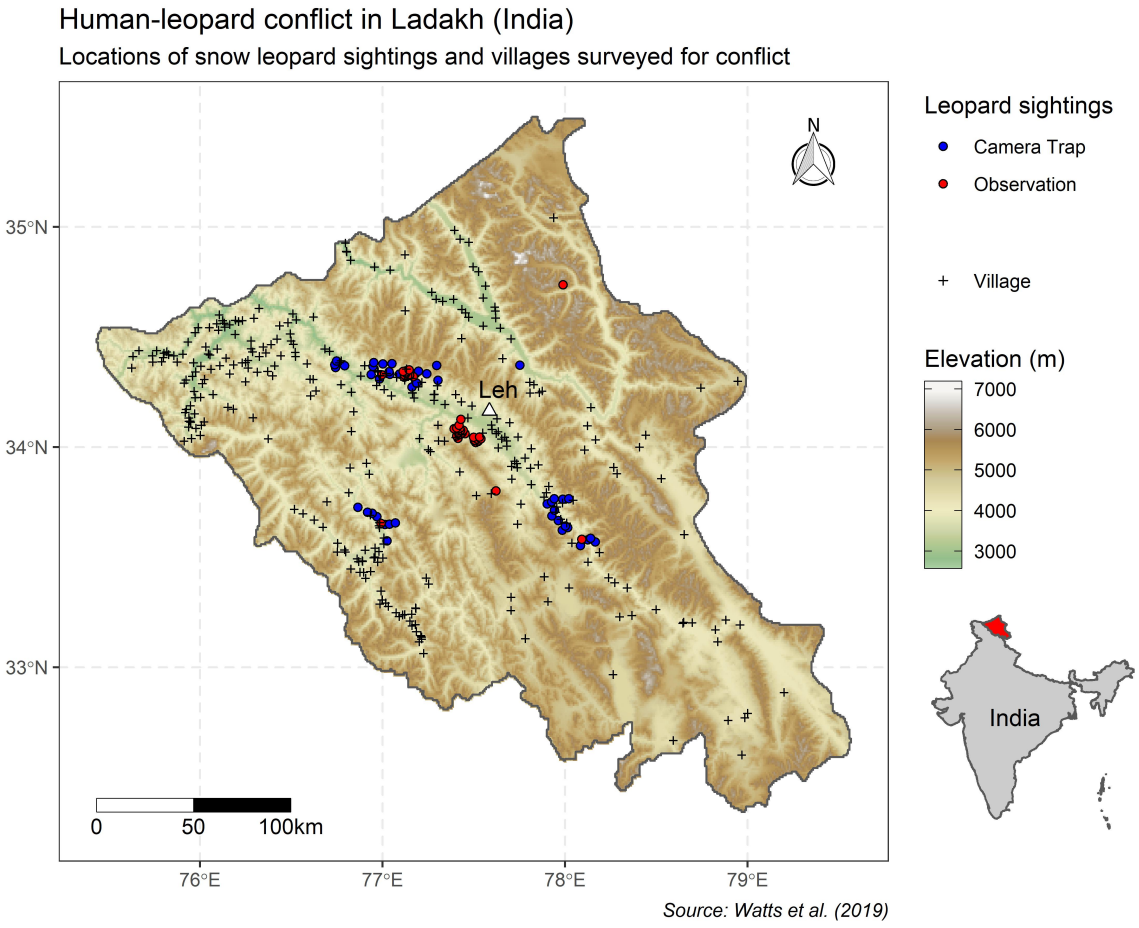
BIOM4031 Tutorial 7: Making maps in R



Introduction

In Tutorials 1-6 you learned how to process, analyse and visualize vector and raster data using R; however the plots and maps we have created so far have been quite 'functional' and lacking many of the key elements we expect from a good map (scale bar, north arrow, informative legend, inset maps, place names etc.). In this tutorial, you will learn how to format, customize and combine maps produced using R to create publication quality figures.

To illustrate, we are going to return to our Ladakh snow leopard study by Watts et al. (2019) introduced in previous tutorials. The overarching goal of this study was to relate snow leopard distributions to indicators of human-wildlife conflict, including locations of villages with recorded livestock predation. We are going to make a map like the one above.

Learning objectives

By the end of the tutorial you will know how to:

1. Customize and style maps created using ggplot
2. Add titles, captions, scale bars and north arrows
3. Combine maps to create multi-panel layouts and insets

Further reading

Additional tutorials on making maps using R and `ggplot2` can be found at the following sources:

"Making Maps with R" by Nico Hahn.

"Drawing beautiful maps programmatically" by Mel Moreno and Mathieu Basille at http://www.r-spatial.org

Data

In the Tutorial 7 data folder you will find the following files:

ladakh.gpkg Administrative boundary of Ladakh from Tutorial 3.

leopards.gpkg Snow leopard sightings data sourced from Watts et al. (2019) and used in Tutorial 3

dem.tif A 90m resolution digital elevation model of Ladakh from Shuttle Radar Topography Mission and accessed through the R package `geodata`. See Tutorial 3.

villages.gpkg Locations and names of villages surveyed by Watts et al. (2019) for evidence of livestock depredation.

tavg.tif A raster of climatological mean temperature for Ladakh from WorldClim.org and downloaded directly through R using the `geodata` package.

Packages

This tutorial uses our standard suite of vector and raster processing and packages introduced in previous sessions.

```
library(sf)
library(tidyverse)
library(terra)
library(tidyterra)
library(colorspace)
```

Along with a set of `ggplot` extension packages needed for more advanced map making.

```
library(ggnewscale)
library(ggspatial)
library(patchwork)
library(cowplot)
```

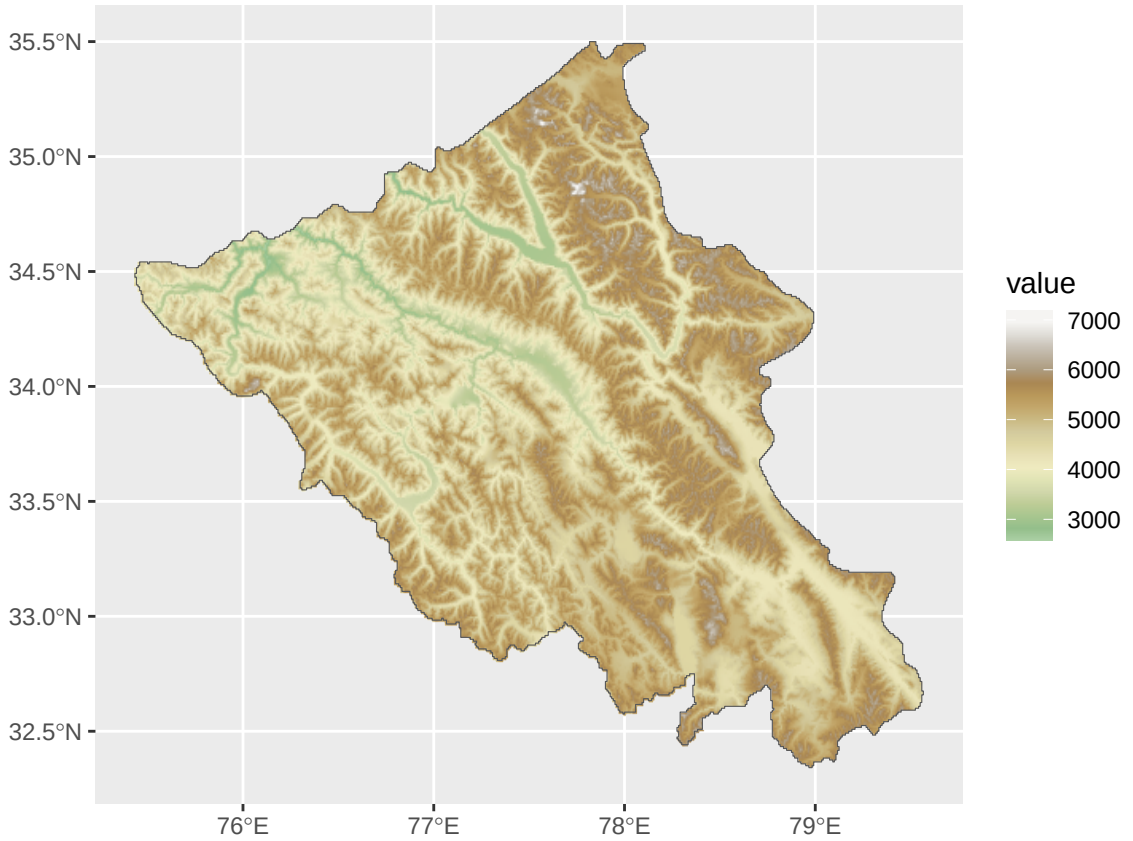
1. Load data

We will start off by loading in our snow leopard sighting data, elevation raster and Ladakh administrative boundary polygon used in Tutorial 3:

```
dem = rast('data/dem.tif')
ladakh   = st_read('data/ladakh.gpkg')
```

First, let's plot our elevation raster to remind ourselves what it looks like:

```
dem.plot =
ggplot() +
  geom_spatraster(data = dem) +
  geom_sf(data = ladakh, fill = 'transparent') +
  scale_fill_hypso_c(palette = 'wiki-2.0_hypso')

dem.plot
```



This plot is good for visualizing data, but it lacks a lot of key elements and the default ggplot grey canvas isn't to everyone's taste. Next, we're going to customize it a bit.

2. Introducing 'themes'

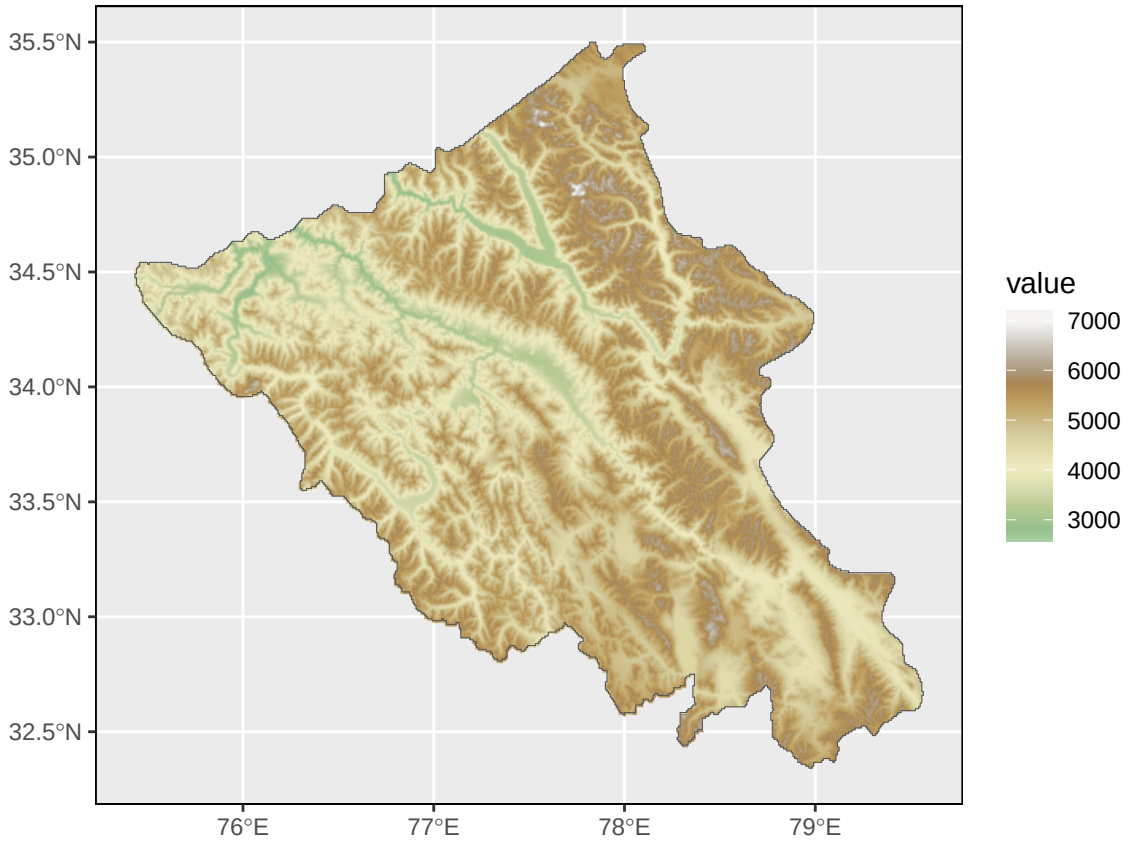
In `ggplot`, much of the way that a plot looks is controlled by the `theme`, and the same applies to maps. If you type `?theme` into the console you will see the huge number of visual elements that you can adjust on a

ggplot. We are only going to demonstrate a selection of the more common ones here, but it is worth taking some time to experiment with others.

Each part of your plot that you can edit in theme is called an element and there are three main types: element_rect (rectangular polygons, like panel borders or backgrounds), element_line (lines, like axes, tick marks and gridlines) and element_text (all text). You control the aesthetics of these elements using the same aesthetic properties that you use for geoms in ggplot (e.g. colour,linetype, size, fill etc.).

One common thing that we might want to is to add a border to our plotting area, or panel in ggplot language, which we control using the panel.border argument in the theme. This is a rectangular element so we control it using element_rect and set the colour and fill like this:

```
dem.plot +
  theme(panel.border = element_rect(colour = 'black', fill = 'transparent'))
```



In some cases we might also want to change the colour of the background, which we do using panel.background:

```
dem.plot +
  theme(panel.background = element_rect(fill = 'lightblue'))
```

In our case we might want to remove the background altogether, which we do by setting element_blank():

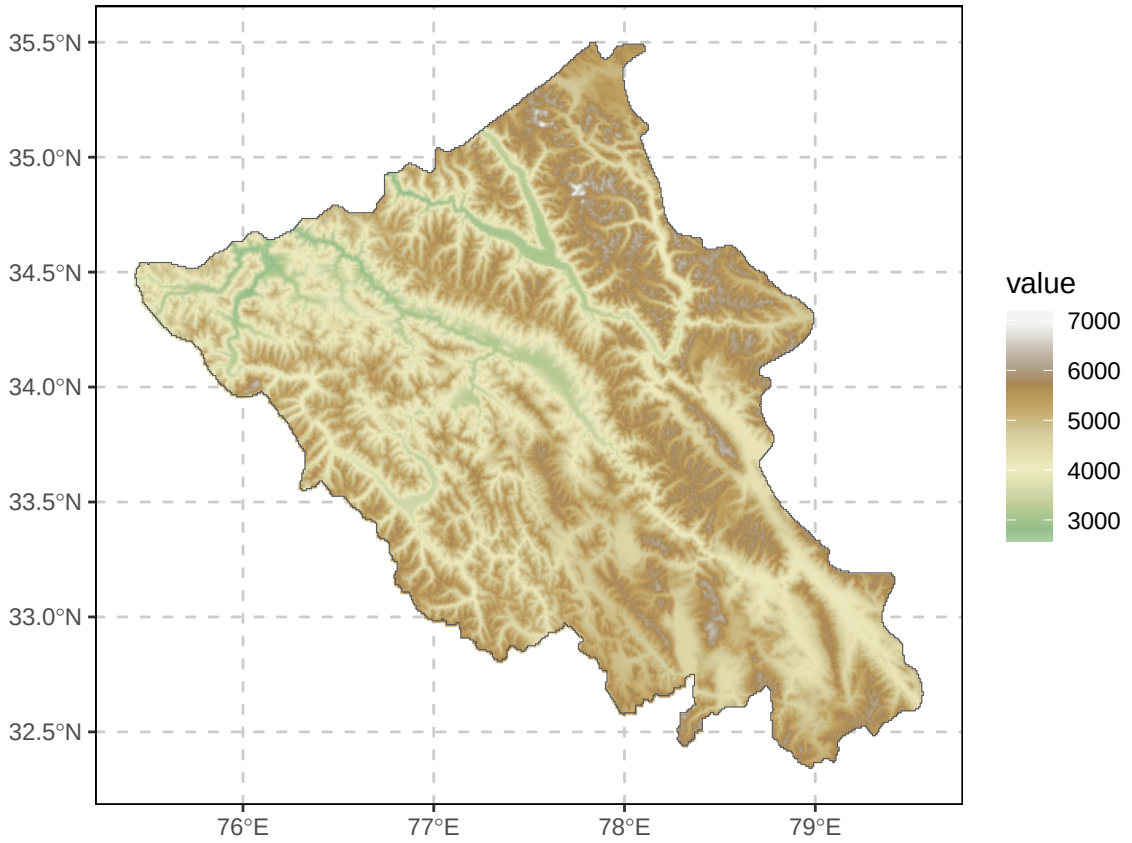
```
dem.plot +
  theme(panel.background = element_blank())
```

We can also control the appearance of the graticule (grid lines) using the `panel.grid.major` or `panel.grid.minor` arguments (major grid lines are the ones aligned with the axis tick marks, and minor are unlabelled ones in between; in our case we have no minor lines). These are line elements, so we control them with `element_line`:

```
dem.plot +
  theme(panel.grid.major = element_line(linetype = 'dashed',colour = 'grey30'))
```

And we can combine as many of these arguments as we want in a single theme:

```
dem.plot +
  theme(
    panel.background = element_blank(),
    panel.border = element_rect(colour = 'black', fill = 'transparent'),
    panel.grid.major = element_line(linetype = 'dashed',colour = 'grey80'),
        )
```



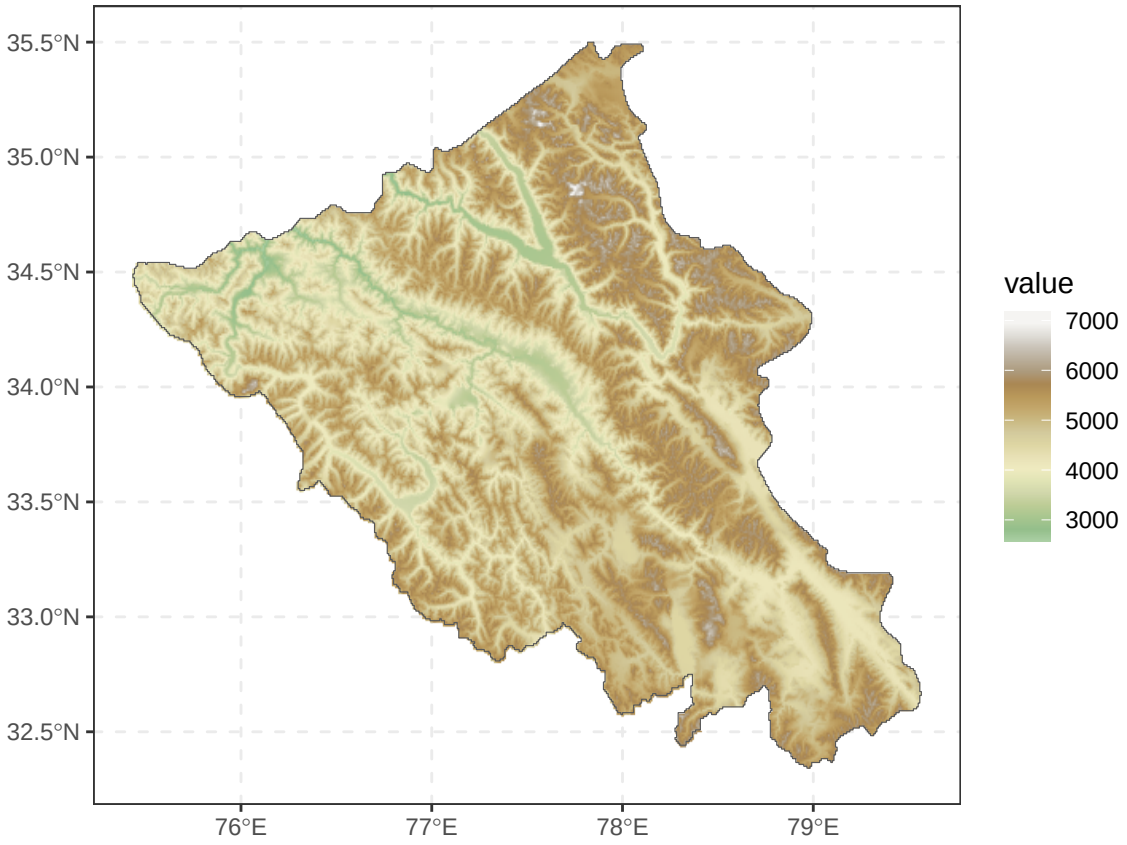
There are also some preset themes you can add to your plot. A full list can be found here https://ggplot2.tidyverse.org/reference/ggtheme.html, but try experimenting with some of the preset themes below to see what they do:

```
dem.plot +
  theme_void()

dem.plot +
  theme_classic()
```

```
dem.plot +
  theme_minimal()

dem.plot +
  theme_bw()
```

It is possible to customize elements of a preset theme by adding additional `theme` arguments to it. So, for example if we liked most of the things about `theme_bw` but would rather that the gridlines were dashed than solid, we can fo that like this:

```
dem.plot =
dem.plot +
  theme_bw() +
  theme(panel.grid.major = element_line(linetype = 'dashed'))

dem.plot
```

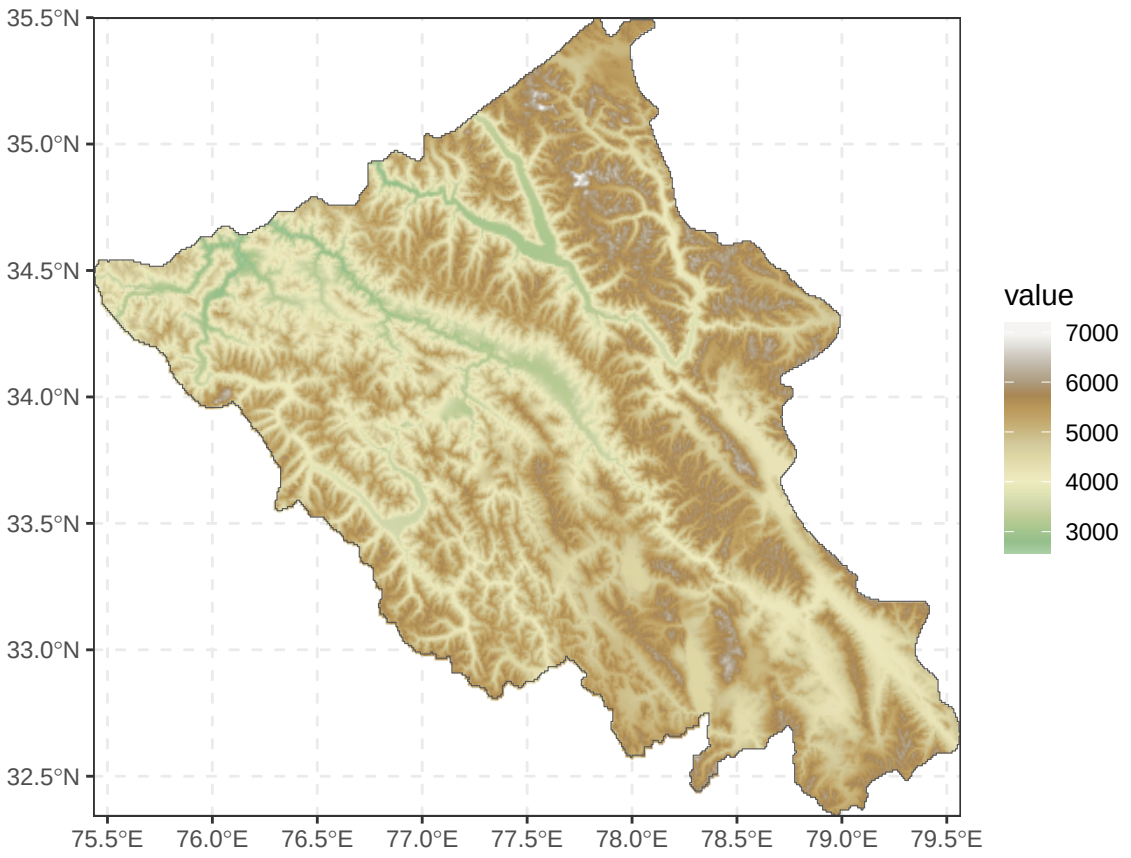


2. Controlling scales and graticules

Besides `theme` - the other main way of formatting the overall appearance of our plotting area is through changes to the coordinate system, which in a spatial plot is controlled using `coord_sf`.

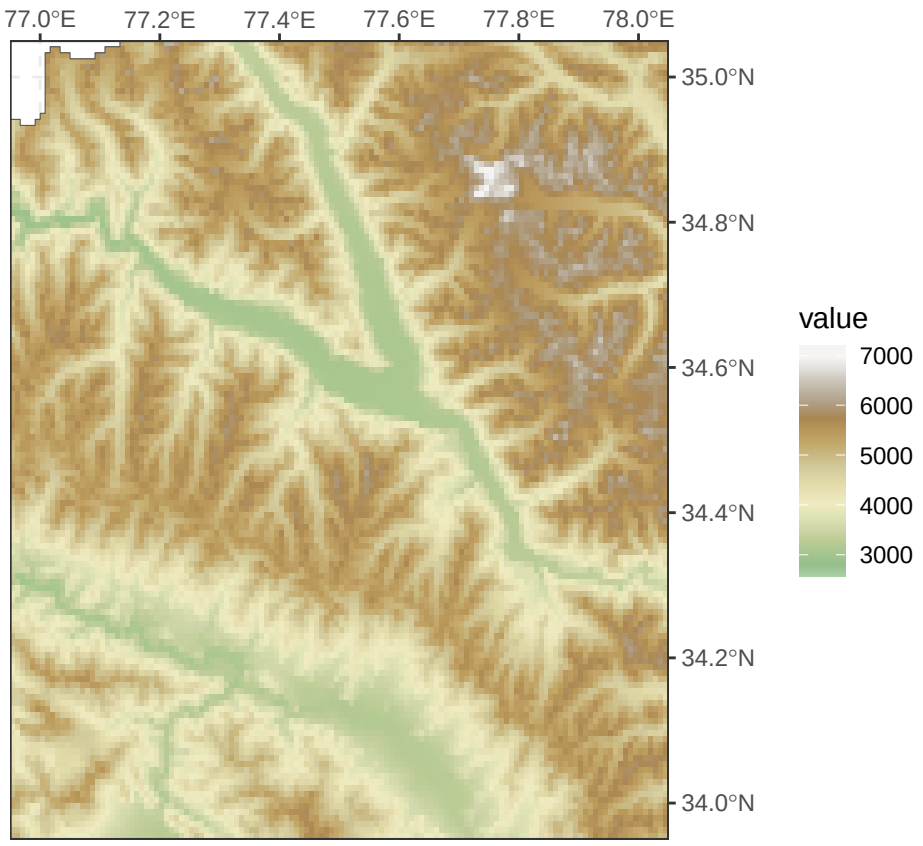
By default `ggplot` adds some padding between the edge of our data and the axes. That looks nice in this example, but if you had a rectangular raster or vectors clipped to the bounding box it can look quite odd. One of the most common uses of `coord_sf` in map making is to switch this padding off using `expand = F`

```
dem.plot +
  coord_sf(expand = F)
```



Using `coord_sf` we can also manually control the x and y scale limits using `xlim` and `ylim`, and decide which sides the graticule (axis) labels are plotted on using `label_graticule`.

```
dem.plot +
  coord_sf(xlim = c(77,78), ylim = c(34,35),
           label_graticule = 'NE')
```



The other common form of customization that we often want to do with our graticule is to modify the where the grid lines and axis labels are drawn. For example, in our case, the default scales have 0.5 degree increments on the y axis and 1 degree increments on the x. We might want these to be the same In `ggplot`, axis labelling is controlled by functions called `scale_y_continuous` and `scale_x_continuous`, and the breakpoints are controlled using the `breaks` argument:

```
dem.plot =
dem.plot +
  scale_y_continuous(breaks = c(33,34,35))
```

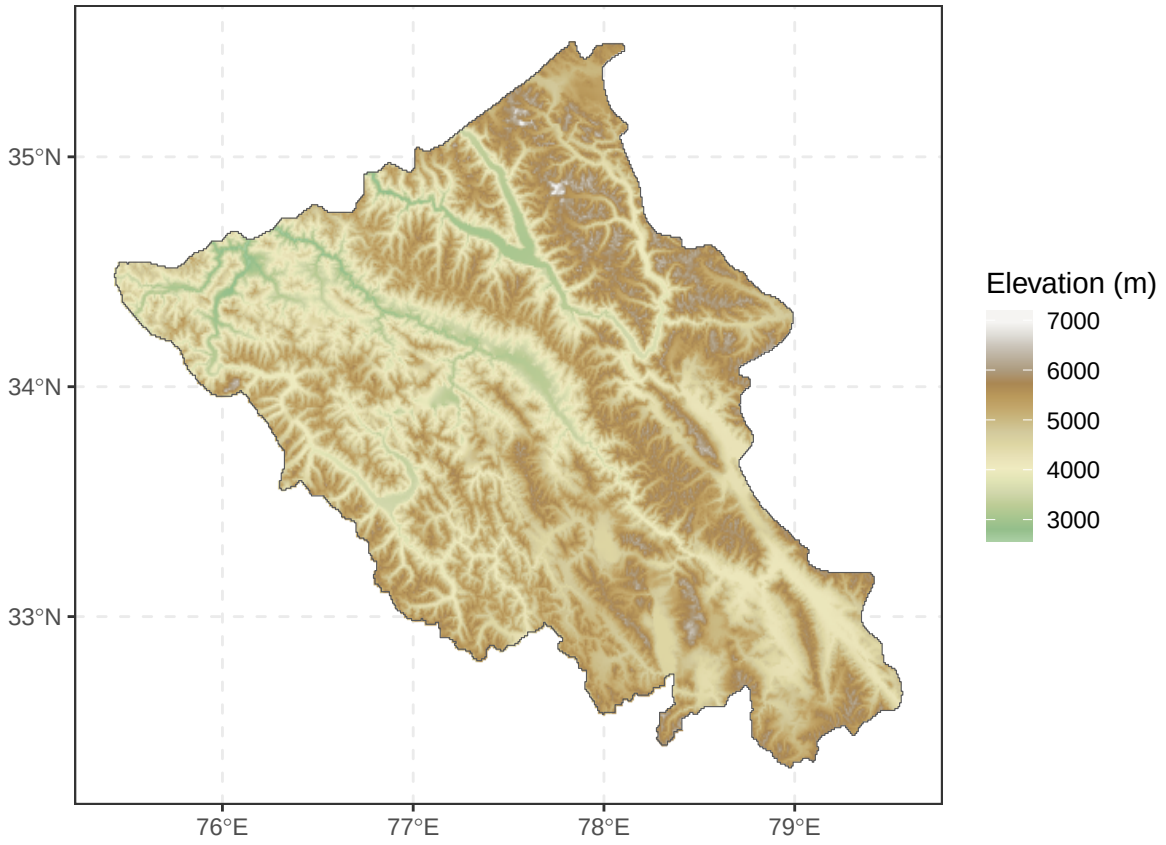
We've finished styling our map canvas; however, this map still has lots of issues. For one, the legend title is currently just `value` (the default for a raster layer) and has no units.

3. Formatting legends

Legends in `ggplot` are referred to as `guides`. There are two common types of guides that are used: `guide_legend` for categorical data (e.g. vector data) and `guide_colourbar` for continuous fill scales (e.g. raster data).

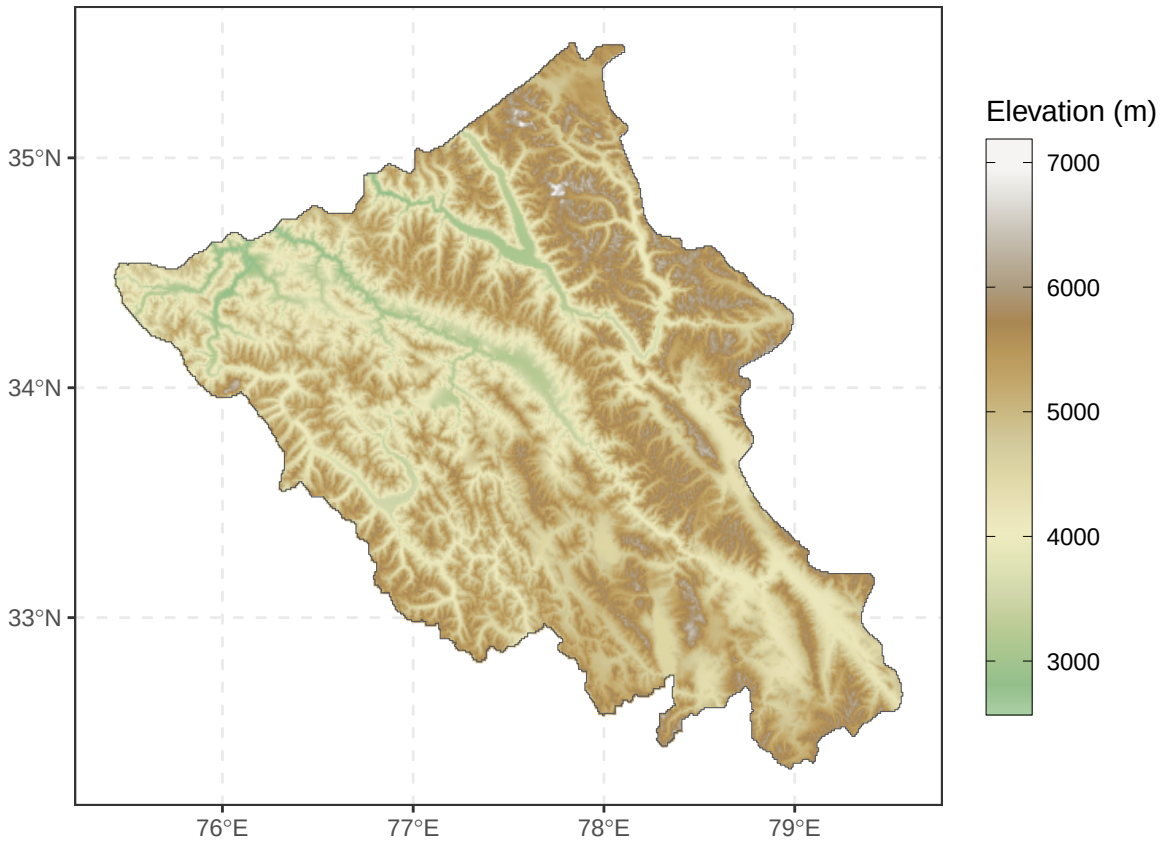
You can customize the guides in a few different ways. One way is to use the `guides` function and add it to the map. Inside `guides`, you name the aesthetic whose legend you want to adjust (in this case the fill guide for elevation) and then give it the relevant guide type defining some formatting options. So, to change the title of the fill legend we can do:

```
dem.plot +
  guides(fill = guide_colourbar(title = 'Elevation (m)'))
```



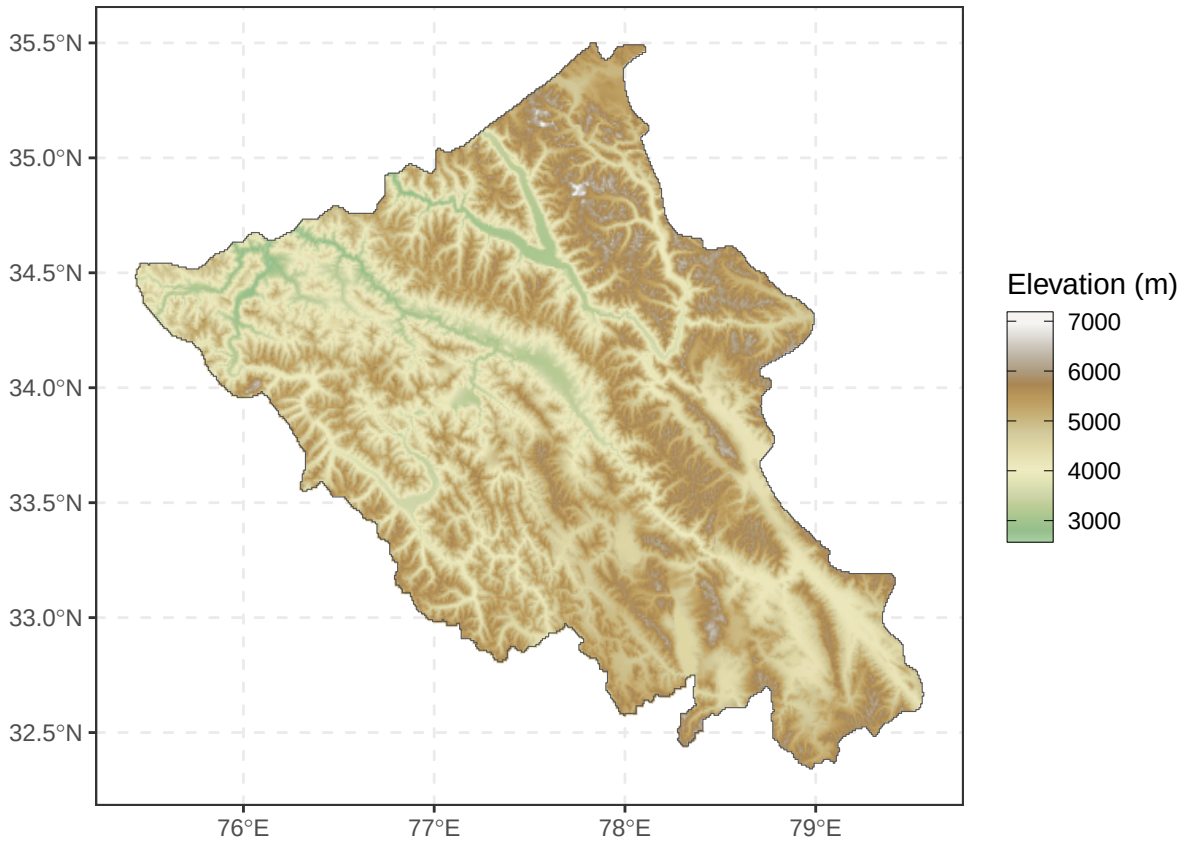
We can also change a bunch of other properties (type `?guide_colorbar` for examples). Here we will make the frame and tick marks black and increase the height of the bar

```
dem.plot +
  guides(fill = guide_colourbar(title = 'Elevation (m)',
                                frame.colour = 'black',
                                ticks.colour = 'black',
                                barheight = 15))
```



This approach works for simple maps where we have just a single fill scale, but (as we will see later), in GIS it is not unusual to have multiple scales for certain layer aesthetics (e.g. several layers each with a different colour scale). A better approach is to modify the guides inside the scale for the aesthetic that they relate to (we will see why that is important later). Here, we remake our `dem.plot` with the colourbar formatted inside the `scale_fill_hypso_c` applied to our elevation layer:

```
dem.plot =
ggplot() +
  geom_spatraster(data = dem) +
  geom_sf(data = ladakh, fill = 'transparent') +
  scale_fill_hypso_c(palette = 'wiki-2.0_hypso',
                     guide = guide_colourbar(title = 'Elevation (m)',
                                             frame.colour = 'black',
                                             ticks.colour = 'black')) +
  theme_bw() +
  theme(panel.grid.major = element_line(linetype = 'dashed'))

dem.plot
```

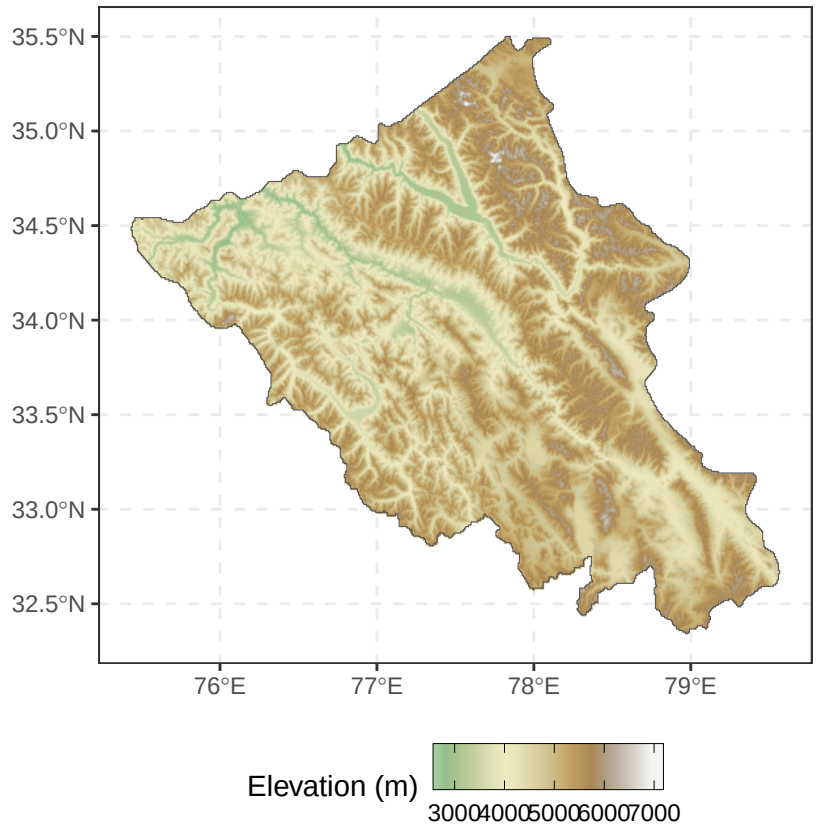


We can also control several aspects of the legend using `theme`. For example, we can change the size of legend title and text using `legend.title` and `legend.text`:

```
dem.plot +
  theme(legend.title = element_text(size = 16),
        legend.text = element_text(size=8))
```

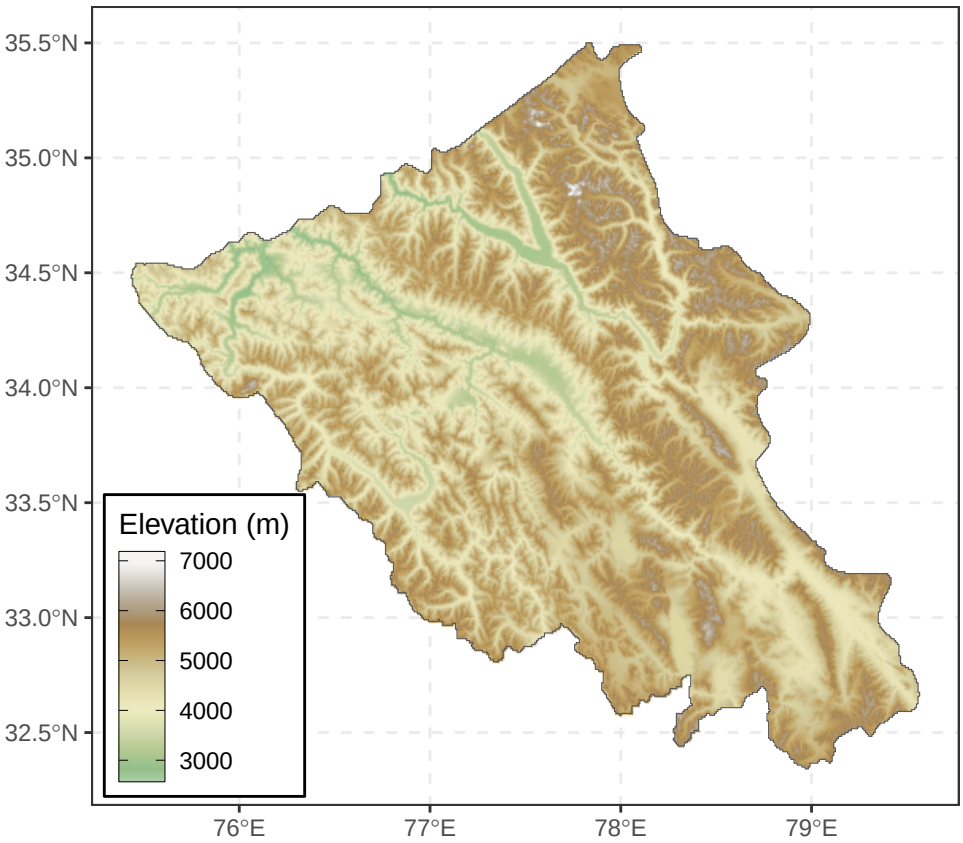
And we can use `legend.position` to change the postion of the legend. This will place the legend at the bottom of the plot (and make it horizontal by default):

```
dem.plot +
  theme(legend.position = 'bottom')
```



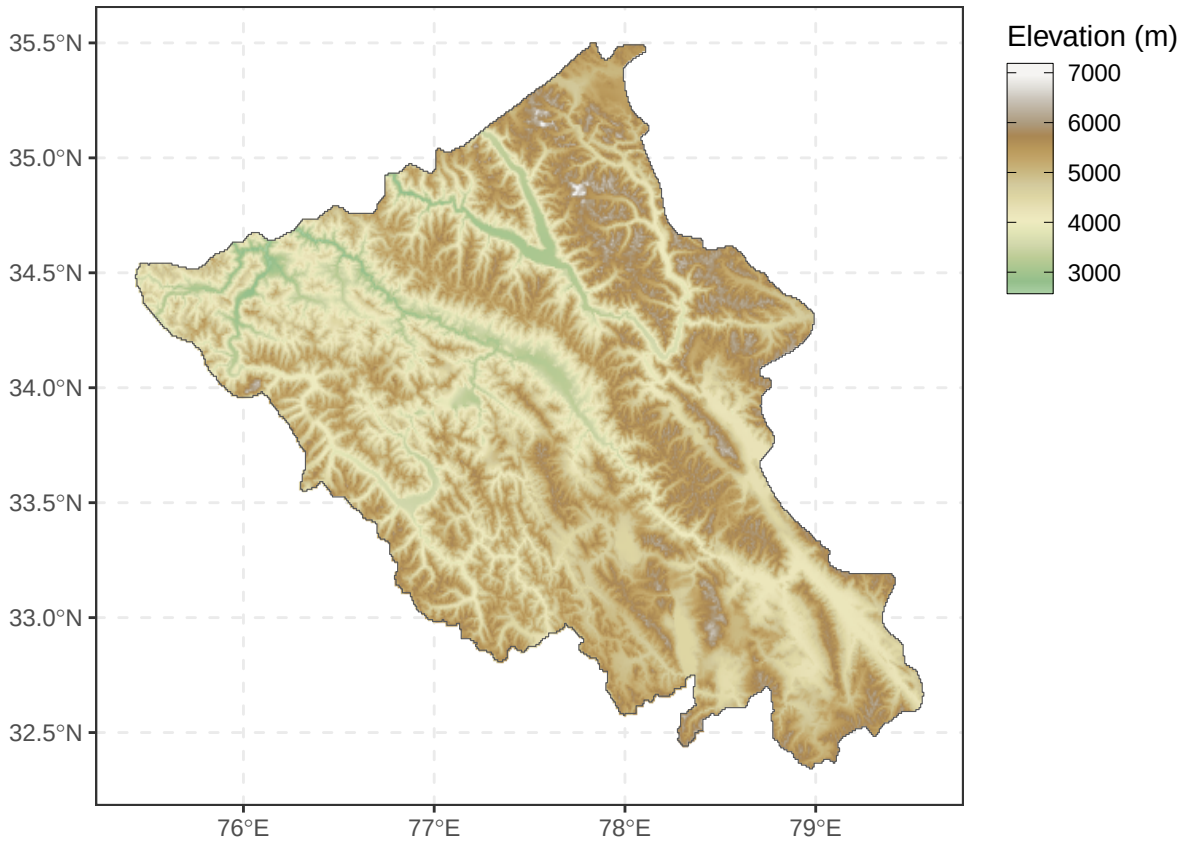
We can also control the position of the legend manually using x and y coordinates. In this case, coordinates are **not the map coordinates** but are relative plotting region coordinates with (0,0) being the bottom left corner and (1,1) being the top right. This usually requires a bit of trial and error to get right:

```
dem.plot +
  theme(legend.position = c(0.13,0.2),
        legend.background = element_rect(colour = 'black'))
```



For our example, we are going to use another `theme` argument, `legend.justification` to move our legend on the right hand side to the top to make space for an inset map at the bottom.

```
dem.plot =
  dem.plot +
  theme(legend.justification = 'top')

dem.plot
```



So we've plotted our elevation data, styled the plot a bit and got the legend looking as we want. Now let's add some more data to our map. We'll start by reading in our leopard sightings data from the Tutorial data folder.

```
leopards = st_read('data/leopards.gpkg')
```

4. Handling multiple legends for the same aesthetic

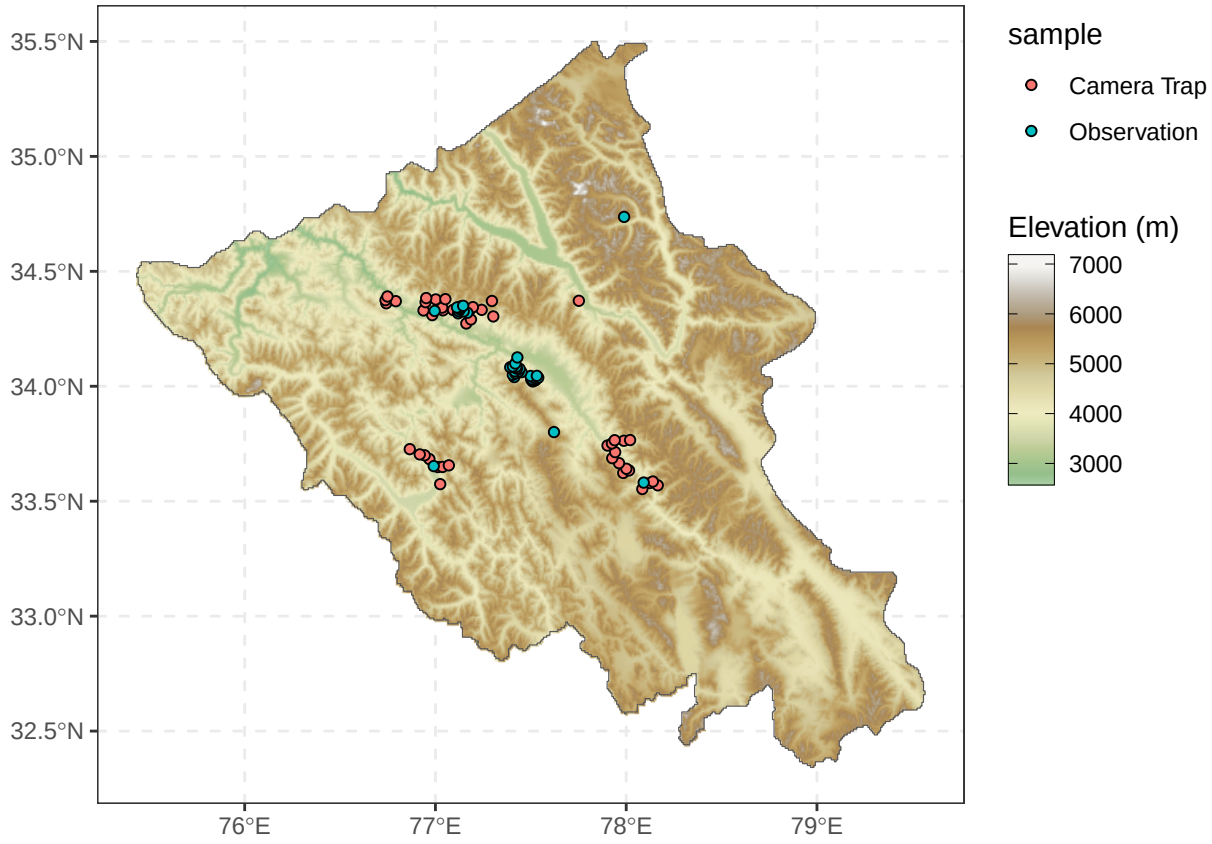
As explained above, in GIS, we often need to have several legends for different layers that use the same aesethics (e.g. colour, fill). In our case, for example, we might want to plot our leopard sightings over our elevation basemap and fill the points based on observation or camera trap sample:

```
dem.plot +
 geom_sf(data = leopards, mapping = aes(fill = sample), shape = 21)
```

But this will give us an error, because this plot already has a fill scale for the elevation raster, which we are now trying to apply to our categorical leopard sighting data. It won't work. By default, `ggplot` will only allow us to have one scale for each aesthetic. To use multiple fill scales we need to enlist the help of an extension package called `ggnewscale`. In this case we use `new_scale_fill` before we add our next layer and the old fill scale for elevation is forgotten.

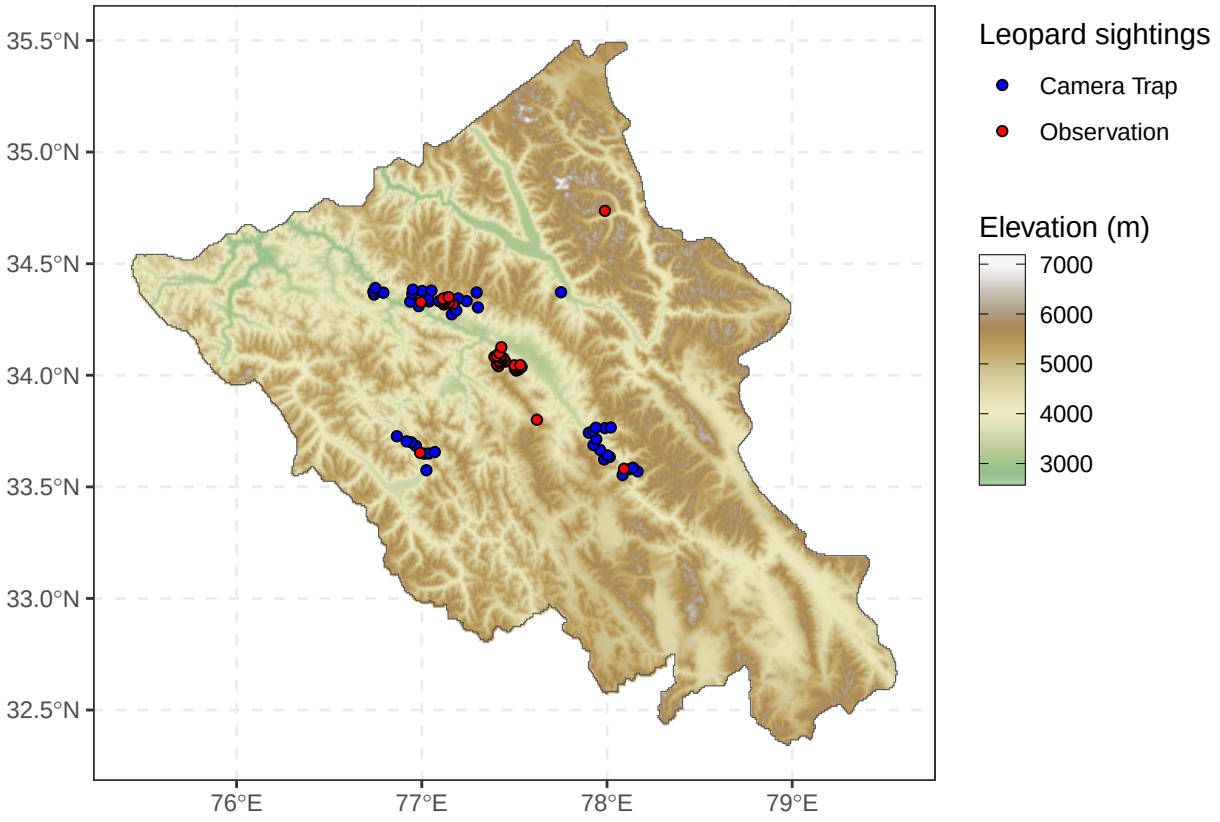
```
dem.plot =
dem.plot +
 new_scale_fill() +
```

```
 geom_sf(data = leopards, mapping = aes(fill = sample), shape = 21)

dem.plot
```



Any editing we do of the fill scale will now apply only to the new layer:

```
dem.plot =
  dem.plot +
  scale_fill_manual(values = c('blue','red'),
                    guide = guide_legend(title = 'Leopard sightings'))

dem.plot
```

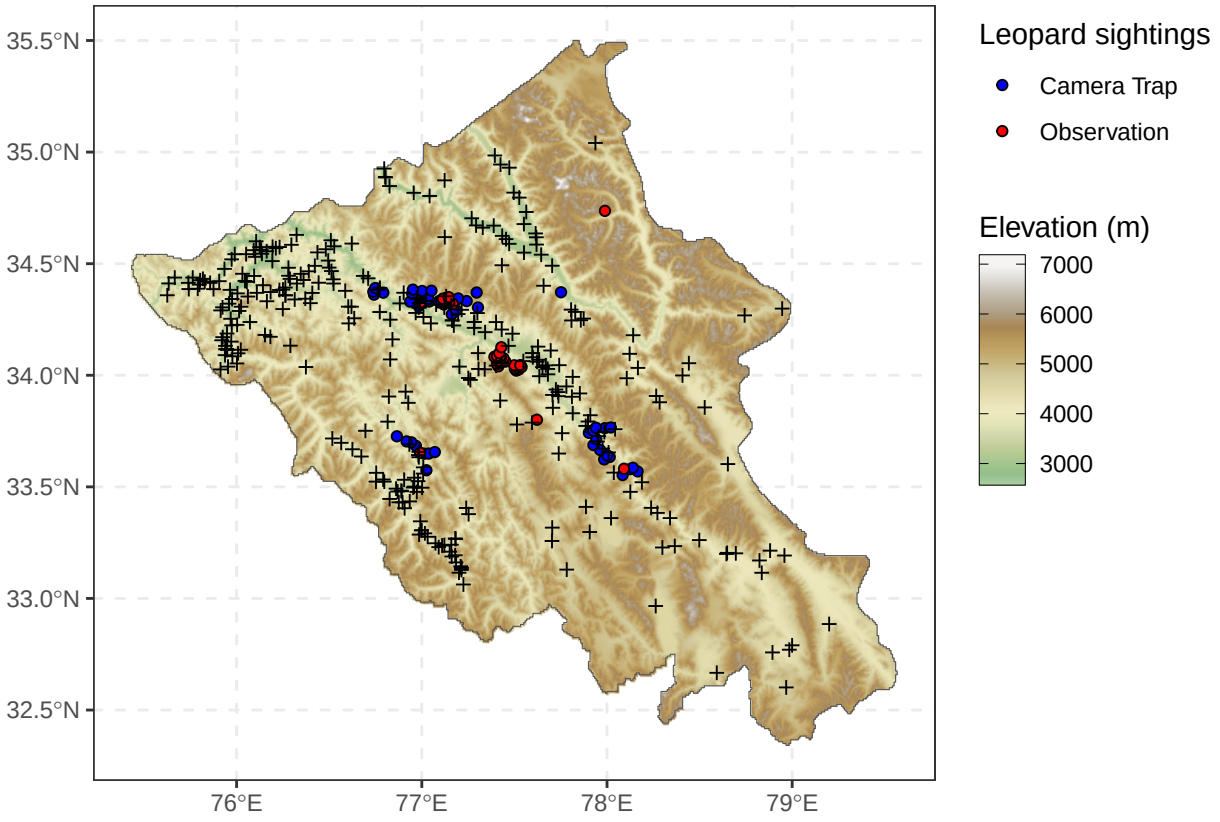


You can use `new_scale_colour` and `new_scale_shape` to do the same for other aesthetics.

5. Adding legends for single category features

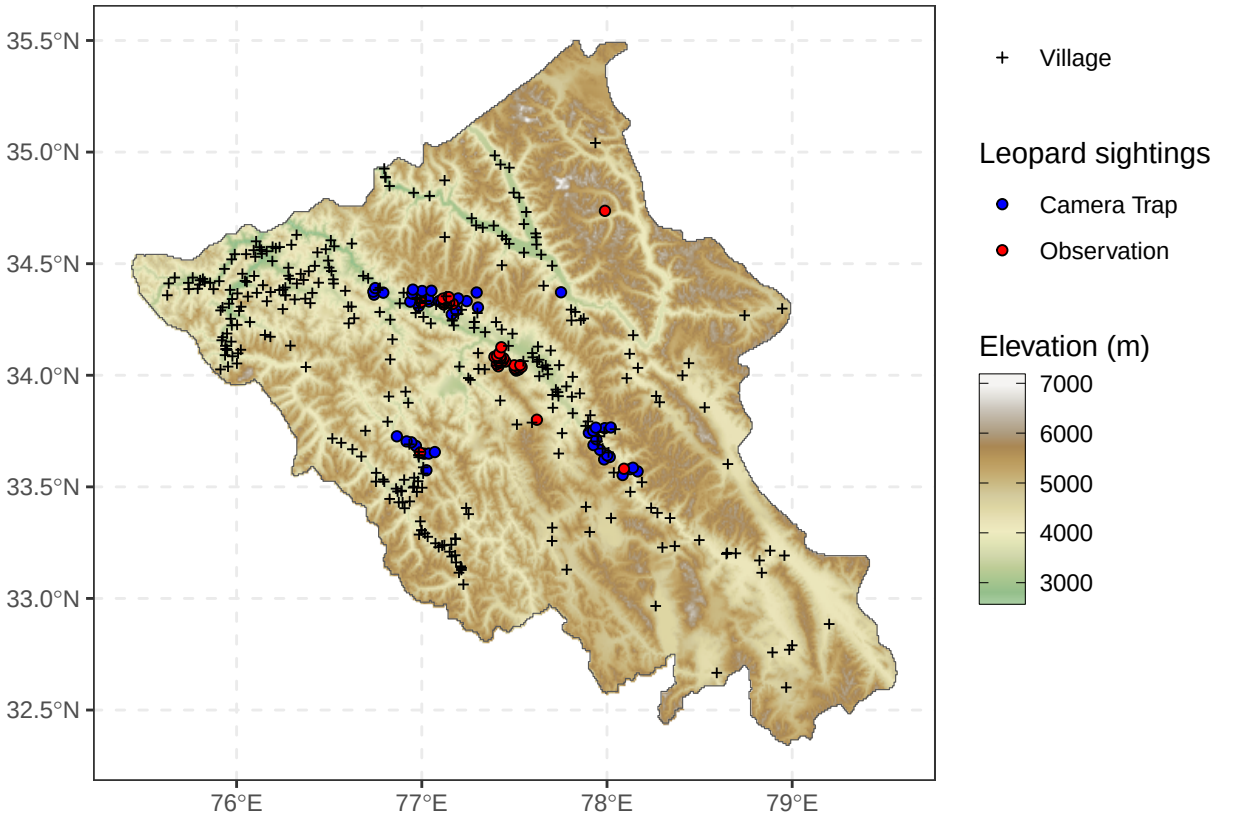
`ggplot` is designed for data visualization rather than as a GIS and by default it only produces legends for layers where there are multiple categories to differentiate between (i.e. where you have set an aesthetic property using `mapping = aes()`). This makes sense for data plotting, but in GIS we often want a legend to explain what certain map symbols mean, even if there is only a single symbol type for a given layer. In our case, for example, let's plot the distribution of villages that Watts et al. (2019) visited to assess evidence of human-leopard conflicts.

```
villages = st_read('data/villages.gpkg')

dem.plot +
 geom_sf(data = villages, shape = 3)
```



You can see that we get no legend here for village because there is only one category. However, we can trick `ggplot` into giving us a legend by mapping an aesthetic, in this case `shape`, to the description of that feature that we want to appear in the legend. This creates a sort of dummy category and results in a `shape` legend being printed. We can then manually modify this scale to specify the plotting shape we want for our single category and format the guide (legend) to remove the uninformative title.

```
dem.plot =
dem.plot +
 geom_sf(data = villages, mapping = aes(shape = 'Village'), size=1) +
 scale_shape_manual(values = 3, guide = guide_legend(title = NULL))

dem.plot
```



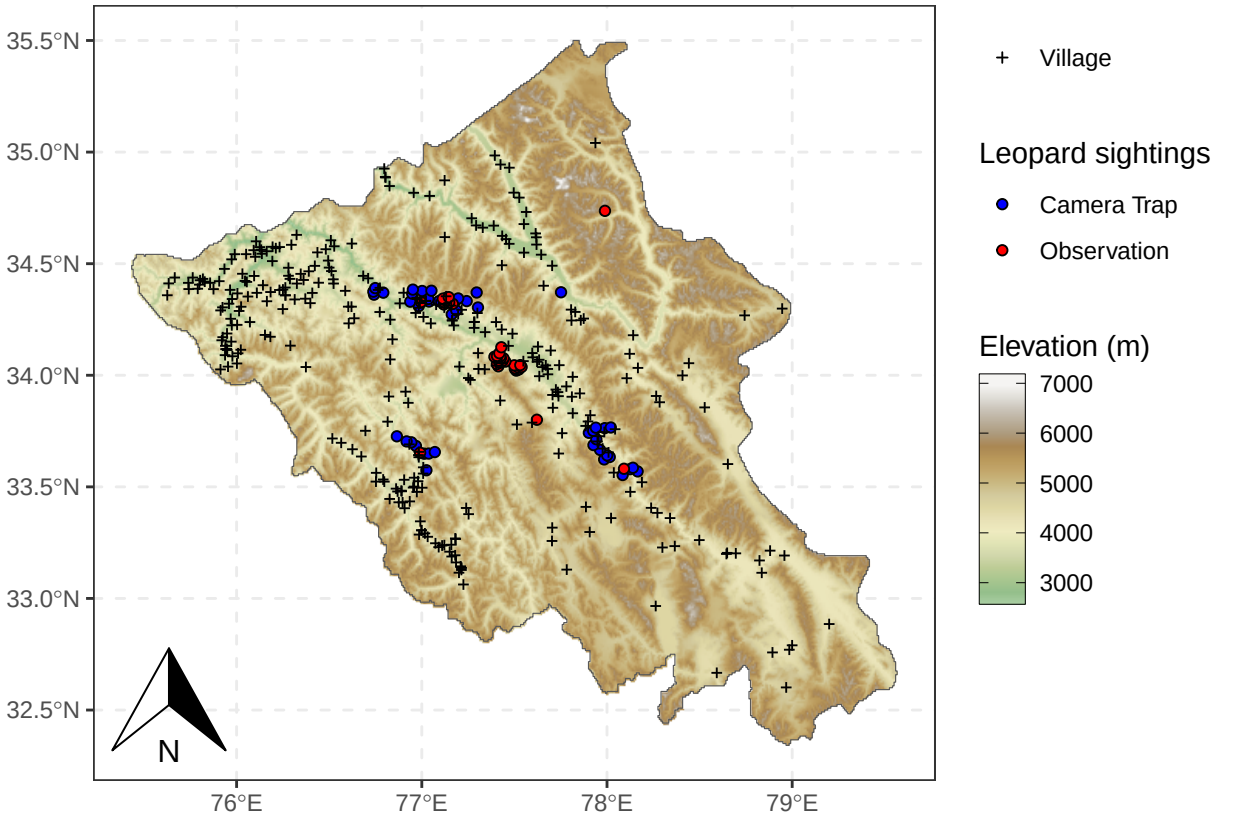
Our legend is now quite informative but there are still a few elements missing from this map, including a north arrow and scalebar.

6. Scalebars and north arrows

There are a few different extension packages for `ggplot` that allow you to add scale bars and north arrows to maps. Here, we are going to use the `annotation_scale` and `annotation_north_arrow` functions from `ggspatial`.

Let's start by adding a north arrow:

```
# Note that there is also a function called 'north' in terra, so we
# need to tell R which version we want to use by giving the source
# package name followed by '::'

dem.plot +
  annotation_north_arrow()
```



We can also adjust some additional features like the location, size and symbol type (see `?annotation_north_arrow` for all options).

We adjust the location by specifying which corner to place it using the `location` argument (options are: `tl`, top left; `bl`, bottom left; `tr`, top right; and `br`, bottom right) and adjusting the amount of padding between the frame and the arrow using the `pad_x` and `pad_y` arguments. The latter are `unit` objects which give the unit and value. So here we place in the top corner and indent by 1 cm along the x axis:

```
dem.plot +
  annotation_north_arrow(location = 'tl', pad_x = unit(1,'cm'))
```

We can also adjust the symbol using the `style` option. There are 4 styles we can choose from which are set using the functions `north_arrow_minimal()`, `north_arrow_nautical()`, `north_arrow_orienteering()` and `north_arrow_fancy_orienteering()`

```
dem.plot +
  annotation_north_arrow(location = 'tl',
                         style = north_arrow_nautical()
  )
```

Within each of these functions we can also change the styling of the symbol. So here we will make the fill of the symbol grey and white and the outline a darker grey:

```
dem.plot +
  annotation_north_arrow(
    location = 'tl',
```

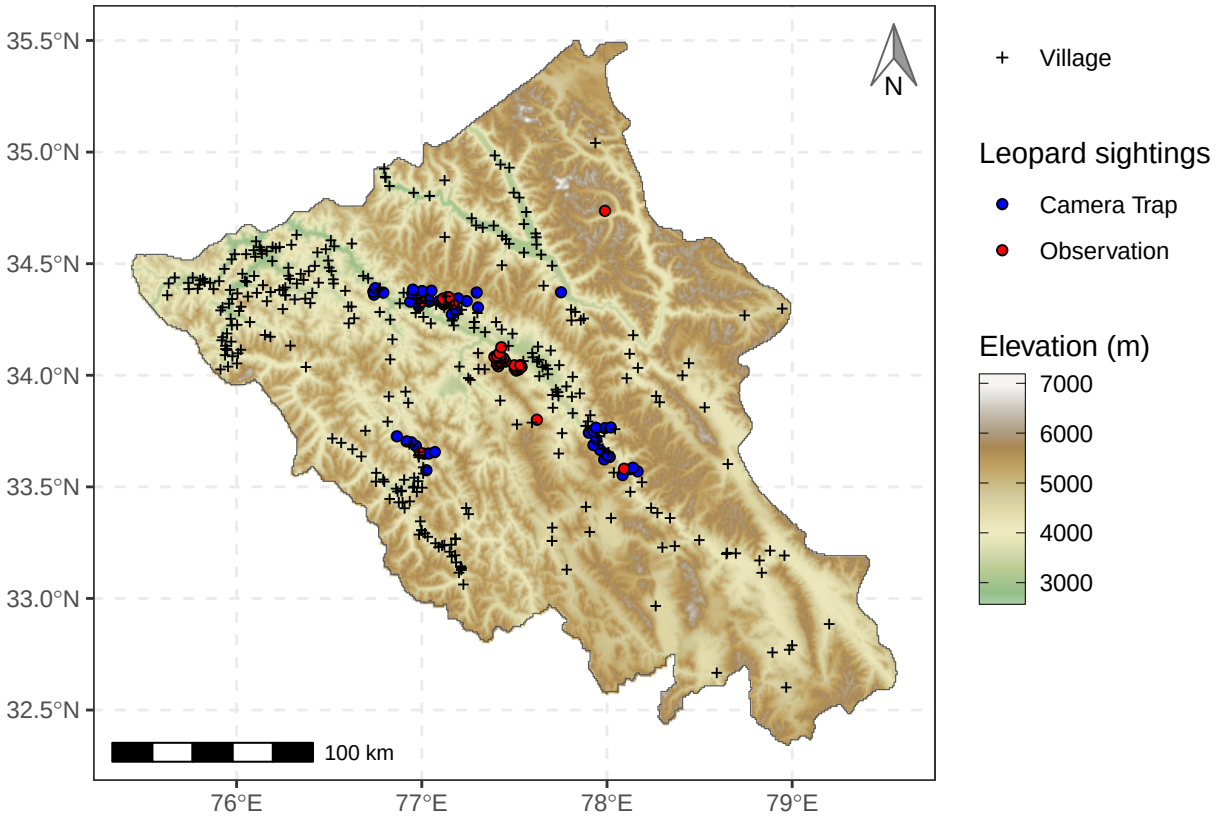
```
    style = north_arrow_nautical(
      fill = c('grey60','white'),
      line_col = 'grey40'
    )
  )
```

Here, we will keep the default symbol but reduce its size using the `width` and `height` arguments (also `unit` objects):

```
dem.plot =
dem.plot +
  annotation_north_arrow(
    location = 'tr',
    width = unit(0.6,'cm'),
    height = unit(0.9,'cm'),
    style = north_arrow_orienteering(
      fill = c('white','grey60'),
      line_col = 'grey40',
      text_size=8)
)

dem.plot
```

The `annotation_scale` function works in much the same way (see `?annotation_scale` for all options):

```
dem.plot +
  annotation_scale()
```



We can adjust its location using the `location`,`pad_y` and `pad_x` arguments, with the same options as for `annotation_north_arrow`:

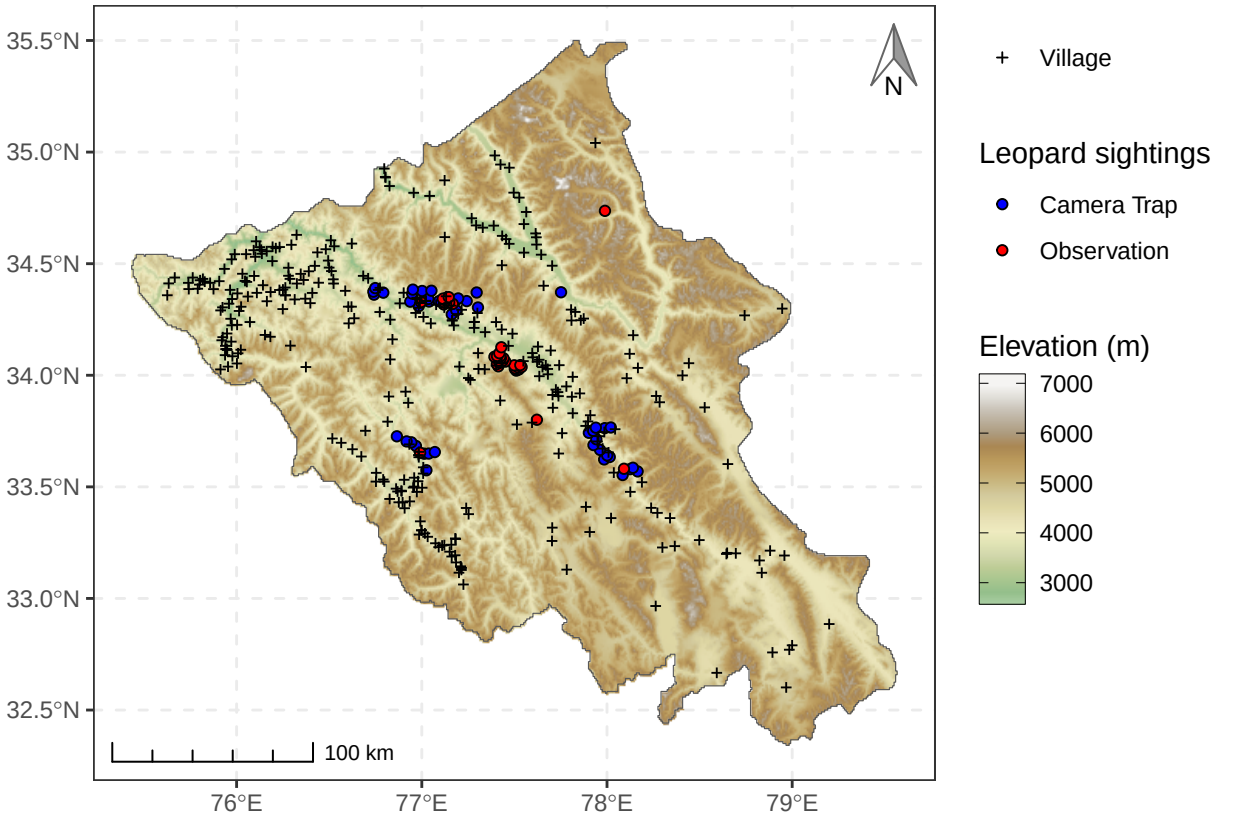
```
dem.plot +
  annotation_scale(location = 'tr',pad_y = unit(1,'cm'))
```

We can also adjust the fill and outline colours:

```
dem.plot +
  annotation_scale(
    bar_cols = c('grey60','white'),
    line_col = 'grey40'
    )
```

And change the style to a simple line and tick marks (`ticks`):

```
dem.plot +
  annotation_scale(style = 'ticks')
```



Here we will just keep the default bar but reduce its height slightly :

```
dem.plot = dem.plot + annotation_scale(height = unit(0.2,'cm'))
```

This map is starting to come together. We have plotted all our data, but we might want to add a few labels or placenames to our map for geographic context.

7. Adding text labels to maps

There are two ways of adding text to a map in `ggplot`. The first is to simply add text annotations at some manually controlled point on the map which we do using the `annotate` function. In `annotate` we set the `geom` argument to 'text' and the label that we want to print, along with its x and y coordinate. We can also control all the usual text aesthetics like colour, size, transparency and angle. Here we'll write the name of the 'Himalaya' along the range.

```
dem.plot +
  annotate(geom = 'text', label = 'H I M A L A Y A', x = 78, y = 34, size = 6, alpha = 0.3, angle=310)
```

We might also want to label individual points or features based on some attributes in the data. To demonstrate, let's load in a simple geopackage containing a point feature with the location of the regional capital, Leh, and plot it on the map.

```
cities = st_read('data/cities.gpkg')

dem.plot =
  dem.plot +
  geom_sf(data = cities, shape = 24, fill = 'white',size = 2)
```

To label this point we can then use a special `geom` called `geom_sf_text`. It takes its coordinates from the `sf` layer but needs you to tell it which variable in the data to map to the label. We will use the 'name' column. We will also use the `nudge_x` and `nudge_y` arguments to adjust the plotting coordinates slightly so it doesn't overlap the point itself:

```
cities = st_read('data/cities.gpkg')

dem.plot =
  dem.plot +
  geom_sf_text(data = cities, mapping = aes(label = name),
               nudge_y = 0.1, nudge_x = 0.05)
```

8. Adding titles and captions

In addition to adding text annotations to the map itself, in some situations we might want to add titles or captions around our plot describing the content, source of our data, or other relevant information (e.g. projection system).

These types of text are controlled using the `labs` function (short for labels). Using `labs` we can add titles, subtitles and captions, as well as controlling the axis labels. In this case we might want to remove the axis altogether by setting them to `NULL`:

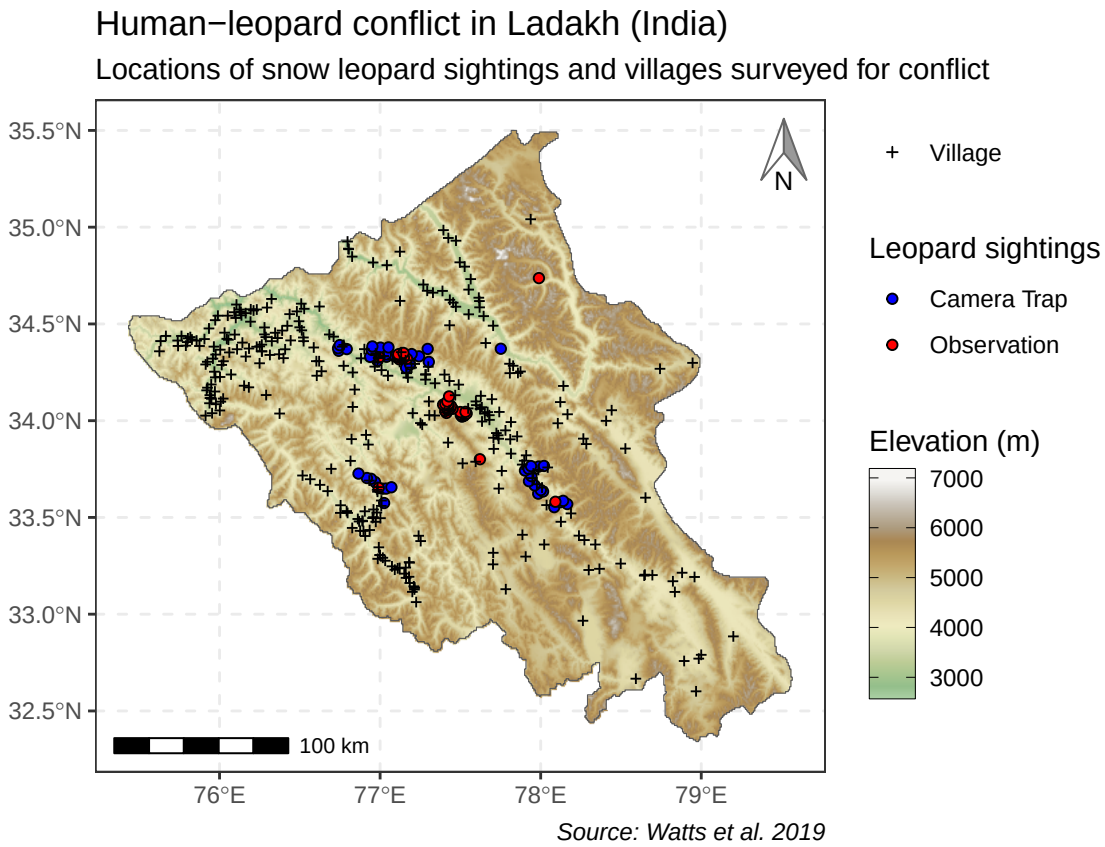
```
dem.plot =
dem.plot +
  labs(title = 'Human-leopard conflict in Ladakh (India)',
       subtitle = 'Locations of snow leopard sightings and villages surveyed for conflict',
       caption = 'Source: Watts et al. 2019',
       x = NULL, y = NULL)
```

We then control the look of the titles using theme:

```
dem.plot +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(size = 6),
        plot.caption = element_text(face = 'italic'))
```

We will just make the caption italic for our map.

```
dem.plot =
dem.plot +
  theme(plot.caption = element_text(face = 'italic'))


dem.plot
```



This plot is looking pretty good. The final thing we might want to is to add an inset map showing the location of our Ladakh study area at a smaller scale e.g. relative to the whole of India. To do that we need to learn how to combine different maps on the same canvas.

9. Combining multiple figures

Using base graphics in R, you may have learned to plot multiple figures side-by-side by setting graphical parameters like `par(mfrow = T)`. However, this does not work for `ggplot` and other high level *object-based* graphics packages where each of your plots is stored as an object (in this case we have a graphical object called `dem.plot`) rather than just printed in the graphics window. There are many, many advantages to storing graphics this way, but it means we need to learn a different approach for arranging them on our canvas.

There are two common use cases where you might want to combine multiple plots in a single figure, and two different `ggplot` extension packages that can be used to achieve each most efficiently.

9a. Multipart plots

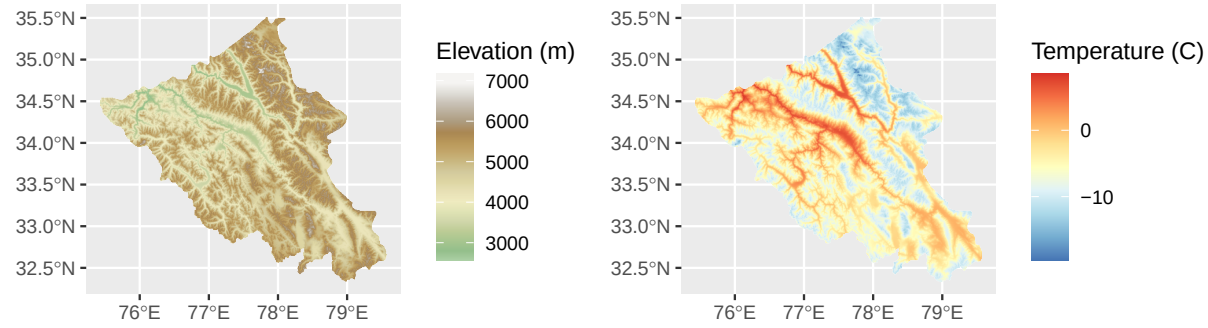
Often, we might want to have a series of panels that display different but related data as part of the same figure, usually with individually lettered panels (e.g. Figure A, B, ..). For example, in our case we might want to display the elevation and average air temperature for our Ladakh study area in a two-part figure.

First we will read in our temperature data and create two simple plots for temperature and elevation.

```
temp = rast('data/tavg.tif')

temp.plot =
  ggplot() +
  geom_spatraster(data = temp) +
  scale_fill_whitebox_c(palette = 'bl_yl_rd',
                        guide = guide_colorbar(title = 'Temperature (C)'))

elev.plot =
  ggplot() +
  geom_spatraster(data = dem) +
  scale_fill_hypso_c(palette = 'wiki-2.0_hypso',
                     guide = guide_colourbar(title = 'Elevation (m)'))
```

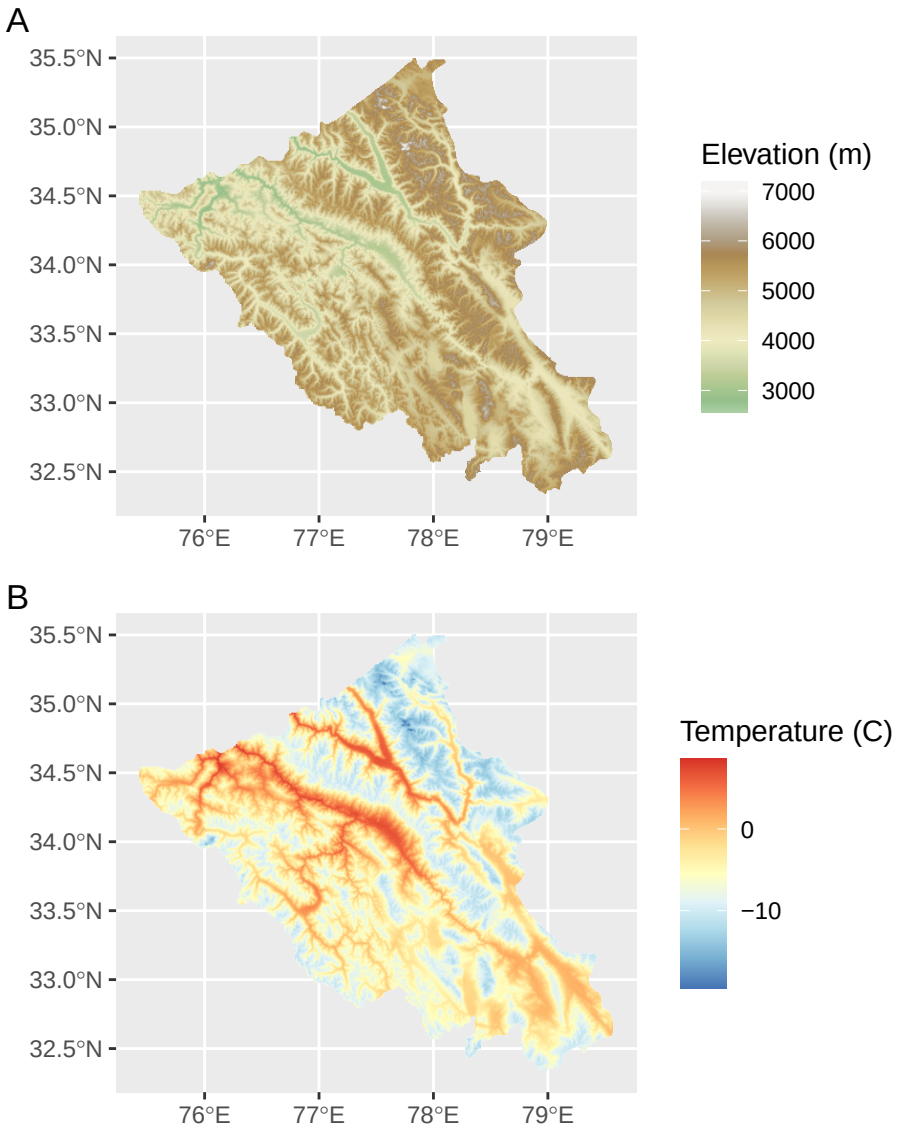
Now we want to combine it with our elevation map in a single plot. The simplest extension package for doing this is called `patchwork`. With `patchwork` loaded, we can simply add any number of `ggplot` objects together to arrange them in a single object:

```
library(patchwork)

multi.plot = elev.plot + temp.plot

multi.plot
```



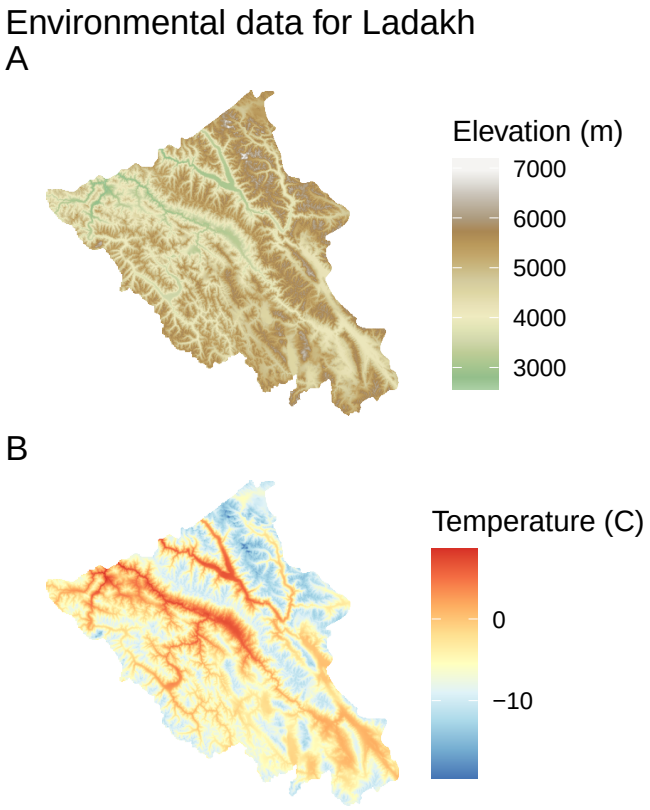
`patchwork` then gives us lots of options for customizing our multi-part figure. For example we can change the layout by adding a `plot_layout` and add panel labels (`tag_levels`) or general titles or captions by adding `plot_annotation`:

```
multi.plot =
multi.plot +
  plot_layout(ncol = 1) +
  plot_annotation(tag_levels = 'A',
                  title = 'Environmental data for Ladakh',
                  caption = 'Source: a) SRTM; b) WorldClim')

multi.plot
```



`patchwork` even allows you to modify the `theme` of all of the plots in a multipart simultaneously to ensure consistency by simply adding a `ggplot` theme (note that we do this using the ampersand `&` and not the `+` in this case to differentiate from adding a theme to a single `ggplot`).

```
multi.plot &
  theme_void()
```



See the vignettes at https://patchwork.data-imaginist.com/articles/ for a demonstration of all the formatting options available with `patchwork`.

9b. Inset maps

The other common use case for combining maps is to add inset maps (zoomed out views) of your study area to provide geographic frame of reference. The first step in creating an inset map is to create the inset itself - usually a simple outline of your study area on a smaller scale map. Here, we will load in a basemap of India, and make a simple map with Ladakh administrative boundary plotted on it in red.

```
india = st_read('data/india.gpkg')

inset =
  ggplot() +
  geom_sf(data = india, fill = 'grey80') +
  geom_sf(data = ladakh, fill = 'red') +
  theme_void()

inset
```

In some cases you might not have a vector layer that neatly describes the extent your study area for plotting on the inset. For example, you might have a raster layer or a series of points in your main map and need to plot a rectangular box describing the extent in your inset. In these cases, you need to create a polygon that describes the spatial extent of your data. This code shows you how to make a vector polygon from the extent of a `SpatRaster` or `sf` object and plot it on the inset:

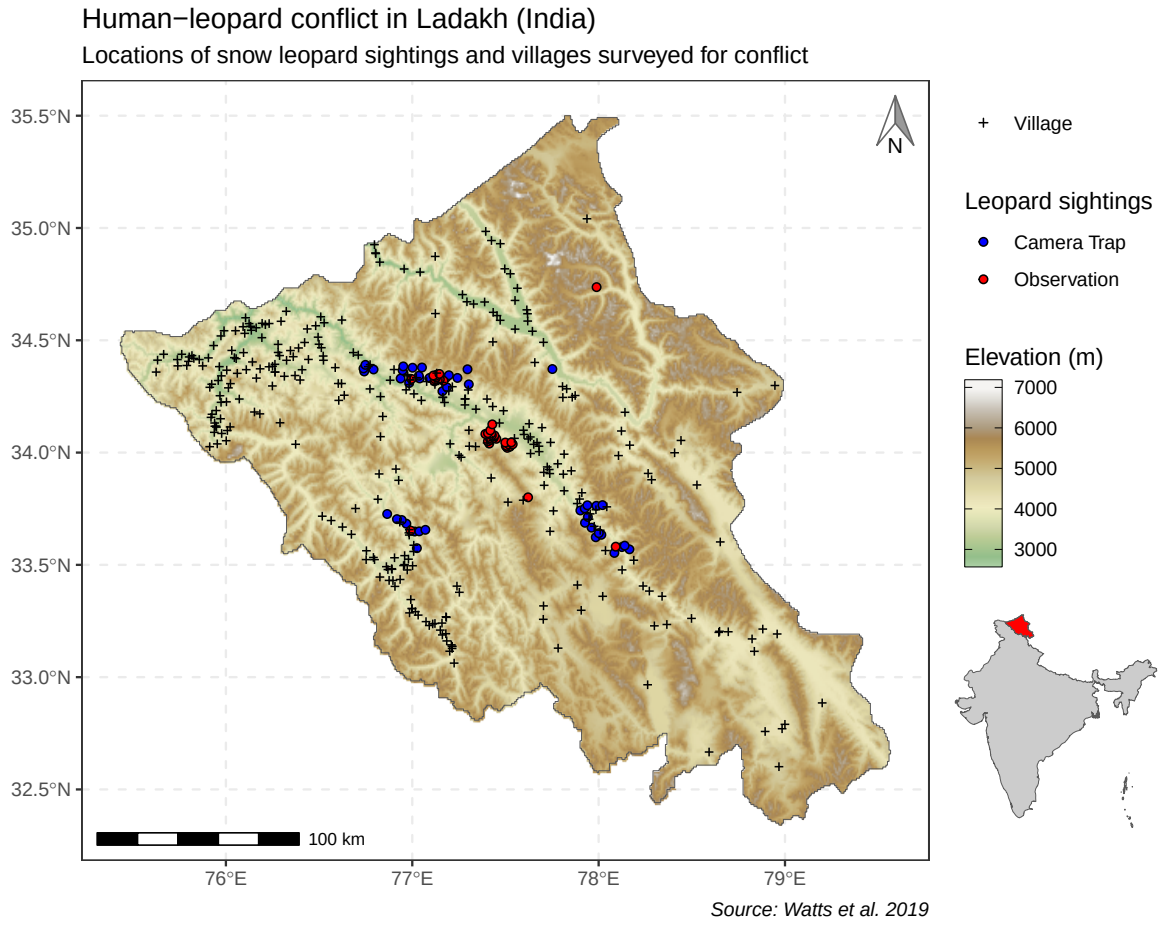
```
box = as.polygons(ext(dem))

box = st_as_sfc(st_bbox(ladakh))

ggplot() +
geom_sf(data = india, fill = 'grey80') +
geom_sf(data = box, colour = 'red', fill = 'transparent') +
theme_void()
```

Now we will add our inset map to the leopard conflict map we created earlier. Although `patchwork` can do

this, there is another package called `cowplot` which has a slightly less intuitive syntax but that offers more flexible re-positioning of figures.

To combine maps using `cowplot` we first use the `ggdraw()` function to draw a large canvas and then add individual `ggplot` objects to it using `draw_plot`. Within `draw_plot` we can set the x and y coordinate for the bottom left corner of each plot along with its height and width, giving us complete control over where each plot is drawn. As with legend re-positioing earlier, the coordinates used in `draw_plot` are not mapping coordinates, but relative coordinates on our `ggdraw` canvas where (0,0) is bottom left and (1,1) is top right. Here, we will draw our `dem.plot` at full size and then draw our inset smaller in the bottom right corner.

```
final =
ggdraw() +
  draw_plot(dem.plot) +
  draw_plot(inset, x = 0.75, width = 0.25,  y = 0.1, height = 0.25)


final
```



It can take a bit of trial and error to get plots drawn in the right place and it is usually not worth trying to get them looking perfect in your graphical window in RStudio because the exported figure can look different - for reasons explained below.

10. Exporting maps

Once we are happy with our figure, the final step is to export it as a .jpeg, .tiff, .pdf, or some other image format that we can include in a report, paper or thesis. The function for exporting `ggplot` objects, including multiparts made with `patchwork` or `cowplot` is `ggsave`. The first arguments that `ggsave` needs are the filepath to export to (relative to the working directory) and the name of the plot object you want to export. You typically also set the dimensions of the plot you want to create (in inches by default) and the resolution (in dpi: 300 is the minimum recommended, 600 is often better). Here we export an 8 x 6 inch .jpeg of our leopard conflict map:

```
ggsave('final.jpeg',plot = final, width = 8, height = 6, dpi = 300)
```

A note on aspect ratios. When exporting maps from R using `ggsave` - and particularly multi-part maps - the dimensions you set for your exported image will often have a big influence on the end result. That is because the aspect ratio of maps is fixed and they cannot be stretched to fill the available space. Therefore, *composing your maps to look perfect in your RStudio window is generally not recommended.* It is better to export your plot at the dimensions and resolution you want and then tweak the code to make it look right in the exported version.

11. Conclusion

Ultimately, if you want total flexibility in cartography, sometimes you have no choice but to go to a GUI-based software like QGIS. However, hopefully this tutorial demonstrates that it is possible to produce attractive, publication-ready maps using R, without needing to export layers or interrupt your workflow. The code may appear dense at first, but if you break it down into the series of logical steps that we have just been through, progressively adding and formatting elements on the map, then there is nothing to worry about.