* First, tell R you’re using ggplot()
* Then, tell it the object name where variables exist (data = df\_name)
* Next, tell it the aesthetics aes() to specify which variables you want to plot
* Then add a layer for the type of geom (graph type) with geom\_\*() - for example, geom\_point() is a scatterplot, geom\_line() is a line graph, geom\_col() is a column graph, etc.

Some common arguments we’ll use first are:

* color = or colour =: update point or line colors
* fill =: update fill color for objects with areas
* linetype =: update the line type (dashed, long dash, etc.)
* pch =: update the point style
* size =: update the element size (e.g. of points or line thickness)
* alpha =: update element opacity (1 = opaque, 0 = transparent)

Sometimes, however, we **do** want the aesthetics of a graph to depend on a variable. To do that, we’ll **map variables onto graph aesthetics**, meaning we’ll change how an element on the graph looks based on a variable characteristic (usually, character or value).

When we want to customize a graph element based on a variable’s characteristic or value, add the argument within aes() in the appropriate geom\_\*() layer. In short, if updating aesthetics based on a variable, make sure to put that argument inside of aes().

**Example:** Create a ggplot scatterplot graph where the **size** and **color** of the points change based on the **number of visitors**, and make all points the same level of opacity (alpha = 0.5). Notice the aes() around the size = and color = arguments.

Also: this is overmapped and unnecessary. Avoid excessive / overcomplicated aesthetic mapping in data visualization.

gg\_base +

geom\_point(

aes(size = visitors,

color = visitors),

alpha = 0.5

)

In the example above, notice that the two arguments that **do** depend on variables are within aes(), but since alpha = 0.5 doesn’t depend on a variable then it is outside the *aes()* but still within the *geom\_point()* layer.

Create a column plot of Channel Islands National Park visitation over time, where the **fill color** (argument: fill =) changes based on the number of **visitors**.

gg\_base +

geom\_col(aes(fill = visitors))

**ggplot2 complete themes**

While every element of a ggplot graph is manually customizable, there are also built-in themes (theme\_\*()) that you can add to your ggplot code to make some major headway before making smaller tweaks manually.

Here are a few to try today (but also notice all the options that appear as we start typing theme\_ into our ggplot graph code!):

* theme\_light()
* theme\_minimal()
* theme\_bw()

Here, let’s update our previous graph with theme\_minimal():

gg\_base +

geom\_point(

aes(size = visitors,

color = visitors),

alpha = 0.5

) +

theme\_minimal()

Use labs() to update axis labels, and add a title and/or subtitle to your ggplot graph.

gg\_base +

geom\_line(linetype = "dotted") +

theme\_bw() +

labs(

x = "Year",

y = "Annual park visitors",

title = "Channel Islands NP Visitation",

subtitle = "(1963 - 2016)"

)

**Note**: If you want to update the formatting of axis values (for example, to convert to comma format instead of scientific format 2e+05), you can use the scales package options.

## Combining compatible geoms

As long as the geoms are compatible, we can layer them on top of one another to further customize a graph.

For example, adding points to a line graph:

gg\_base +

geom\_line(color = "purple") +

geom\_point(color = "orange",

aes(size = year),

alpha = 0.5)

## Multi-series ggplot graphs

In the examples above, we only had a single series - visitation at Channel Islands National Park. Often we’ll want to visualize multiple series. For example, from the ca\_np object we have stored, we might want to plot visitation for all California National Parks.

To do that, we need to add an aesthetic that lets ggplot know how things are going to be grouped. A demonstration of why that’s important - what happens if we don’t let ggplot know how to group things?

ggplot(data = ca\_np, aes(x = year, y = visitors)) +

geom\_line()

Well that’s definitely a mess, and it’s because ggplot has no idea that these **should be different series based on the different parks that appear in the ‘park\_name’ column**.

We can make sure R does know by adding an explicit grouping argument (group =), or by updating an aesthetic based on park\_name:

ggplot(data = ca\_np, aes(x = year, y = visitors, group = park\_name)) +

geom\_line()

## Faceting ggplot graphs

When we facet graphs, we split them up into multiple plotting panels, where each panel contains a subset of the data. In our case, we’ll split the graph above into different panels, each containing visitation data for a single park.

Also notice that any general theme changes made will be applied to all of the graphs.

gg\_np +

geom\_line(show.legend = FALSE) +

theme\_light() +

labs(x = "year", y = "annual visitors") +

facet\_wrap(~ park\_name)

## Exporting a ggplot graph with ggsave()

If we want our graph to appear in a knitted html, then we don’t need to do anything else. But often we’ll need a saved image file, of specific size and resolution, to share or for publication.

ggsave() will export the most recently run ggplot graph by default (plot = last\_plot()), unless you give it the name of a different saved ggplot object. Some common arguments for ggsave():

* width =: set exported image width (default inches)
* height =: set exported image height (default height)
* dpi =: set dpi (dots per inch)

So to export the faceted graph above at 180 dpi, width a width of 8" and a height of 7", we can use:

ggsave(here("figures", "np\_graph.jpg"), dpi = 180, width = 8, height = 7)

Notice that a .jpg image of that name and size is now stored in the figures\ folder within your working directory. You can change the type of exported image, too (e.g. pdf, tiff, eps, png, mmp, svg).

**Sync your project with your GitHub repo.**

* Stage
* Commit
* Pull (to check for remote changes)
* Push!