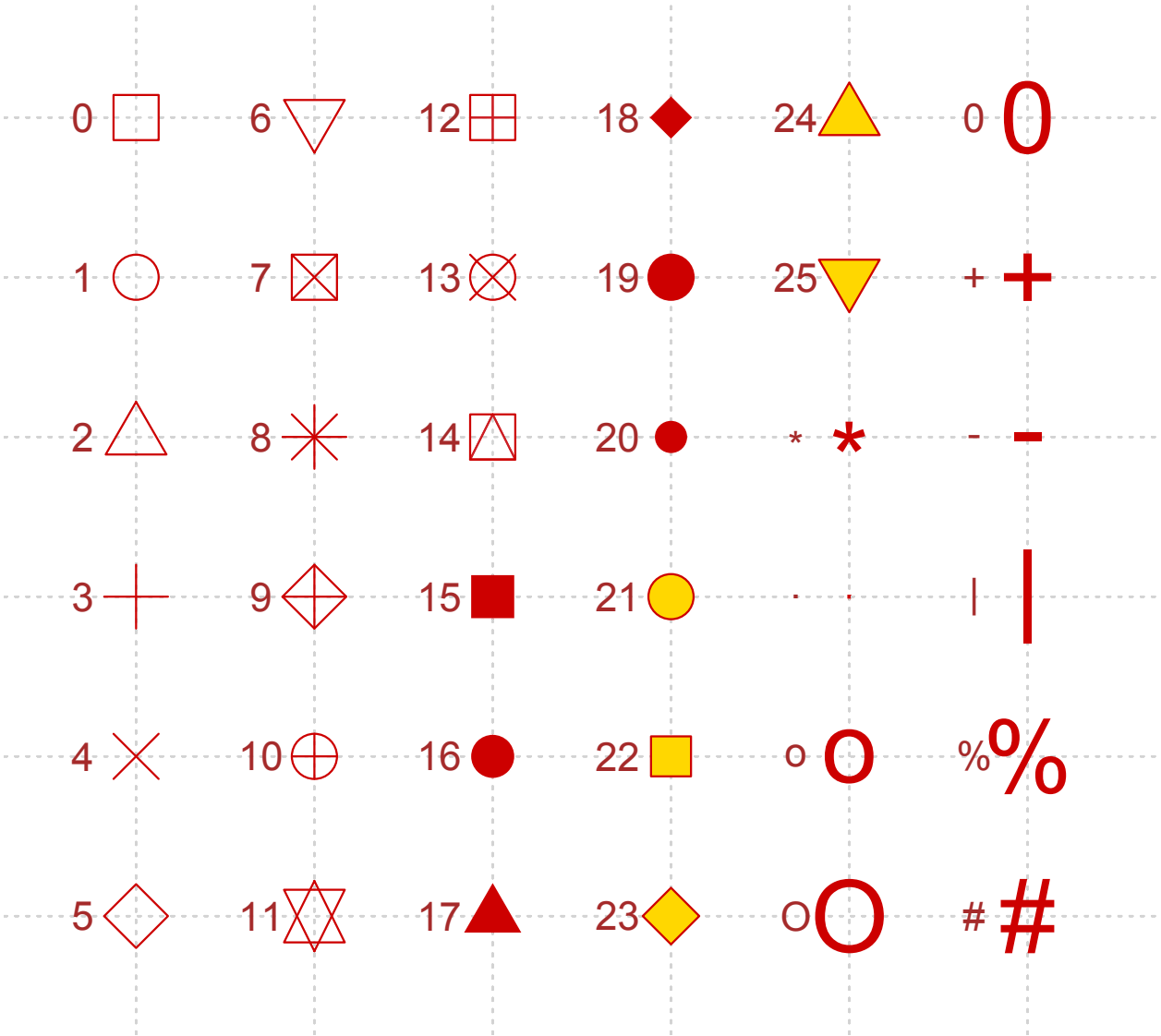


Colored regions in R are controlled by `fill`*, not* `color`

Note we had to move the `fill` *control into the* `aes` *argument of* `geom_smooth`

Also needed to manually set its `scale`

Aside on a feature we didn't use: Varying the plotting symbol



R has surprisingly limited built-in support for various glyphs

All packages share these common codes for symbols

Set via `pch=` in `grid`, `tile`, and `base`, and via `shape=` in `ggplot2`

Often need a vector the length of `x` and `y` with a numeric value indicating the symbol for each point

Low-level graphics systems

If you looked inside `ggplot2` or `tile`, what would you see?
Lots of `grid` graphics code actually making all the plotting elements

`ggplot2` and other high-level graphical systems work on top of lower level graphical system that do all the actual plotting. These systems:

1. create coordinate spaces & scales
2. layout various spaces in which plot elements are placed
3. plot "primitive" graphical elements like text, symbols, lines, and polygons
4. allow customization of these primitives using "graphical parameters"
   e.g., for color, thickness, transparency, and glyph-type
5. provide convenient axis drawing tools

Imagine you had to draw a graph just using these tools:
starting with a blank canvas, you'd assemble primitives to build a complete graphic

Time-consuming, but great for creativity and avoiding unnecessary elements

The grid system

The base graphics system in R is hampered by poor defaults for layout (wasted space) and limited support for small multiples and alternative coordinate systems

It's even hard to plot anything in the margin of a plot that's not an axis title

Paul Murrell created the powerful but complex grid engine to replace base

Developing new graphical software in R? Avoid base completely.
Write it in grid *or a high-level system built on* grid.

Three things you can do with grid:

- Create a "plotting region" (with implicit coordinates & axes) anywhere on the canvas
- Nest these plotting regions, producing a hierarchical graphical object
- Reference (and plot to) points with respect to any plotting region using any system of measurement

Viewports in `grid`

A grid plotting region is called a **viewport**

Some key commands:

`pushViewport()`, `upViewport()`, `downViewport`

What can you do with viewports?

- Create separate plotting regions: Grids of plots
- Fine control of margins
- Plots inside plots
- Even plots inside of plotting symbols

Commonly used units in grid graphics

`native`	Based on the current *x*, *y* scales (e.g., your data)
`npc`	Treats the current viewport as (0,0) to (1,1)
`inches`	This and other physical unit available, given device
`strwidth`	Multiples of the width of a given string
`strheight`	Multiples of the height of a given string
`null`	In layouts, any remaining space is divided among nulls

`grid` needs to know the units of things it plots; instead of plotting `x=0.5, y=0.25`, do:

```
grid.points( x = unit(0.5, "native"), y = unit(0.25, "native") )
```

The last three units in the table above are very powerful

For example, `unit( 1, "strwidth", "this string" )` creates a unit as wide as the text "this string" given the graphics device, viewport, and graphics parameters

Can't `c()` on `unit()` terms. Use `unit.c()` instead

Primitives in the grid system

grid contains no high-level graphics code at all

We can only draw with primitives,
or simple commands that make lines, text, points, polygons, and so on

These include:

```
grid.lines()
grid.polygon()
grid.points()
grid.text()
etc
```

ggplot2, tile, and base graphics commands don't work in grid!
If you try them, you will get strange and unintended results!

Drawing with grid

```
grid.points(x, y,
            pch = 1,
            size = unit(1, "char"),
            default.units = "native",
            name = NULL,
            gp=gpar(),
            vp = NULL)
```

Making a grob with grid

```
g <- pointsGrob(x, y,
                pch = 1,
                size = unit(1, "char"),
                default.units = "native",
                name = NULL,
                gp=gpar(),
                vp = NULL)
```

We can use grid primitives to directly draw on the current graphics device

Or we can use grid to draw a graphics object, or "grob"

The grob won't be plotted until we ask grid.draw() to do so

In the meantime, we can modify it, measure it on a graphics device, or combine it with other grobs

ggplot2 and tile make graphs out of many grobs,
then draw them all at once at the end

`grid` graphics parameters

Customize the appearance of grid graphical objects; see the help for `gpar()` for more details:

col	Colour for lines and borders.
fill	Colour for filling rectangles, polygons, ...
alpha	Alpha channel for transparency
lty	Line type
lwd	Line width
cex	Multiplier applied to fontsize
lineend	Line end style (round, butt, square)
linejoin	Line join style (round, mitre, bevel)
linemitre	Line mitre limit (number greater than 1)
fontsize	The size of text (in points)
fontfamily	The font family
fontface	The font face (bold, italic, ...)
lineheight	The height of a line as a multiple of the size of text

Other important `grid` commands

layout	Makes a layout of viewports
editGrob	Edits an existing graphical object
unit.length	Returns the length of a unit

Let's use grid to plot a regression line with a shaded confidence region

grid is really meant for designing totally new graphics,
and especially for building functions for end-users

Although you wouldn't normally write a plot from scratch in grid,
it's the best way to learn how grid works

Let's use grid to plot a regression line with a shaded confidence region

grid is really meant for designing totally new graphics,
and especially for building functions for end-users

Although you wouldn't normally write a plot from scratch in grid,
it's the best way to learn how grid works

We begin by loading libraries and helper functions...

```
# Clear memory of all objects
rm(list=ls())

# Load libraries
library(RColorBrewer)       # For nice colors
library(grid)               # For graphics
```

```
# MM-estimator fitting
mmest.fit <- function(y, x, ci, quiet=TRUE) {
    require(MASS)
    y <- y[order(x)]
    x <- x[order(x)]
    result <- rlm(y~x, method="MM")
    if (!quiet) print(result)
    fit <- list(x=x)
    fit$y <- result$fitted.values
    fit$lower <- fit$upper <- NULL
    if (length(na.omit(ci))>0)
        for (i in 1:length(ci)) {
            pred <- predict(result, interval="confidence", level=ci[i])
            fit$lower <- cbind(fit$lower, pred[,2])
            fit$upper <- cbind(fit$upper, pred[,3])
        }
    fit
}
```

```
# Here's an effort at a color lightener (could use work)
lighten <- function(col,
                    pct=0.75,
                    alpha=1){
    if (abs(pct)>1) {
        warning("invalid pct in lighten(); must be between 0 and 1.")
        pcol <- col2rgb(col)/255
    } else {
        col <- col2rgb(col)/255
        if (pct>0) {
            pcol <- col + pct*(1-col)
        } else {
            pcol <- col*pct
        }
    }
    rgb(pcol[1], pcol[2], pcol[3], alpha)
}
```

Build a graphic using `grid` — *Initialize device & make layout*

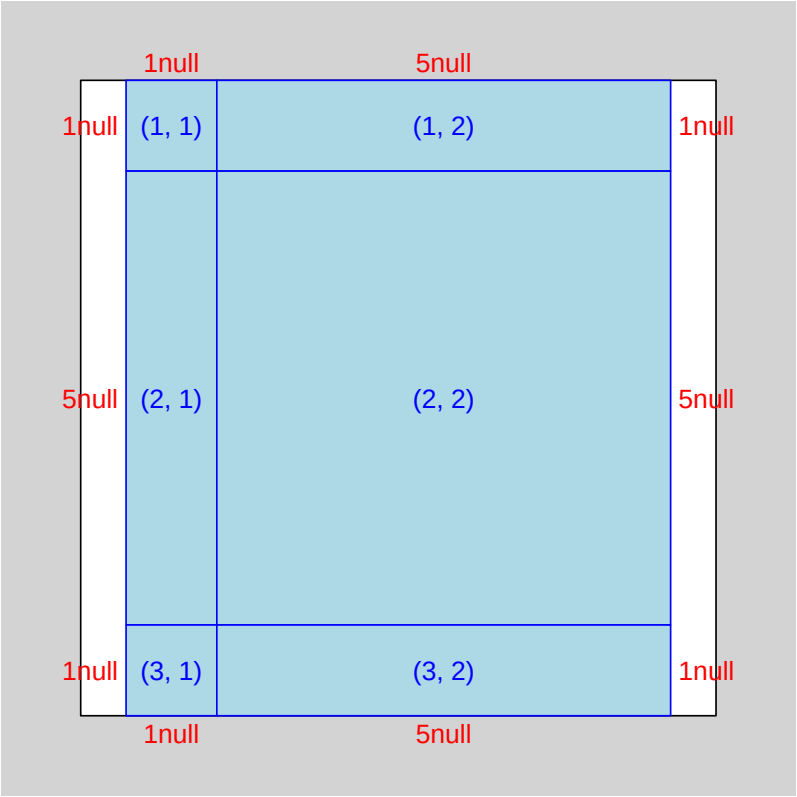
```
# Open a new pdf device
pdf("testgrid.pdf", width=5, height=5)

# Set up plot limits (used later)
limits <- c(log(1.5), log(8), 20, 100)

# Set up tick locations (used later)
yat <- c(20, 40, 60, 80, 100)
xat <- c(2, 3, 4, 5, 6, 7)

# Establishing limits and tick locations
# up front makes them easier to change
# -- otherwise, you need to make lots of edits
```

PDF output is blank so far:
Nothing plotted yet

Build a graphic using grid

Initialize device & make layout

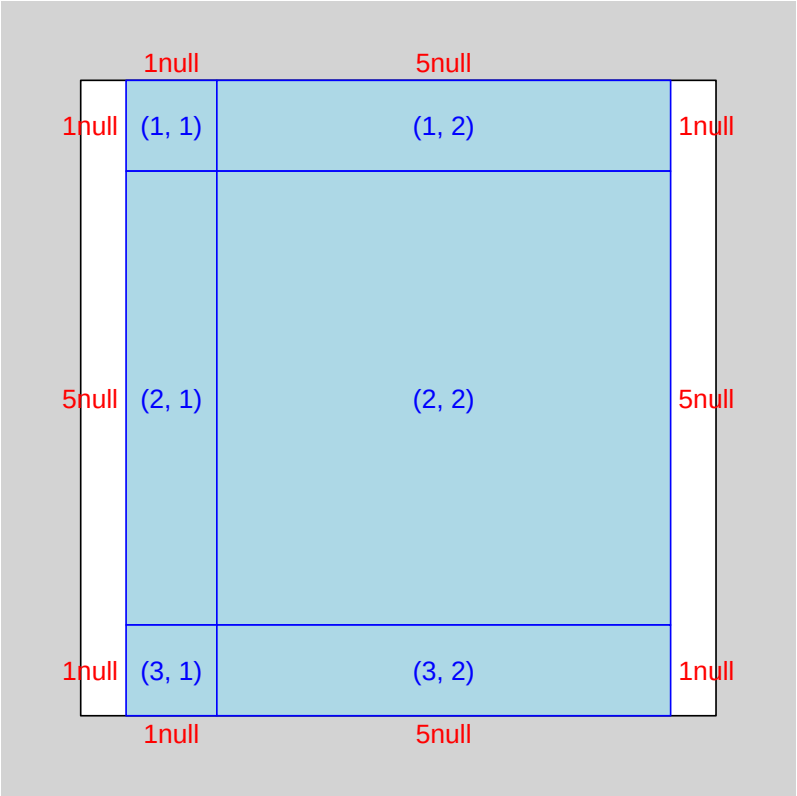
```
# Create layout
overlay <- grid.layout(nrow=3,
                       ncol=2,
                       widths=c(1, 5),
                       heights=c(1, 5, 1),
                       respect=TRUE)

# Display layout
grid.show.layout(overlay)
```



Diagnostic plot of `overlay` layout

Build a graphic using `grid` — *Initialize device & make layout*

```
# Create layout
overlay <- grid.layout(nrow=3,
                       ncol=2,
                       widths=c(1, 5),
                       heights=c(1, 5, 1),
                       respect=TRUE)

## Display layout
#grid.show.layout(overlay)

# "Push" layout to create initial viewport
pushViewport(viewport(layout=overlay))
```

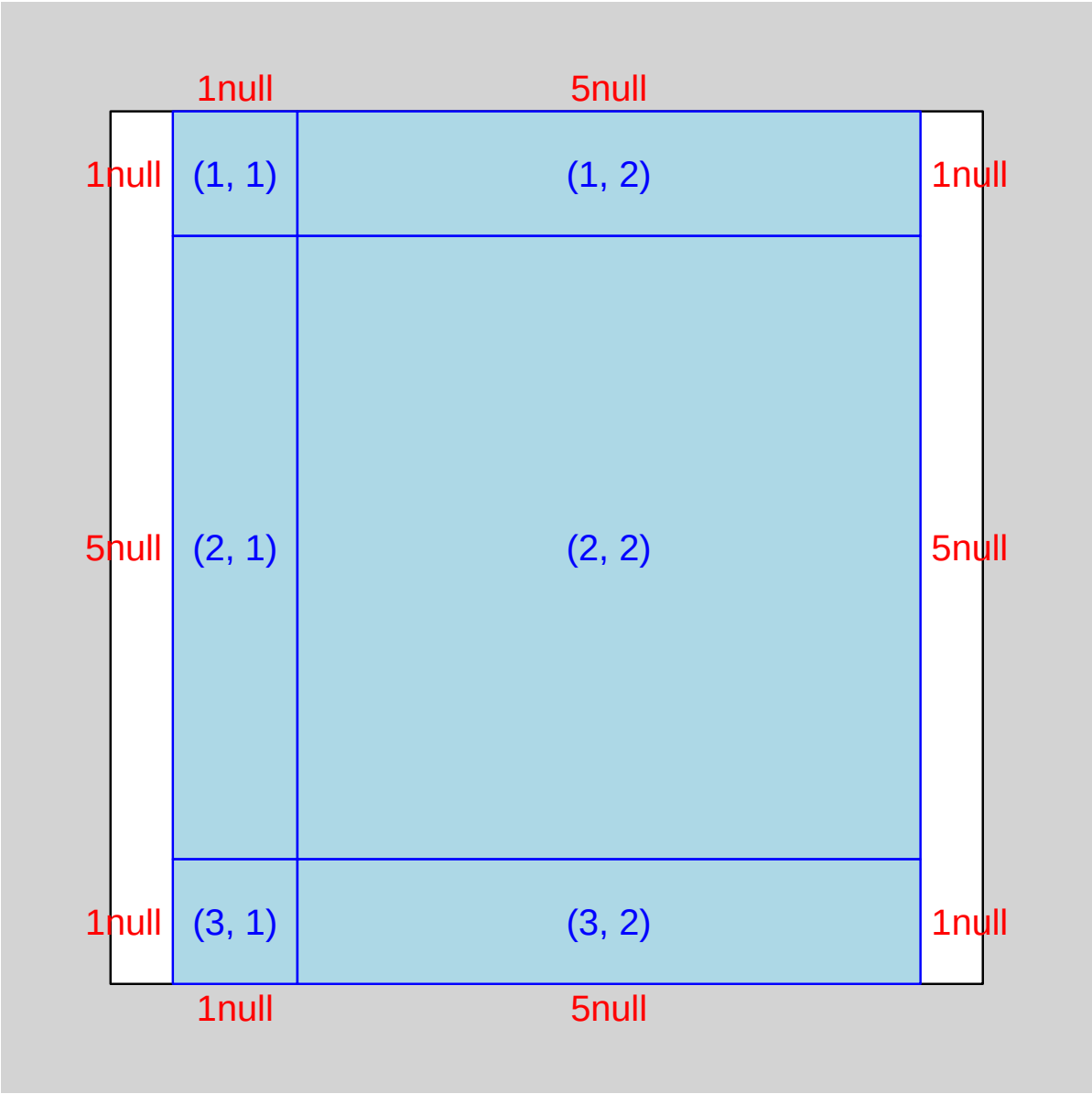


Diagnostic plot of `overlay` layout

If you make a `grid.layout()` you can take a look at it with `grid.show.layout()`

Don't confuse `grid.layout()` with the base command `layout()`!

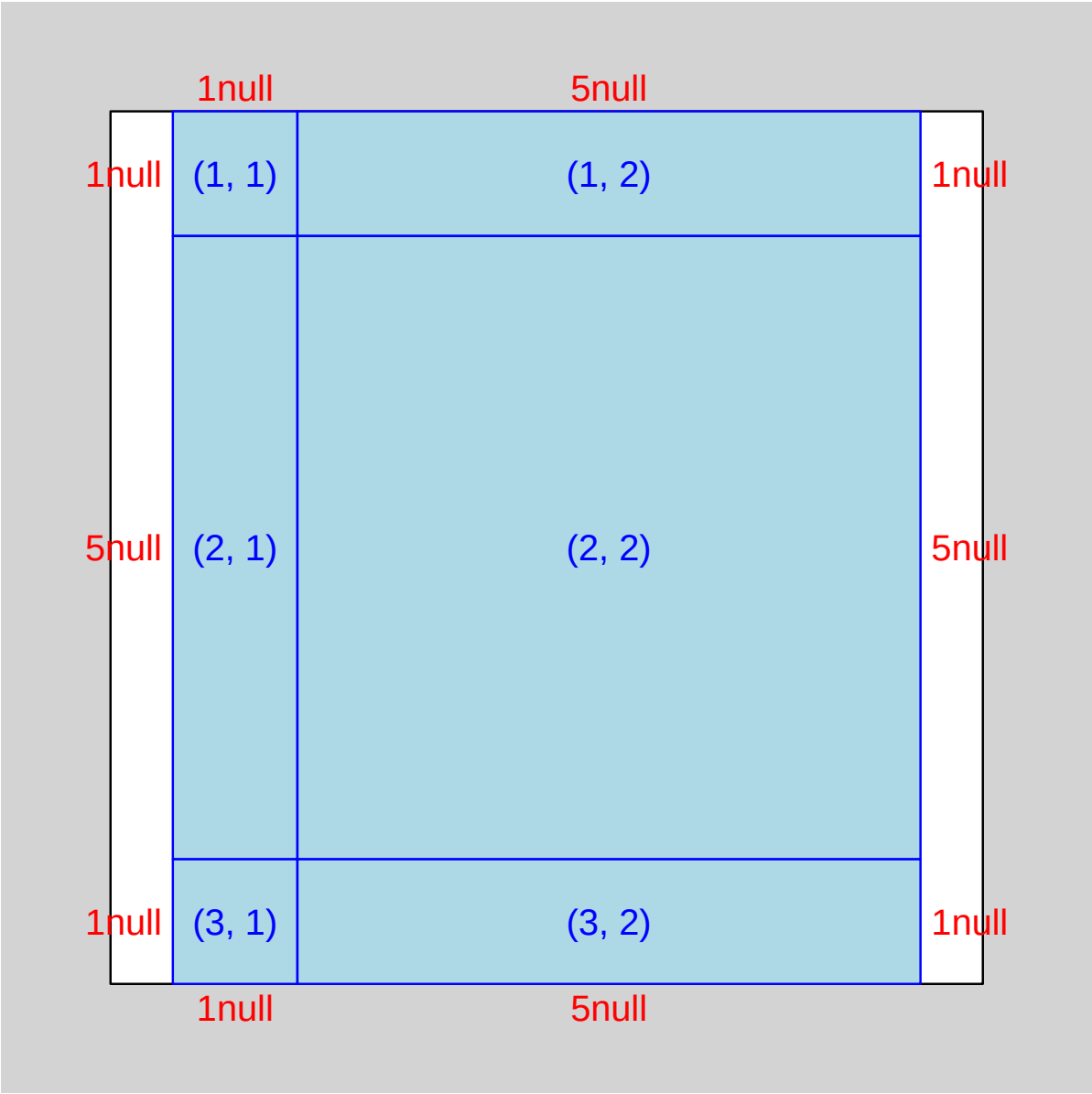
Let's use this diagnostic plot to understand `null` units better...



This graphic is 7 `null`'s high and 6 `null`'s wide

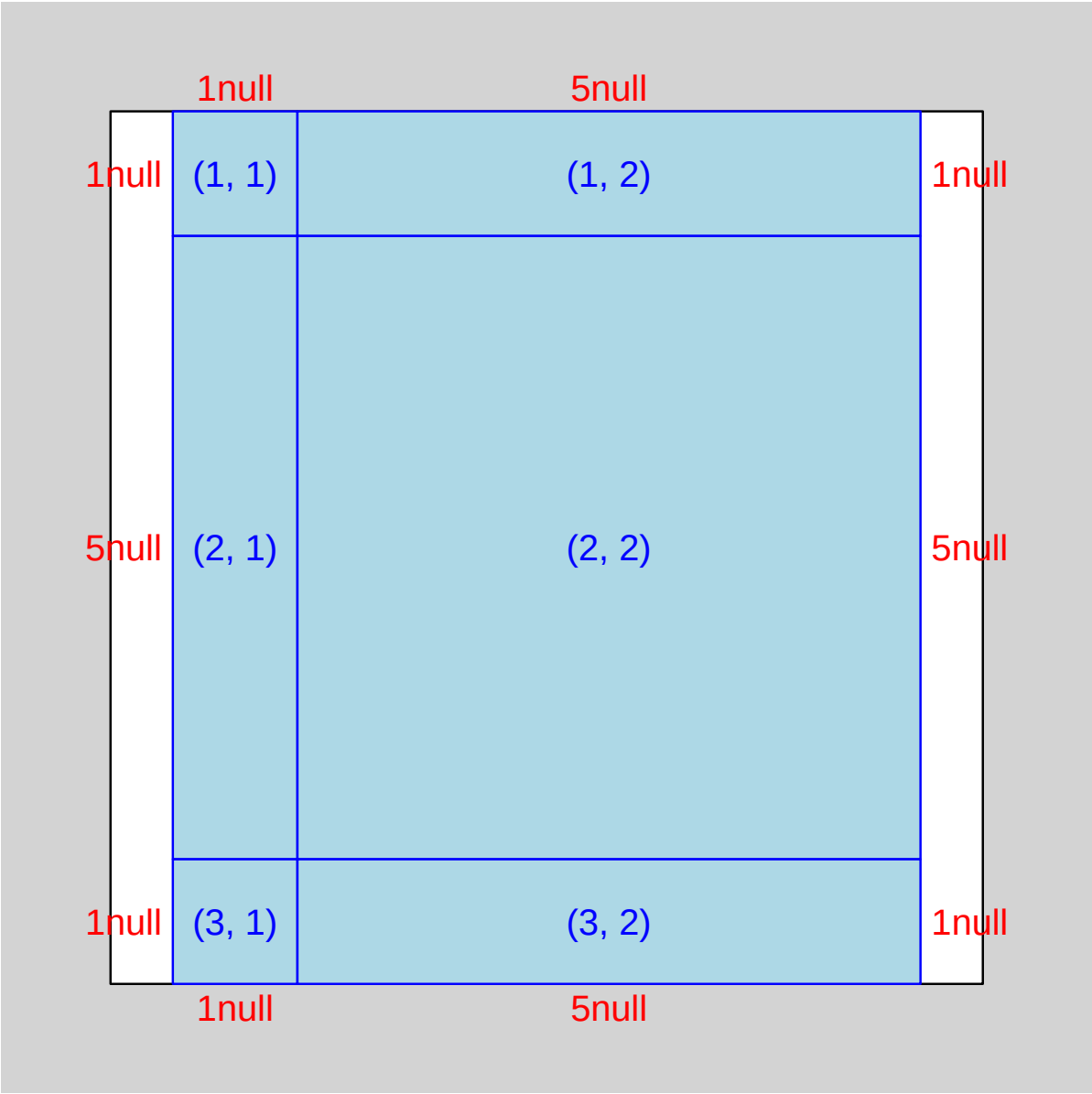
After allocating any fixed units (like `inches`), grid will assign remaining space to `null`

Thus the size of 1 `null` varies based on the fixed content of the graph



Recall this graphic exists in a pdf device that is 5 inches $\times$ 5 inches

So how big is a null in inches?

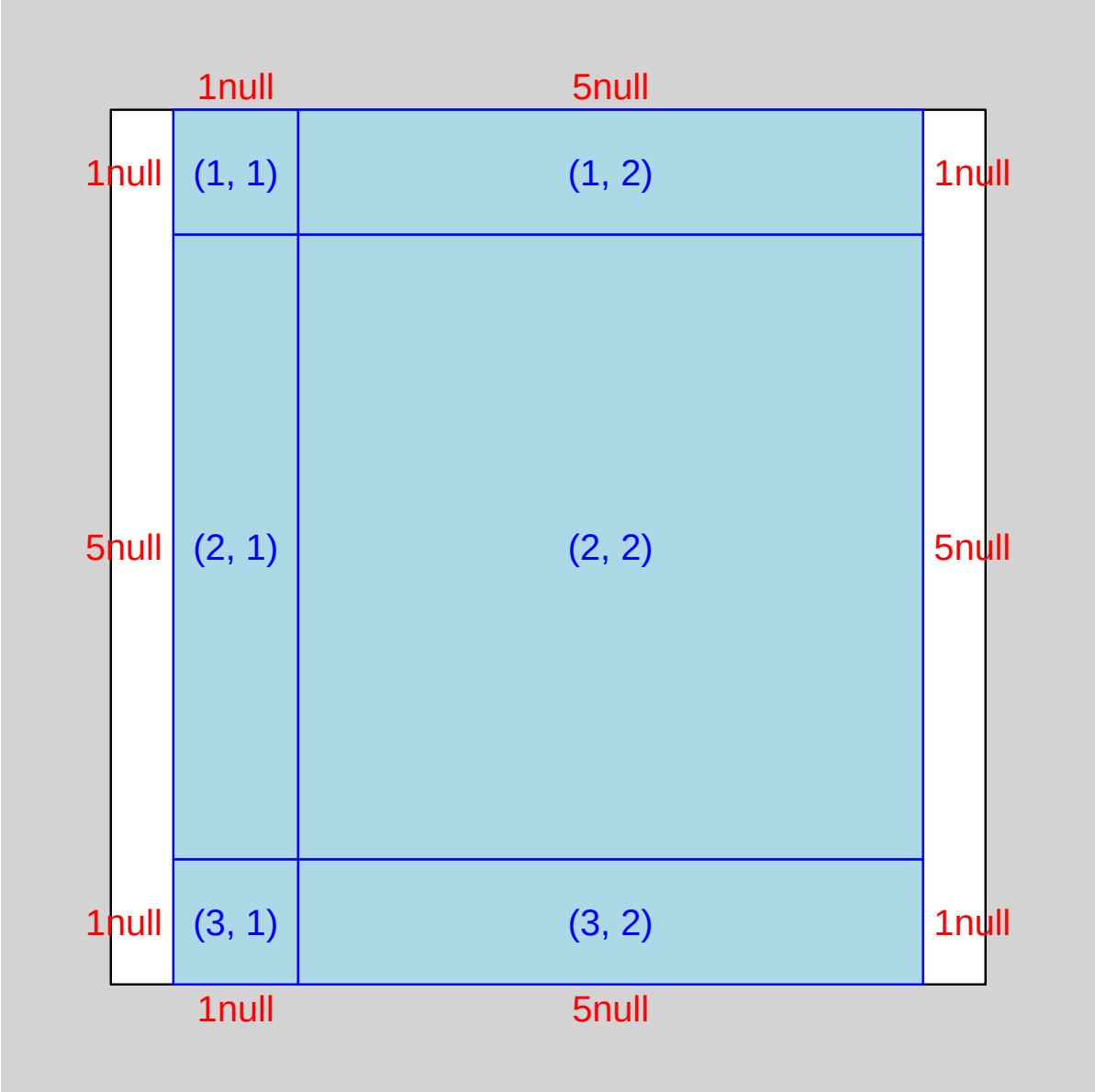


Recall this graphic exists in a `pdf` device that is 5 inches $\times$ 5 inches

So how big is a null in inches?

The graph is 7 `null`'s tall, so 7 `null` = 5 inches

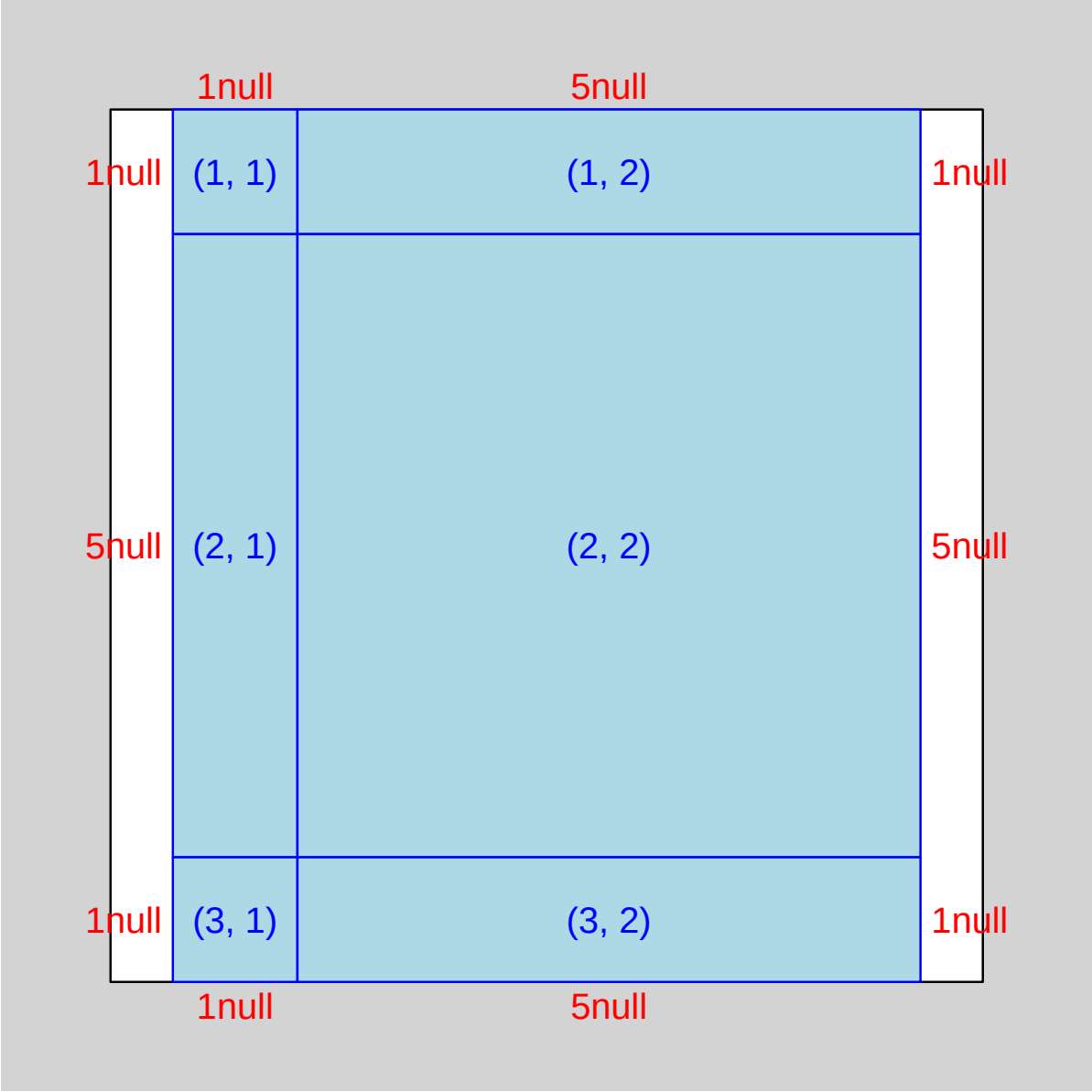
Thus 1 `null` = 5/7 inches ($\approx$0.714 inches)



Why didn't we set the width of 6 nulls equal to 5 inches?

In our layout, we set `respect=TRUE`, which forces `null` widths and `null` heights to be equal

Where nulls might differ, this forces the tightest dimension to "bind" them both to the smaller of the two possible lengths

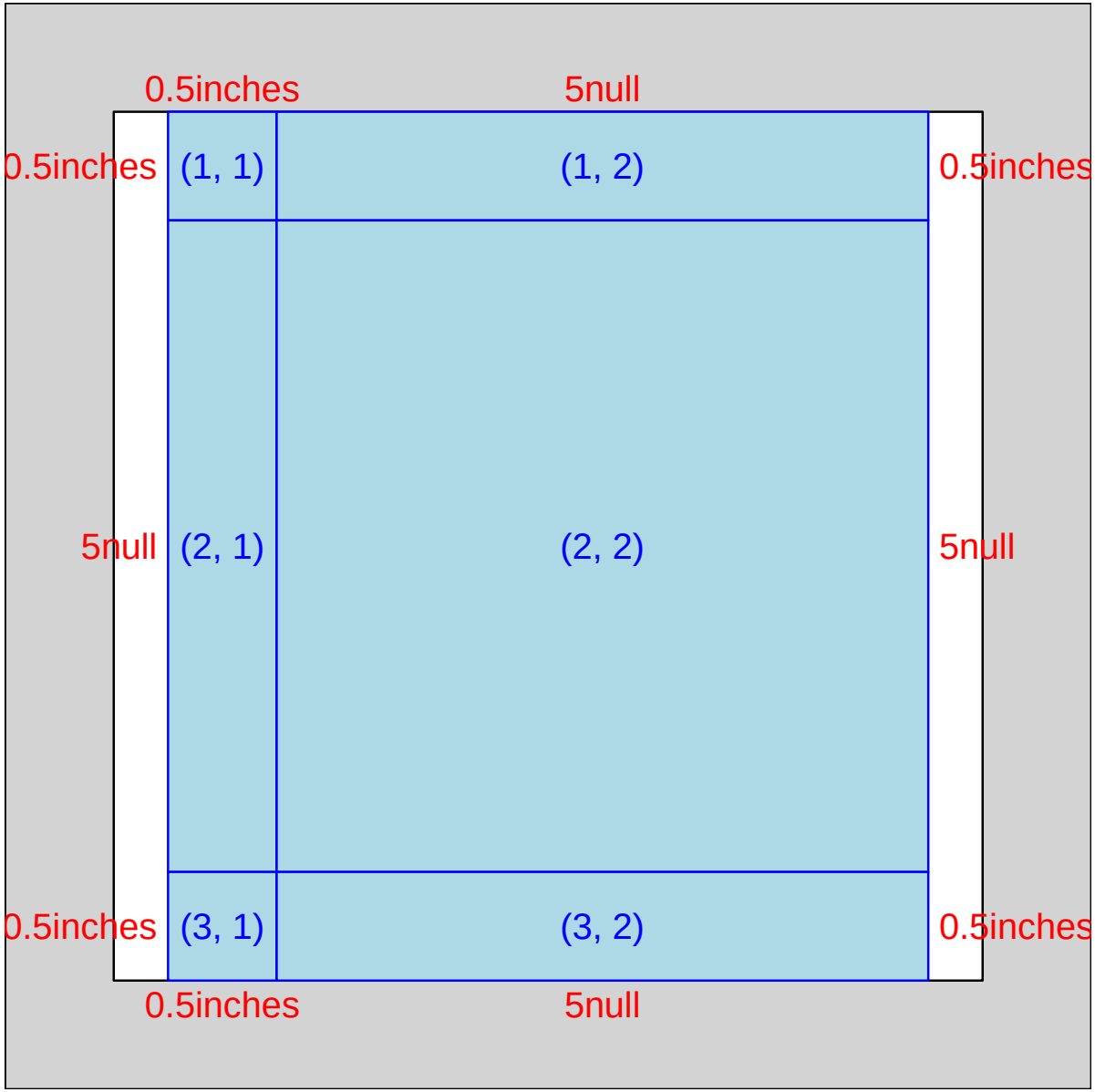


To better understand `null`'s, change the layout so the first column is 0.5 inches wide...

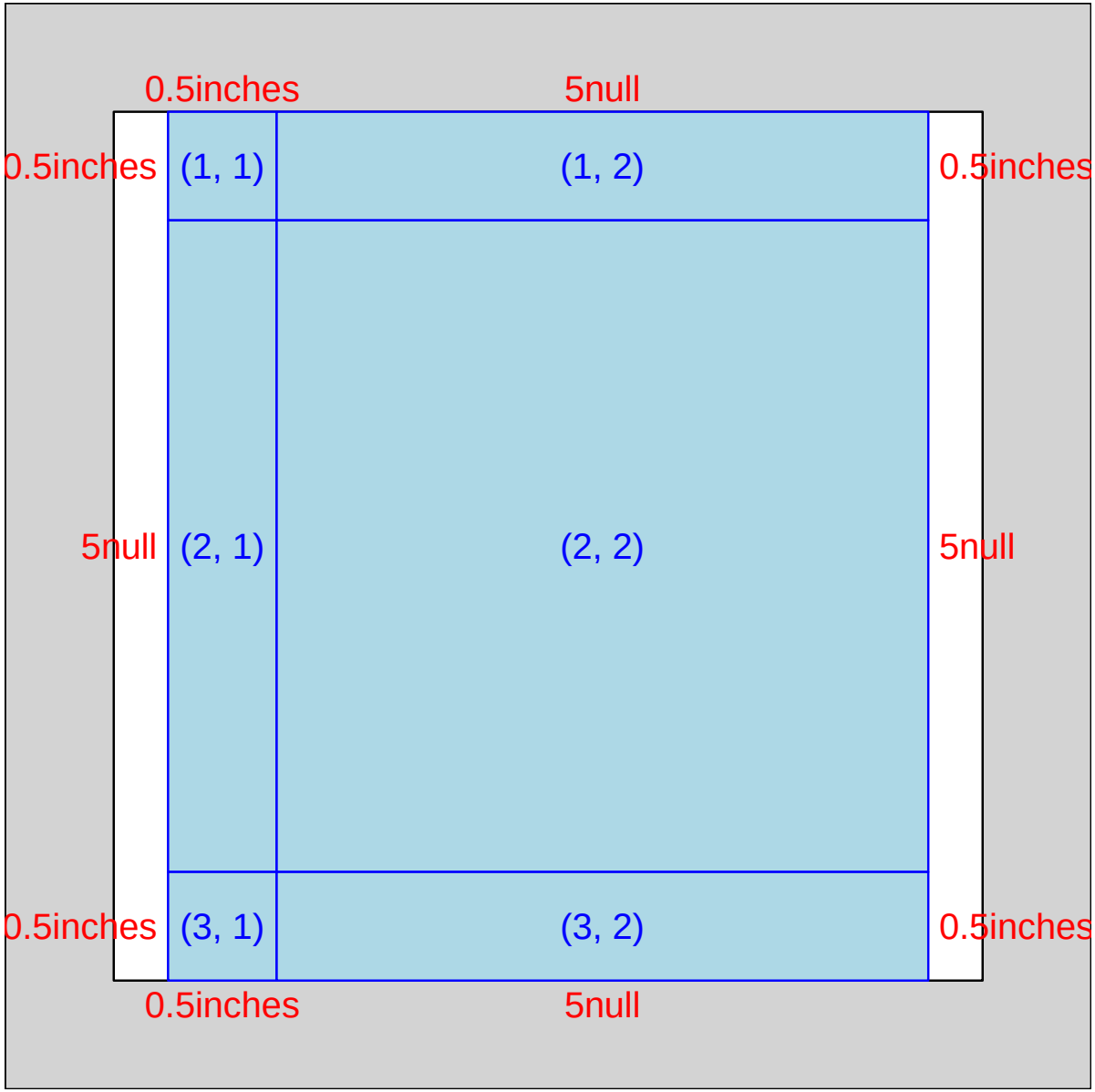
...and the first and last rows are 0.5 inches tall

Why? Because these are spaces for text labels

Fixing to 0.5 inch is one (inelegant) way to ensure they're readable & not too big



What is the size of a `null` in inches now?



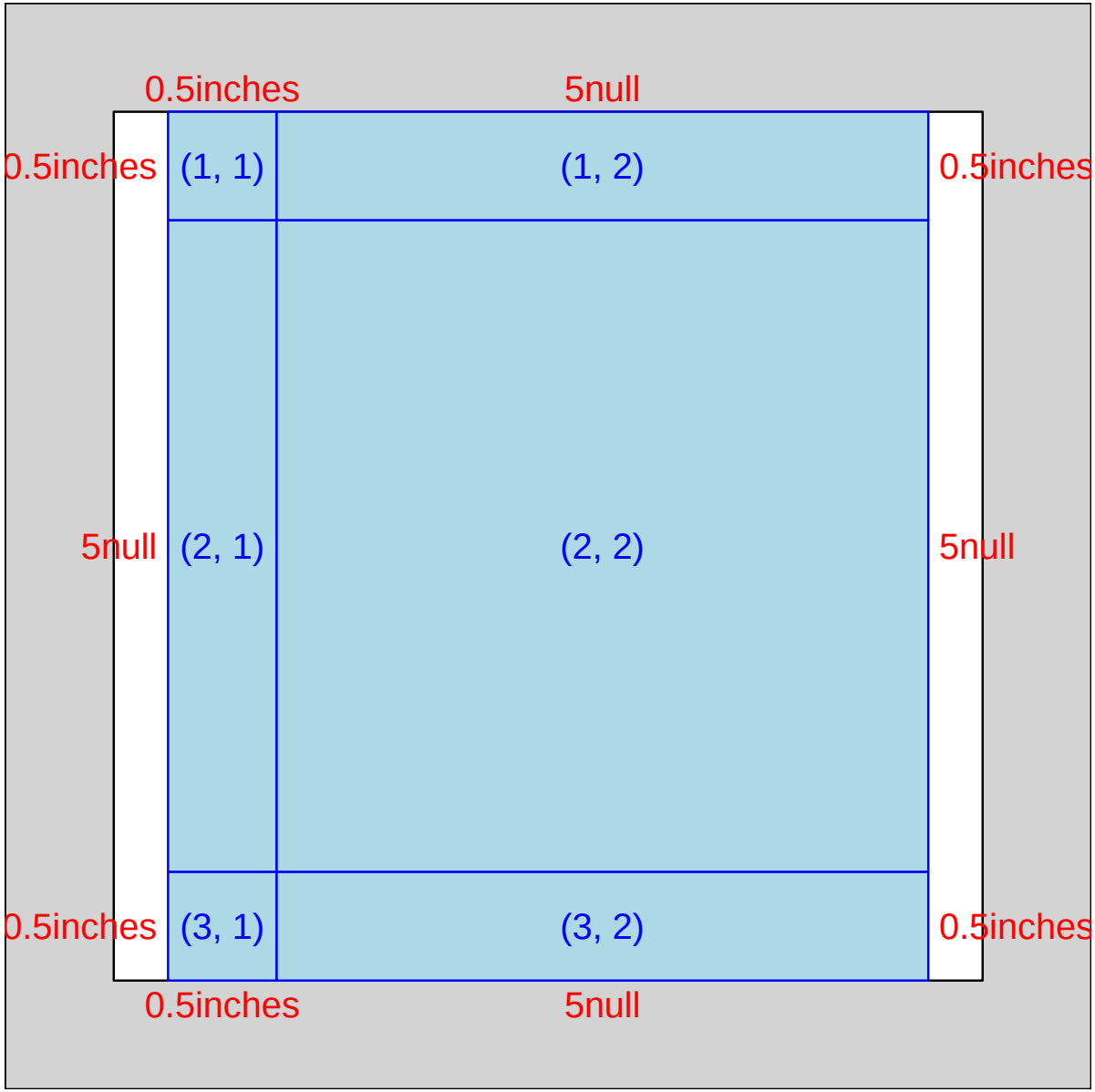
What is the size of a `null` in inches now?

Height is still the binding dimension

Layout is 1 inch and 5 `null`'s tall, and must fit a 5 inch tall pdf

This leaves 4 inches to devote to `null`'s,
so 4 inches = 5 null

A `null` is now 4/5 inches (0.8 in), increasing the space of our graphic devoted to data

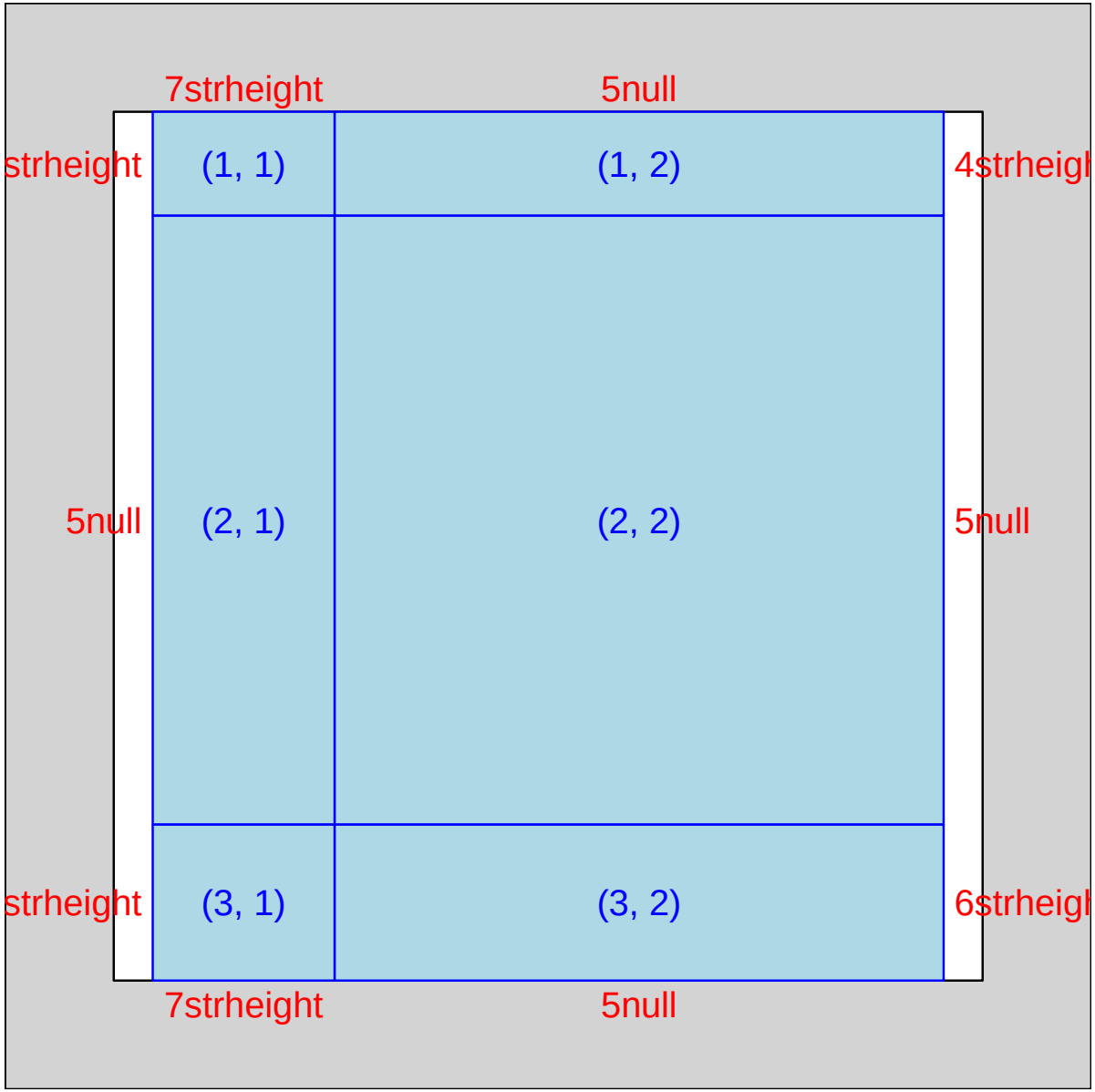


A still more powerful grid unit allows us to tile layouts to the heights and widths of specific strings

We can snugly fit text areas around the text as plotted in the device

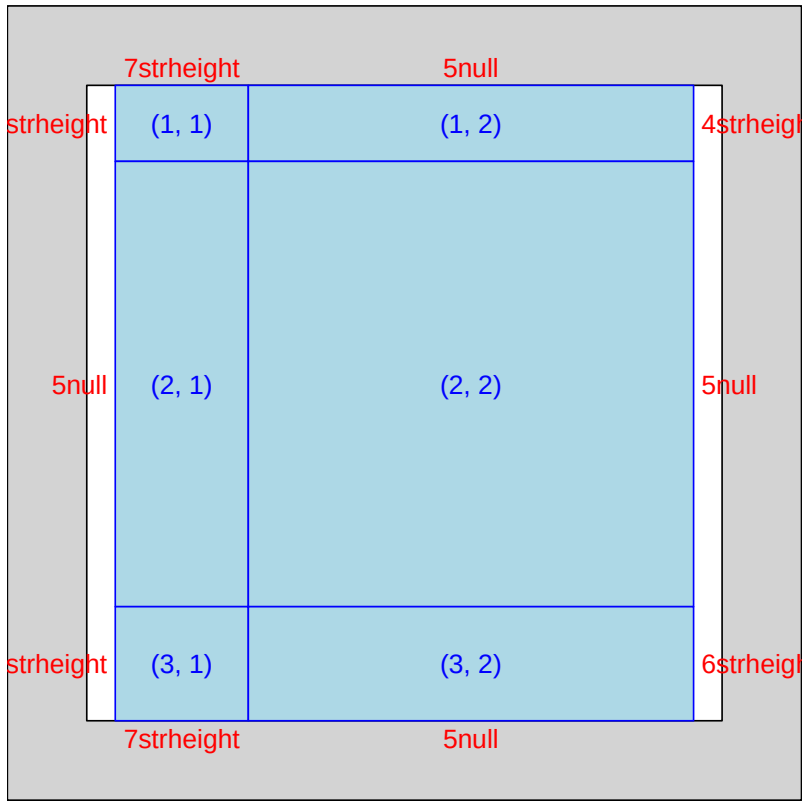
...Leaving the maximum possible space for the graphic

Critical for tiling small multiples effectively – why base does this badly



Build a graphic using grid — *Initialize device & make layout*

```
# Strings for titles
mainTitle <- "Avoid plot titles, except for small multiples"
yaxisTitle <- "Poverty Reduction"
xaxisTitle <- "Effective Number of Parties"
# Set up layout widths
wd <- unit.c(unit(7,"strheight", yaxisTitle),
             unit(5,"null"))
# Set up layout heights
ht <- unit.c(unit(4,"strheight", mainTitle),
             unit(5,"null"),
             unit(6,"strheight", xaxisTitle))
# Create layout
overlay <- grid.layout(nrow=3,  ncol=2,
                       widths=wd,  heights=ht,
                       respect=TRUE)
# "Push" layout to create initial viewport
pushViewport(viewport(layout=overlay))
```



Diagnostic plot of *revised* layout

Build a graphic using grid

Plot the main title

```
# Push the main title viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=1,
                      xscale=c(0,1),
                      yscale=c(0,1),
                      gp=gpar(fontsize=12),
                      name="maintitle",
                      clip="on"
                      ))

# Note the use of a grid primitive
grid.text(mainTitle,
          x=unit(0.5,"npc"), # Why NPC?
          y=unit(0.5,"npc"),
          gp=gpar(fontface="bold")
          )

# Go back up to the top level Viewport
upViewport(1)
```

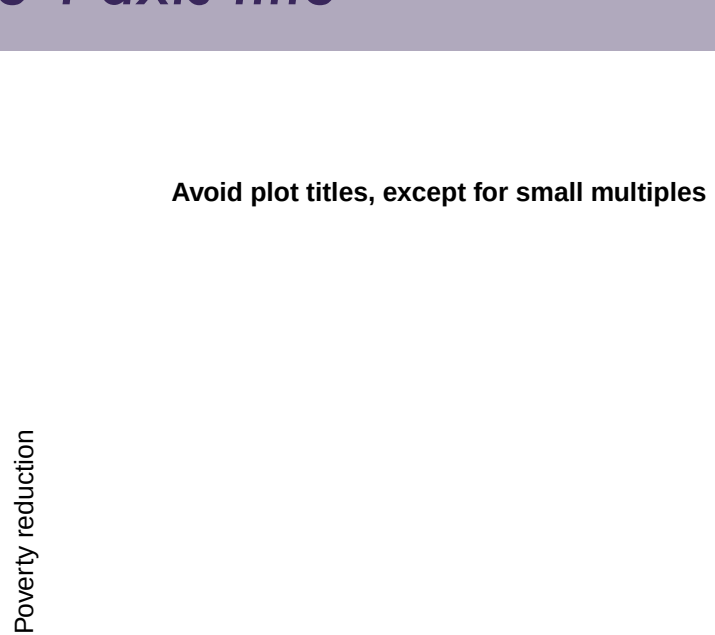
Avoid plot titles, except for small multiples

Our plot so far...

Build a graphic using grid

Plot the Y-axis title

```
# Push the Y-axis title viewport
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2,
                      xscale=c(0, 1),
                      yscale=c(0, 1),
                      gp=gpar(fontsize=12),
                      name="ytitle",
                      clip="on"
                      ))

grid.text(yaxisTitle,
          x=unit(0.15, "npc"),
          y=unit(0.5, "npc"),
          rot=90
          )

upViewport(1)
```

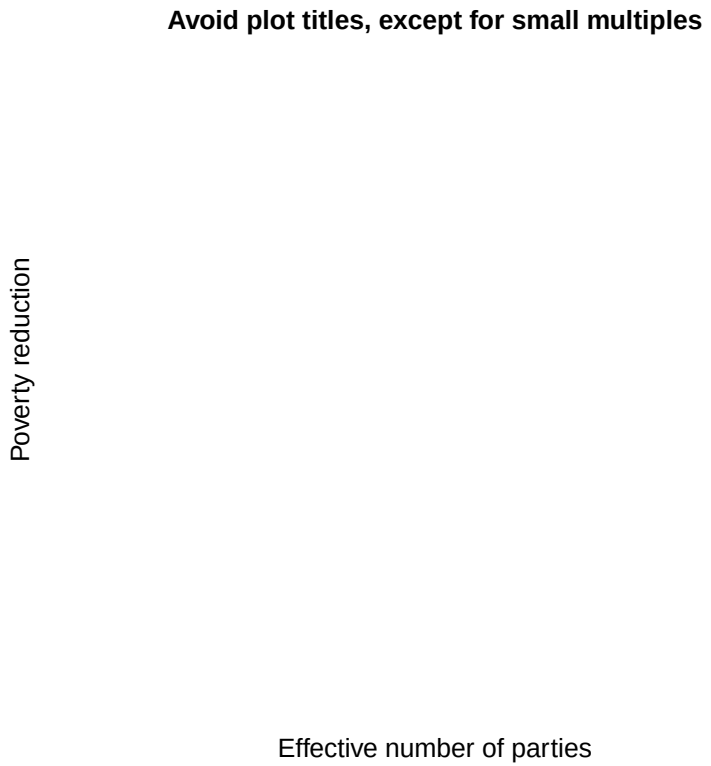


Our plot so far...

Build a graphic using grid

Plot the X-axis title

```
# Push the X-axis title viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=3,
                      xscale=c(0, 1),
                      yscale=c(0, 1),
                      gp=gpar(fontsize=12),
                      name="xtitle",
                      clip="on"
                      ))

grid.text(xaxisTitle,
          x=unit(0.5, "npc"),
          y=unit(0.25, "npc")
          )

upViewport(1)
```



Our plot so far...

Build a graphic using grid *Plot the data*

```
# Push the main plot Viewport. Note the scales
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2,
                      xscale=c(limits[1], limits[2]),
                      yscale=c(limits[3], limits[4]),
                      gp=gpar(fontsize=12),
                      name="mainplot",
                      clip="on"
                      )
             )

# get the fit from the data
fit <- mmest.fit(y=data$povertyReduction,
                 x=log(data$effectiveParties),
                 ci=0.95)
```

Avoid plot titles, except for small multiples

Poverty reduction

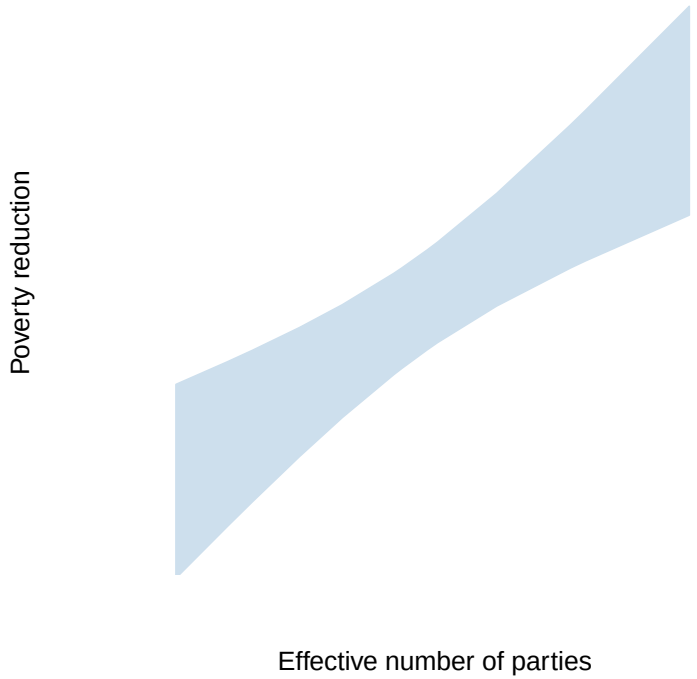
Effective number of parties

Our plot so far...

Build a graphic using grid — *Plot the data*

```
# Make the x-coord of a CI polygon
xpoly <- c(fit$x,  rev(fit$x),  fit$x[1])

# Make the y-coord of a CI polygon
ypoly <- c(fit$lower, rev(fit$upper), fit$lower[1])

# Pick a nice blue
niceblue <- brewer.pal(3,"Set1")[2]

# Choose a pastel color for the polygon
pastelblue <- lighten("blue")

# Plot the polygon first
grid.polygon(x=xpoly,
             y=ypoly,
             gp=gpar(col=pastelblue,
                 border=FALSE,
                 fill=pastelblue),
             default.units="native")
```

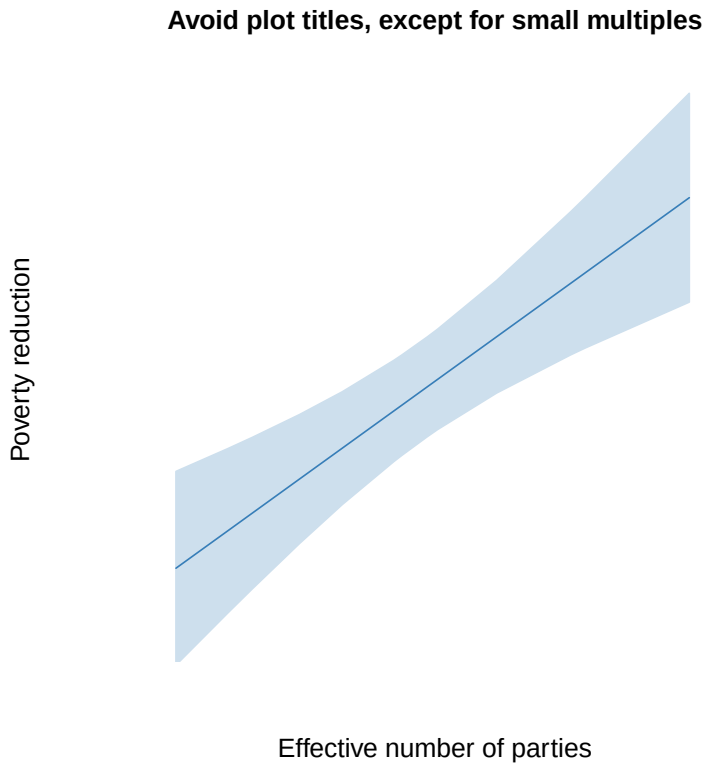
Avoid plot titles, except for small multiples



Our plot so far...

Plot the data

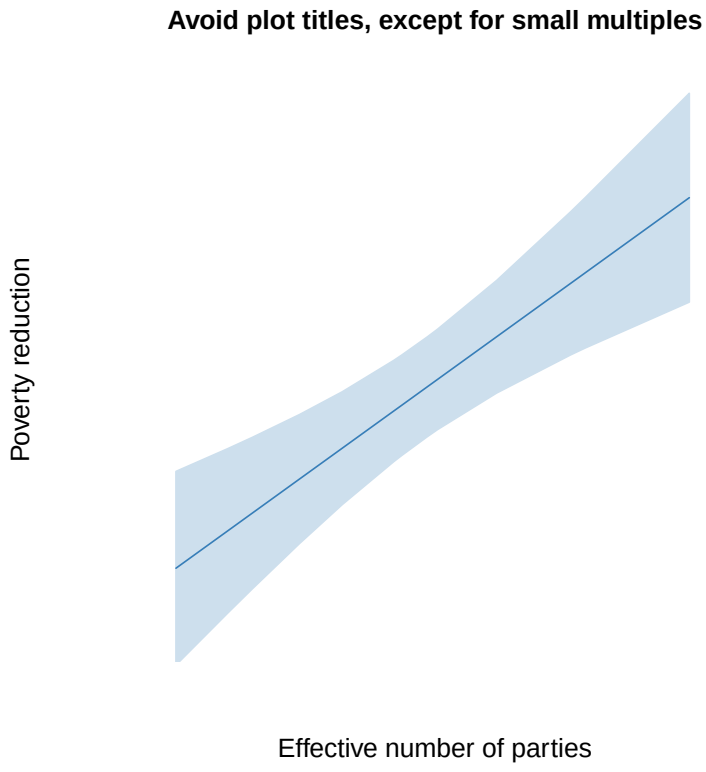
```
# Then plot the line
grid.lines(x=fit$x,
           y=fit$y,
           gp=gpar(lty="solid",
               col=niceblue),
           default.units="native")

# If you want a box around the graph,
# now's the time to plot it
#grid.rect(gp=gpar(linejoin="round"))

# Return to layout level
upViewport(1)
```



Our plot so far...

Build a graphic using grid — *Add the axes*

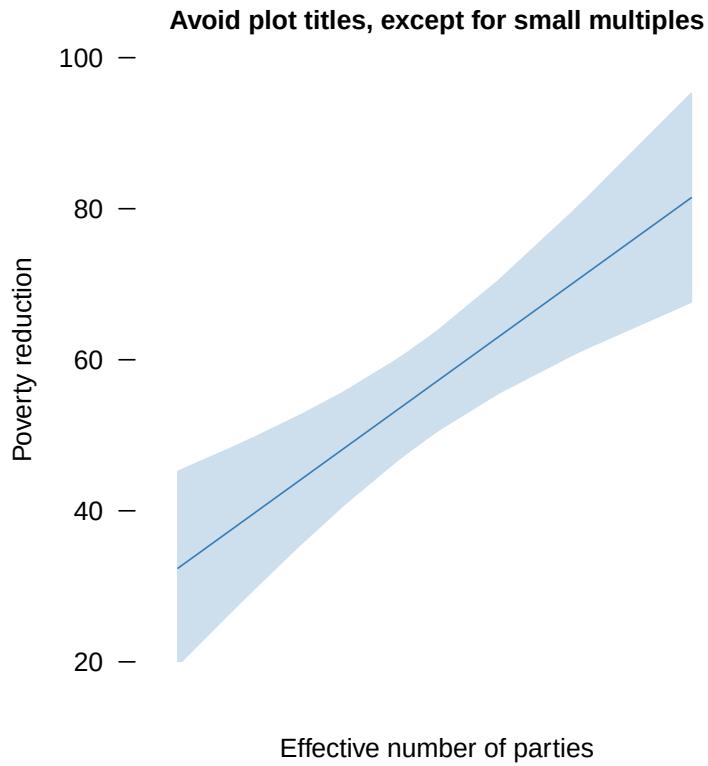
```
# Wait! We haven't made axes!
# Axes plot facing out of the Viewport
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2,
                      xscale=c(limits[1], limits[2]),
                      yscale=c(limits[3], limits[4]),
                      gp=gpar(fontsize=12),
                      name="mainplot",
                      clip="off"
                      ))

# Make axis as a "grob" (graphical object),
# but don't plot yet
yaxis <- yaxisGrob(at = yat,     # Where to put ticks
                   label = TRUE, # Only takes logical values!
                   main = TRUE   # Top (TRUE) or bottom (FALSE)
                   )
```



Our plot so far...

Build a graphic using grid — *Add the axes*

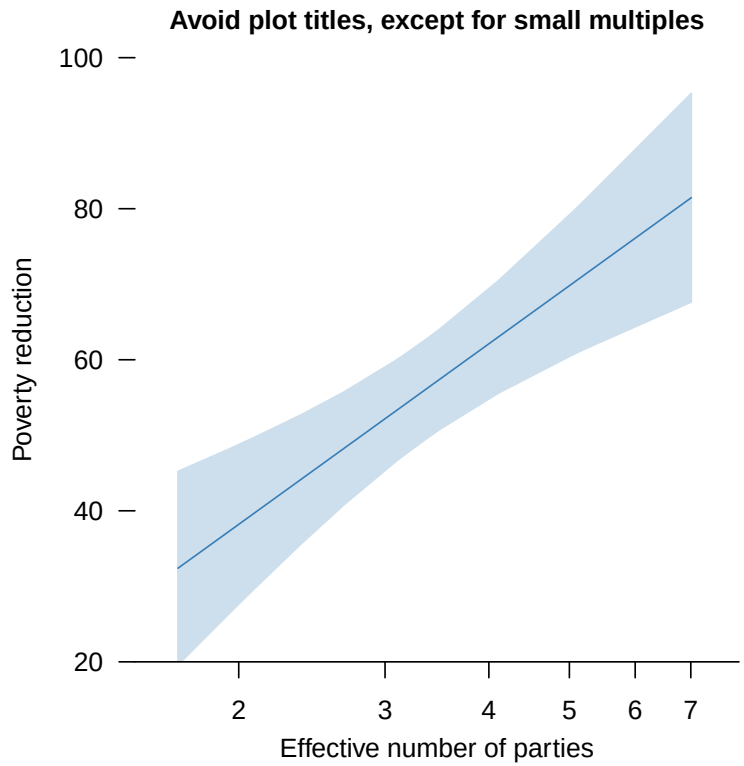
```
# Edit the y-axis to remove the major line
yaxis <- editGrob(yaxis,
                  gPath("major"),
                  major=FALSE
                  )

# Edit the y-axis to remove the major line
yaxis <- removeGrob(yaxis, gPath("major"))

# Draw the y-axis grob
grid.draw(yaxis)
```



Our plot so far...

Build a graphic using grid — *Add the axes*

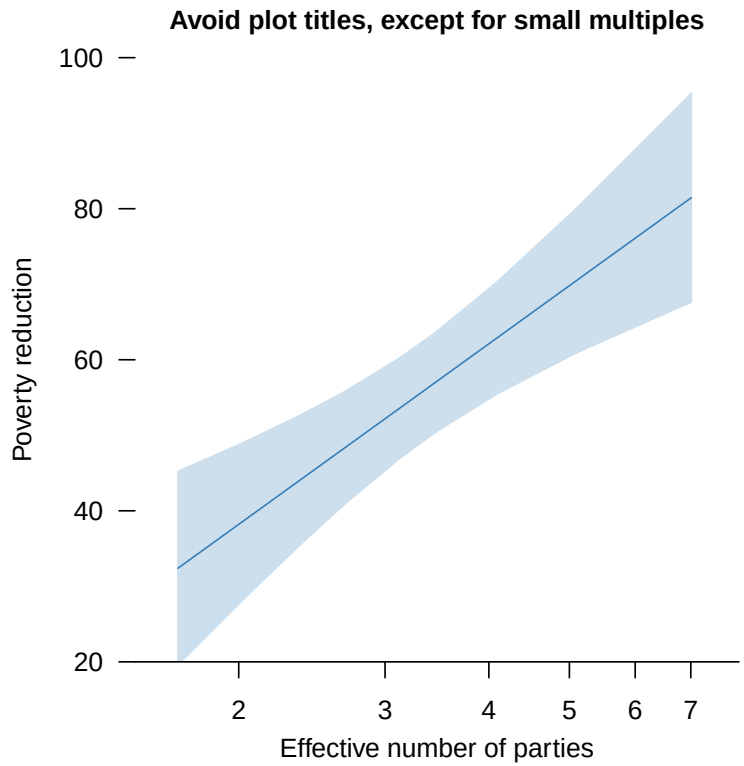
```
# Recall that our tick locations are:
# xat <- c(2, 3, 4, 5, 6, 7)
# Make an x-axis as a "grob"
xaxis <- xaxisGrob(at = log(xat),
          label = TRUE,
          main = TRUE)
# Edit the x-axis to show unlogged units
xaxis <- editGrob(xaxis,
                  gPath("labels"),
                  label = xat)
# Edit the x-axis to fit the plot
xaxis <- editGrob(xaxis,
                  gPath("major"),
                  x=unit(c(0, 1), "npc")
                  )
# Draw the x-axis grob
grid.draw(xaxis)
```



Our plot so far...

Build a graphic using `grid` — *Save the graphic*

```
# Return to layout viewport
upViewport(1)

# Save the graph!
dev.off()
```



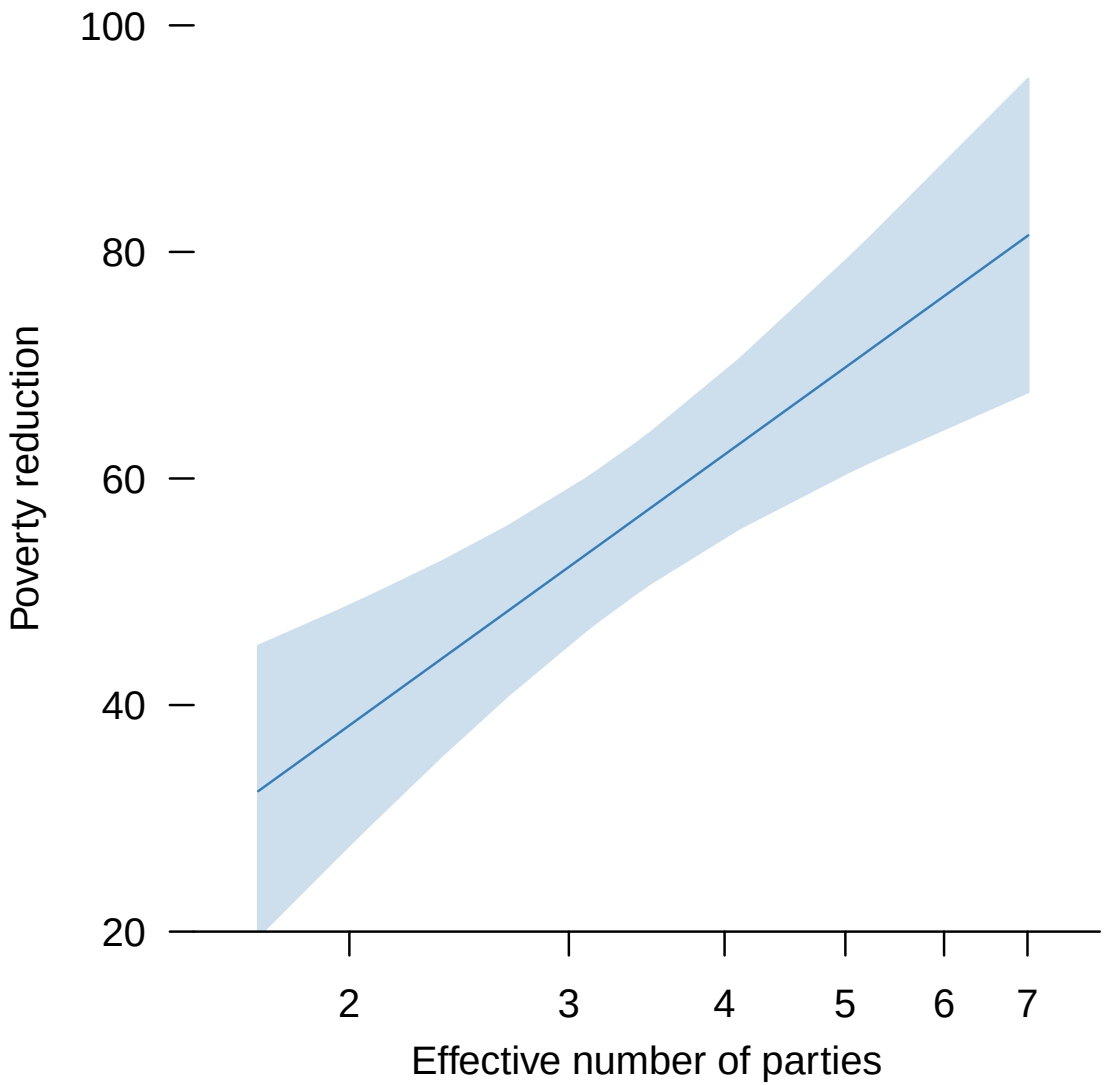
Our plot so far...

`grid` provides detailed control over coordinates, plotting elements, layout, spacing, alignment, etc.

Essential for legible small multiples

Vastly more powerful than base graphics

And vastly more time consuming to make



Suppose we wanted to add more features to the plot, such as replacing the y ticks with thin gridlines?

Turns out to be a lot of work! (Code on course site)

ggplot2 and tile provide users *easy* ways to make these sorts of changes, but ultimatly work through grid code like the above