lecture-8

April 3, 2024

1 Visualization

Professional geospatial visualization and cartography is usually carried in dedicated software packages like Adobe Illustrator or ArcGIS. Most maps produced using programming tend to be simplistic and not particularly inspiring. But does this always have to be the case? There are a few specialists who have been able to produce visualling appealing maps and geovisualizations in Python. In this demo, we will demonstrate how.

1.1 Topographic maps

We will use **Global Multi-resolution Terrain Elevation Data (GMTED2010)** which was produced by the USGS and the National Geospatial-Intelligence Agency (NGA). The data has a spatial resolution of 7.5 arc-seconds and can be downloaded from the Earth Explorer.

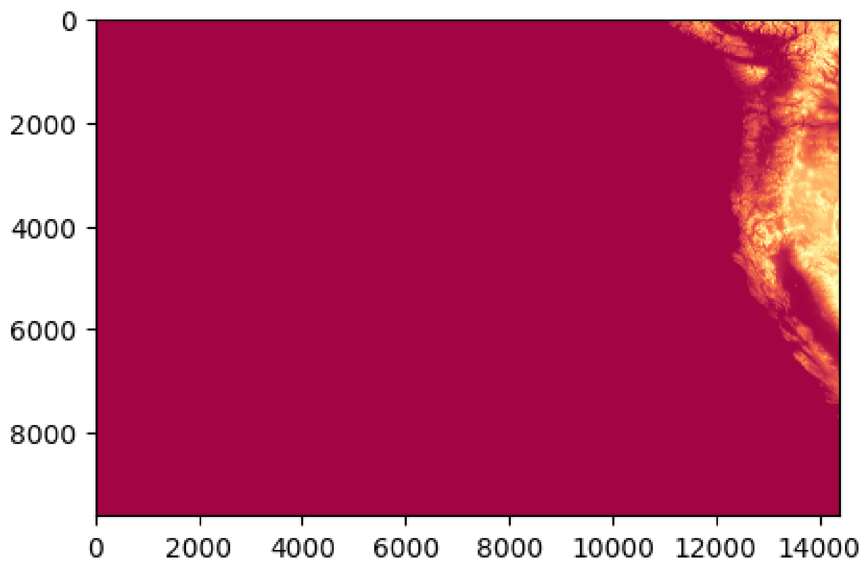
1.1.1 Read elevation data

The first thing we do is open and read the data using `rasterio`. Remember that `open` methods produces a rasterio **dataset object**, which contains not only the elevation data but also information about the projection and extent of the image. The `read` method reads the data in the dataset object into a **2D `numpy` array**.

[3]:
```
import rasterio
import matplotlib.pyplot as plt
import numpy as np

file = rasterio.open('data/30n150w_20101117_gmted_mea075.tif')
dataset = file.read()
print(dataset.shape)
```

```
(1, 9600, 14400)
```

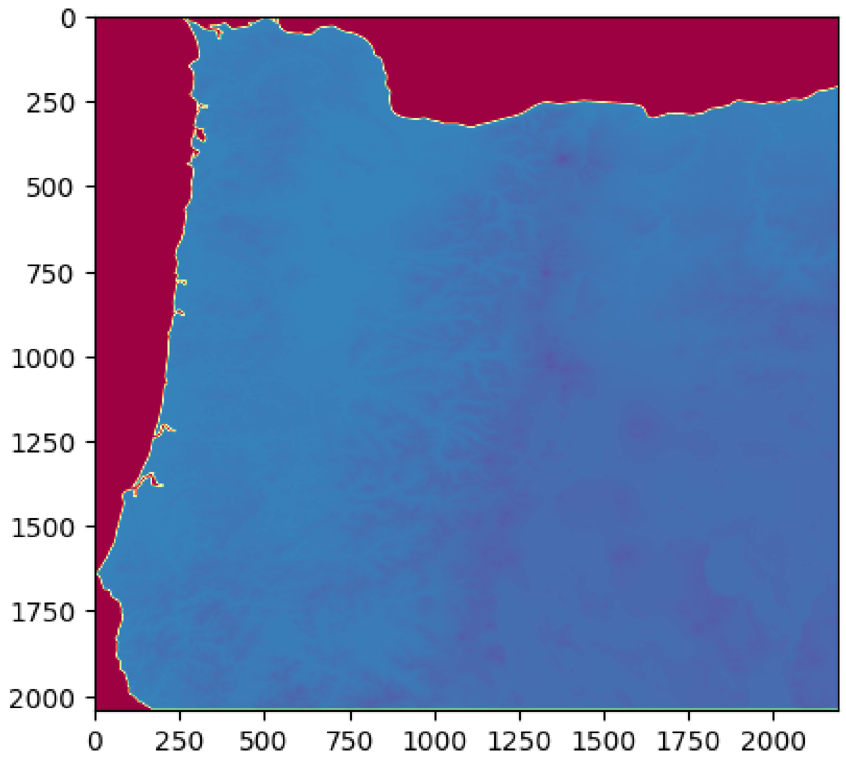
[66]:
```
# Plot data
fig, ax = plt.subplots(figsize=(5,5))
ax.imshow(dataset[0,:,:], cmap='Spectral')
plt.show()
```



1.1.2 Clip to study region

There is a lot of ocean in this plot because geospatial data is often tiled and usually never quite aligns with our study area. Luckily, `rasterio` has a useful method to clip rasters based on a georeferenced shape, e.g. a polygon. In this case we can use an polygon of Oregon which can be downloaded from the Natural Earth database to clip the raster using the `rasterio.mask.mask` function.

[65]:
```
import geopandas as gpd
from shapely.geometry import mapping
from rasterio import mask as msk

# Read Oregon shapefile
oregon_poly = gpd.read_file('data/oregon.shp')

# Clip raster
clipped_array, clipped_transform = msk.mask(file, [mapping(oregon_poly.iloc[0].
  ↪geometry)], crop=True)

# Plot
plt.figure(figsize=(5,5))
plt.imshow(clipped_array[0], cmap='Spectral')
plt.show()
```



Now we have only elevation values that correspond to Oregon. However, by default `rasterio.mask.mask` will assign all values that are not within the Oregon polygon as **zero**. While this is sensible, these zeros will make the plotting tricky because they act as an anchor at the bottom of the **colourmap**. If there is a large gap between zero and the minimum elevation value in Oregon, then the map above will be produced, with real data in one half of the colourmap and the other half of the colourmap absent because there is a gap between zero and the minimum of the real data.

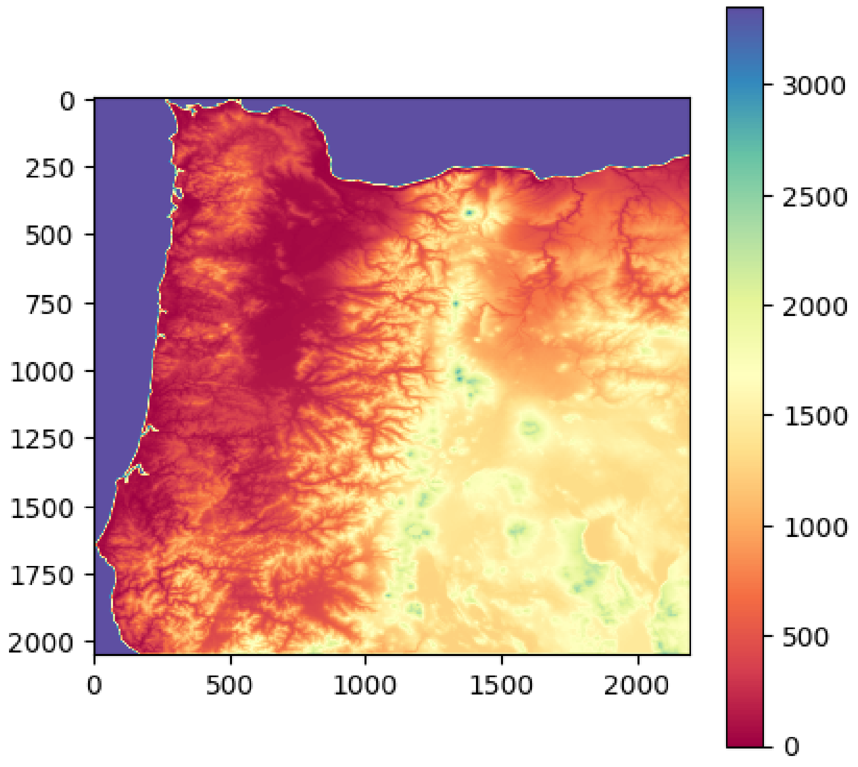
1.1.3 Re-assign nodata values

Fortunately, the `mask` function allows us to explicitly set a specific value to grid cells that are not within the Oregon polygon with the `nodata` keyword argument. In the function below we pass the Oregon **GeoDataFrame** and the `rasterio` **dataset object**.

Notice that the `mask` function is called twice, first it is called as above and values not within the Oregon polygon are returned as zeros. On the second occasion, the `nodata` argument is used and the values not part of the Oregon polygon are set to be **one greater than the maximum value** in the Oregon topography dataset (as calculated with the first mask). The result is that we now have a dataset with no natural gaps. Also being returned by this function is the `value_range` variable. This corresponds to the range between the smallest and largest value in `oregon_topography` array and is useful for constructing colourmaps later.

```
[59]: def clip_raster(gdf, img):
          clipped_array, clipped_transform = msk.mask(img, [mapping(gdf.iloc[0].
       ↪geometry)], crop=True)
          clipped_array, clipped_transform = msk.mask(img, [mapping(gdf.iloc[0].
       ↪geometry)],
                                                           crop=True, nodata=(np.
       ↪amax(clipped_array[0]) + 1))
          clipped_array[0] = clipped_array[0] + abs(np.amin(clipped_array))
          value_range = np.amax(clipped_array) + abs(np.amin(clipped_array))
          return clipped_array, value_range
```

```
[64]: oregon_topography, value_range = clip_raster(oregon_poly, file)

      plt.figure(figsize=(5,5))
      c = plt.imshow(oregon_topography[0], cmap='Spectral')
      plt.colorbar(c)
      plt.show()
```



1.1.4 Construct a colormap

Now we need to construct an appropriate **colourmap**. Here we make a five-class linear colormap using ColorBrewer as inspiration.

[7]:
```
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import colors

oregon_colormap = LinearSegmentedColormap.from_list('oregon',␣
 ↪['#edf8fb','#b2e2e2','#66c2a4','#2ca25f','#006d2c'],
                                                    N=value_range)
```

```
All colors can be described using an **RGB triplet** or **hexadecimal** format (a hex triplet)
```

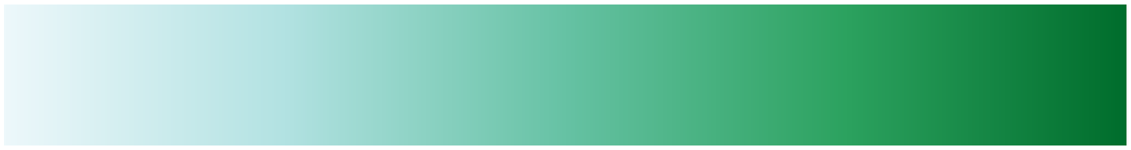
We still need to deal with the values that are not within Oregon. The solution is to construct a **colormap** with enough colours that each unique value within the Oregon elevation data has it's own colour, then replace the last colour in the colormap with our **background color**.

[8]:
```
from matplotlib.colors import ListedColormap

# Define an RGBA color
background_color = np.array([0.9882352941176471, 0.9647058823529412, 0.
 ↪9607843137254902, 1.0])

newcolors = oregon_colormap(np.linspace(0, 1, value_range))
newcolors = np.vstack((newcolors, background_color))
oregon_colormap = ListedColormap(newcolors)
```

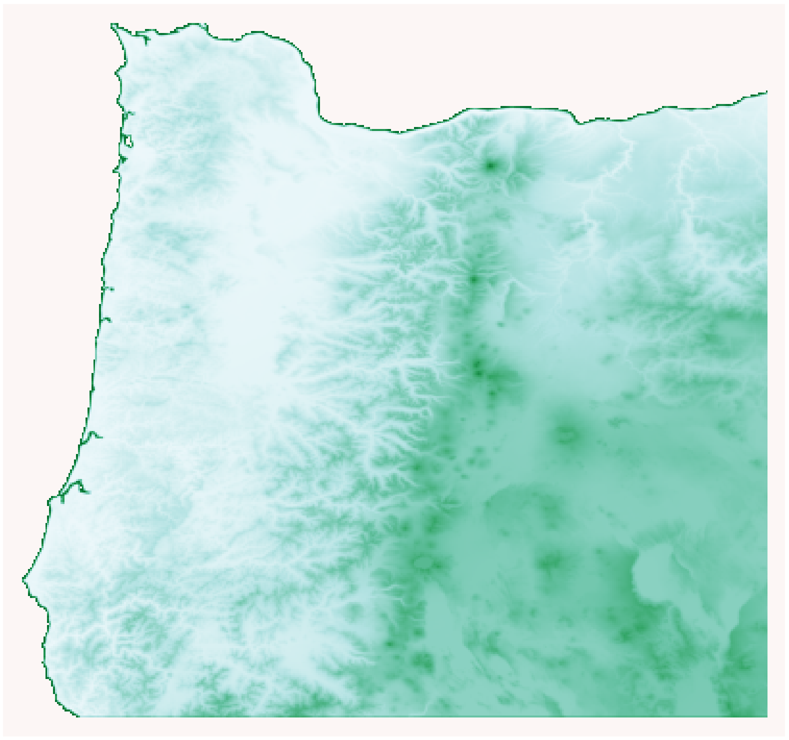
```
This time we defined the background color using an **RGB triplet** + an alpha value which defi
```

[9]:
```
oregon_colormap
```

[9]:



[10]:
```
# Plot
fig = plt.figure(facecolor='#FCF6F5FF', figsize=(5,5))
ax = plt.axes()
plt.imshow(oregon_topography[0], cmap=oregon_colormap)
ax.axis('off')
plt.show()
```



1.1.5 Adding a hillshade

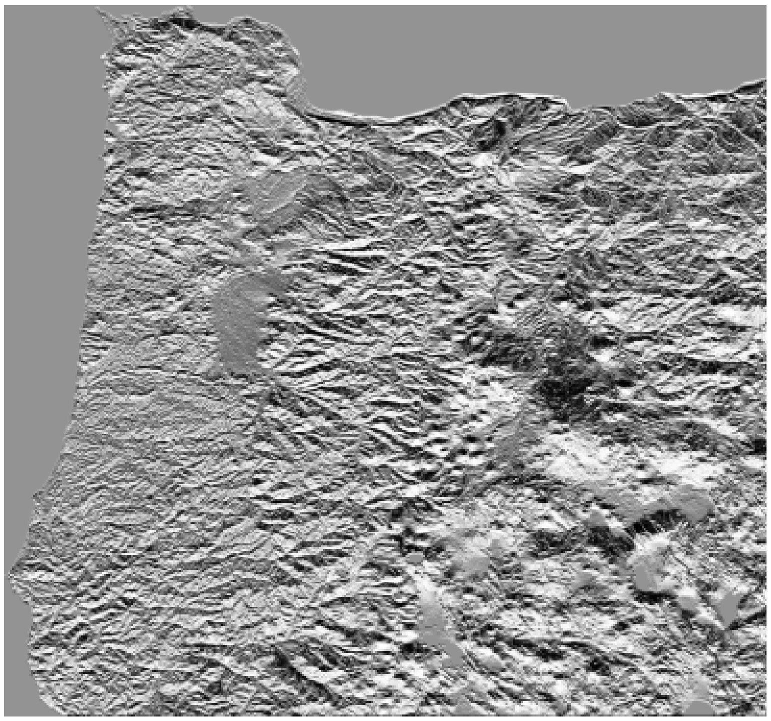
The final step is to add a hillshade to mimic light shining on the topography. A hillshade is a 3D representation of a surface and are generally rendered in greyscale. The darker and lighter colors represent the shadows and highlights that you would visually expect to see in a terrain model.

We will use EarthPy to generate our hillshade data. There are two parameters that can be tuned and they give very different results depending on the dataset. The first is the **azimuth value** which can range from 0-360 degrees and relates to where the light source is shining from. 0 degrees corresponds to a light source pointing due north. The second is the altitude of the light source and can range from 1-90. Below are some examples demonstrating how changing the **azimuth value** produces vastly different results.

```
`EarthPy` can be installed by running `pip install earthpy` from the terminal or command promp
```

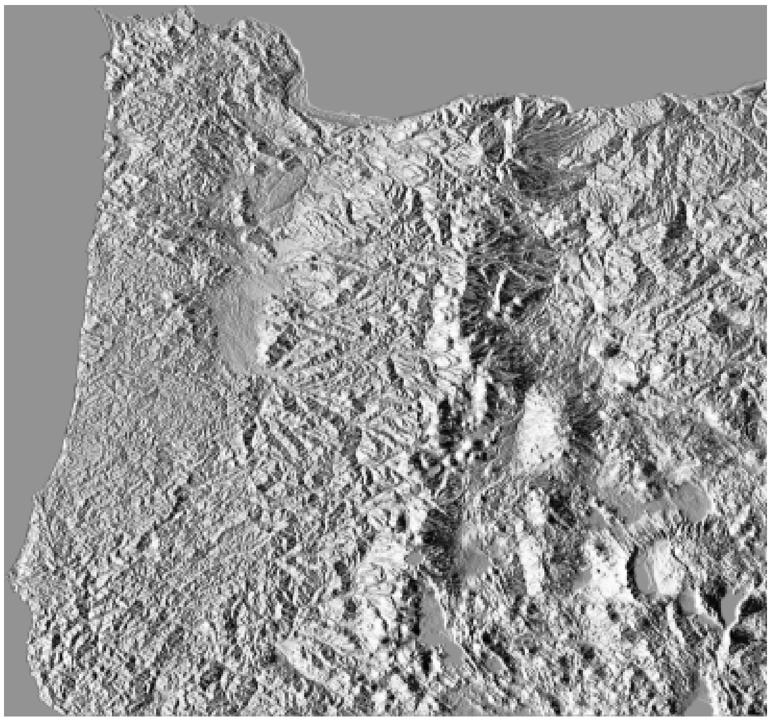
[13]:
```
import earthpy.spatial as es

hillshade = es.hillshade(oregon_topography[0],
                         azimuth=0, altitude=1)
plt.imshow(hillshade, cmap='Greys')
plt.axis('off')
```

[13]:
```
(-0.5, 2191.5, 2043.5, -0.5)
```



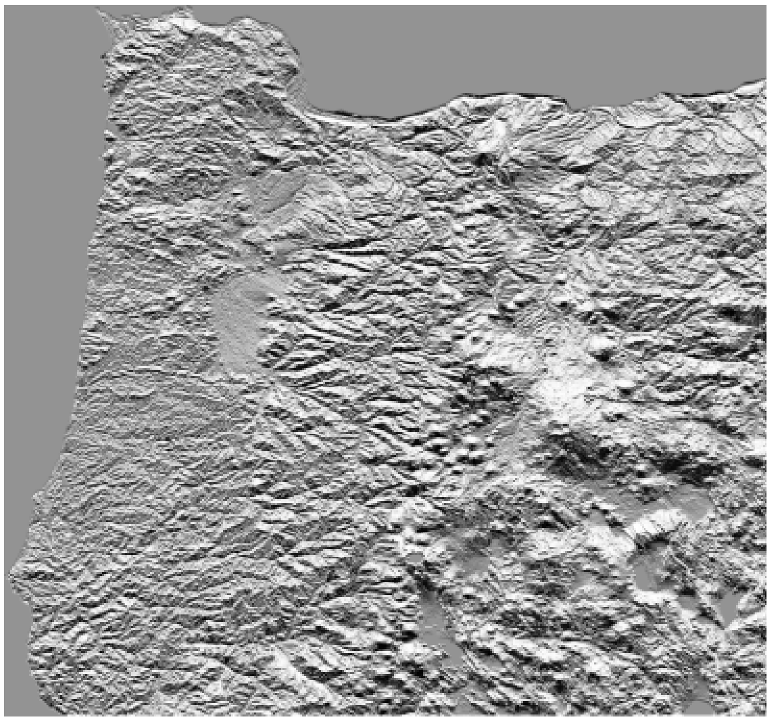
```
[14]: hillshade = es.hillshade(oregon_topography[0],
                              azimuth=90, altitude=1)
      plt.imshow(hillshade, cmap='Greys')
      plt.axis('off')
```

```
[14]: (-0.5, 2191.5, 2043.5, -0.5)
```



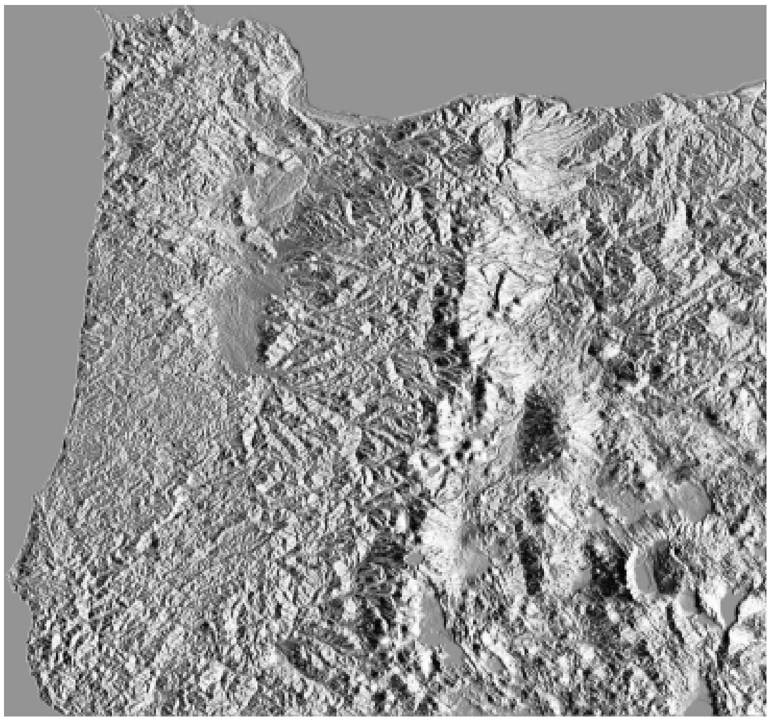
```
[15]: hillshade = es.hillshade(oregon_topography[0],
                              azimuth=180, altitude=1)
      plt.imshow(hillshade, cmap='Greys')
      plt.axis('off')
```

```
[15]: (-0.5, 2191.5, 2043.5, -0.5)
```



```
[16]: hillshade = es.hillshade(oregon_topography[0],
                             azimuth=270, altitude=1)
      plt.imshow(hillshade, cmap='Greys')
      plt.axis('off')
```

```
[16]: (-0.5, 2191.5, 2043.5, -0.5)
```

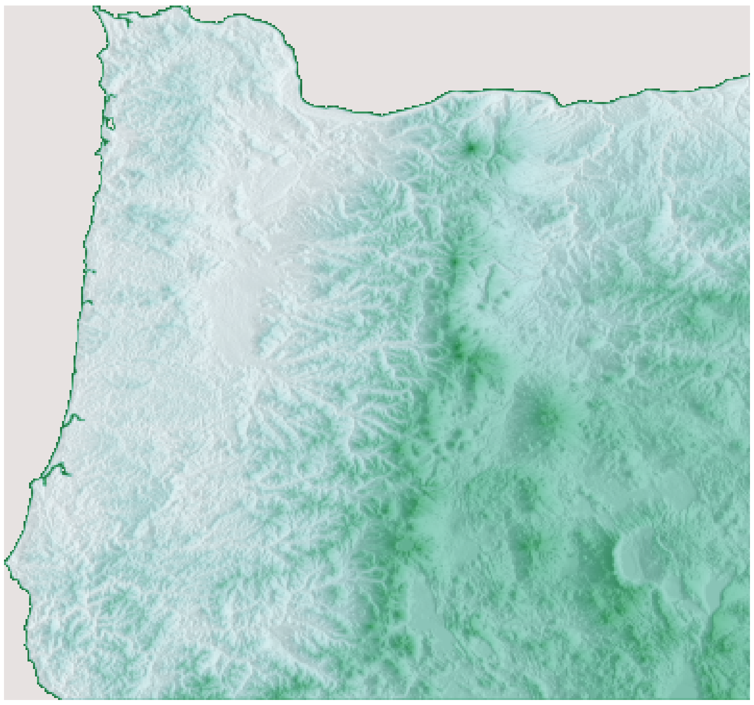


An azimuth value of 90 produces a nice shadow on the eastern side of the Cascades and Crater Lake. Definitely play around with these values though because there may be even better hillshades.

1.1.6 Plot map

We can now plot the finished product. To do is we first plot the Oregon topography, then overlay the hillshade with a small **alpha value** that sets transparency.

[62]:
```
fig, ax = plt.subplots(figsize=(5,5))
im = plt.imshow(oregon_topography[0],
                cmap=oregon_colormap)
ax.imshow(hillshade, cmap='Greys', alpha=0.2)
ax.axis('off')
plt.show()
```



1.1.7 Add a colorbar and scalebar

If we were producing a figure for a journal article, we may also want to add a **colorbar** and a **scalebar**. A colorbar is fairly simple in `Matplotlib` but the default option is a little clumsy. The code below allows us to make the colorbar match the size of the plot.

Adding a scalebar in is a little more annoying because `Matplotlib` doesn't have any native functions to do this. But there is a package called `matplotlib_scalebar` that we can install which does the trick.

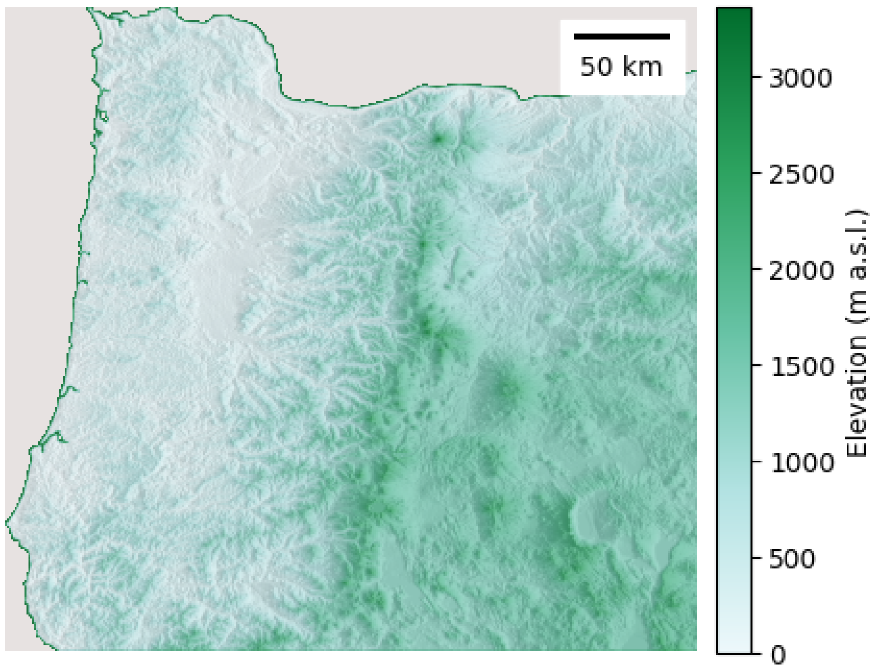
We just have to find the size of each pixel in meters. We can use this website to convert 7.5 arc-seconds to meters at a specific latitude.

```
`matplotlib_scalebar` can be installed by running `pip install matplotlib-scalebar` from the te
```

[34]:
```python
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib_scalebar.scalebar import ScaleBar
```

[63]:
```python
fig, ax = plt.subplots(figsize=(5,5))
im = plt.imshow(oregon_topography[0], cmap=oregon_colormap)
ax.imshow(hillshade, cmap='Greys', alpha=0.2)
ax.axis('off')

# Colorbar
divider = make_axes_locatable(ax)
```

```
cax = divider.append_axes('right', size='5%', pad=0.10)
fig.colorbar(im, cax=cax, orientation='vertical',
             label='Elevation (m a.s.l.)')

#Scalebar
scalebar = ScaleBar(164,"m", length_fraction=0.2, pad=0.5,
                    border_pad=0.5, location=1)
ax.add_artist(scalebar)
plt.show()
```



There we have it, a visually appealing topographic map of *most* of Orgeon ready to hang on a wall (or sell on Etsy...).

1.2 Maps with data

In the second part of this lecture, we will use Python to produce another type of thematic map. But this instead of representing elevation data, we will visualize Census Bureau data.

We can automatically download Census Bureau data from the 2019 American Community Survey using CenPy.

[77]:
```
from cenpy import products
acs = products.ACS(2019)
```

1.2.1 Search for data

Inspired by the article "Plumbing Poverty: Mapping Hot Spots of Racial and Geographic Inequality in U.S. Household Water Insecurity" by Shiloh Deitz and Katie Meehan (2019), we will search for data that determines whether household have adequate plumbing facilities.

The Census Bureau defines "complete plumbing facilities" as (1) piped hot and cold water, (2) a flush toilet, and (3) a bathtub or shower, all located within the housing unit and used only by occupants. We can find the variable **code** in the data tables by searching for the word "**plumbing**".

[160]:
```
acs.filter_tables('PLUMBING', by='description')
```

[160]:
```
                                                         description  \
table_name
B25016      TENURE BY PLUMBING FACILITIES BY OCCUPANTS PER…
B25047              PLUMBING FACILITIES FOR ALL HOUSING UNITS
B25048         PLUMBING FACILITIES FOR OCCUPIED HOUSING UNITS
B25049                          TENURE BY PLUMBING FACILITIES
B25050      PLUMBING FACILITIES BY OCCUPANTS PER ROOM BY Y…
B99259                      ALLOCATION OF PLUMBING FACILITIES

                                                             columns
table_name
B25016      [B25016_001E, B25016_002E, B25016_003E, B25016…
B25047                [B25047_001E, B25047_002E, B25047_003E]
B25048                [B25048_001E, B25048_002E, B25048_003E]
B25049      [B25049_001E, B25049_002E, B25049_003E, B25049…
B25050      [B25050_001E, B25050_002E, B25050_003E, B25050…
B99259                [B99259_001E, B99259_002E, B99259_003E]
```

[89]:
```
# Print list of tables
acs.filter_variables('B25047')
```

[89]:
```
                                                              label  \
B25047_003E  Estimate!!Total:!!Lacking complete plumbing fa…
B25047_002E     Estimate!!Total:!!Complete plumbing facilities
B25047_001E                                   Estimate!!Total:

                                               concept predicateType   group  \
B25047_003E  PLUMBING FACILITIES FOR ALL HOUSING UNITS           int  B25047
B25047_002E  PLUMBING FACILITIES FOR ALL HOUSING UNITS           int  B25047
B25047_001E  PLUMBING FACILITIES FOR ALL HOUSING UNITS           int  B25047

            limit predicateOnly hasGeoCollectionSupport  \
B25047_003E     0           NaN                     NaN
B25047_002E     0           NaN                     NaN
B25047_001E     0           NaN                     NaN

                                            attributes required
```

```
B25047_003E  B25047_003EA,B25047_003M,B25047_003MA      NaN
B25047_002E  B25047_002EA,B25047_002M,B25047_002MA      NaN
B25047_001E  B25047_001EA,B25047_001M,B25047_001MA      NaN
```

1.2.2 Download data

We will focus our analysis on western United States and download data at the **county** level. Of course, a more local analysis could look at smaller spatial scales such as the tract or block group level.

[90]:
```
# Download data
wa_plumbing = products.ACS(2019).from_state('Washington',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
or_plumbing = products.ACS(2019).from_state('Oregon',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
ca_plumbing = products.ACS(2019).from_state('California',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
az_plumbing = products.ACS(2019).from_state('Arizona',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
nm_plumbing = products.ACS(2019).from_state('New Mexico',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
id_plumbing = products.ACS(2019).from_state('Idaho',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
az_plumbing = products.ACS(2019).from_state('Arizona',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
ut_plumbing = products.ACS(2019).from_state('Utah',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
id_plumbing = products.ACS(2019).from_state('Idaho',
                                            level='county',
                                            variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
```

```
nv_plumbing = products.ACS(2019).from_state('Nevada',
                                           level='county',
                                           variables=['B25047_001E',␣
 ↪'B25047_002E', 'B25047_003E'])
```

```
/Users/johnnyryan/.gds/lib/python3.10/site-packages/cenpy/products.py:767:
FutureWarning: The `op` parameter is deprecated and will be removed in a future
release. Please use the `predicate` parameter instead.
  return self._from_name(state, variables, level, "States", **kwargs)
/Users/johnnyryan/.gds/lib/python3.10/site-packages/cenpy/products.py:767:
FutureWarning: The `op` parameter is deprecated and will be removed in a future
release. Please use the `predicate` parameter instead.
  return self._from_name(state, variables, level, "States", **kwargs)
/Users/johnnyryan/.gds/lib/python3.10/site-packages/cenpy/products.py:767:
FutureWarning: The `op` parameter is deprecated and will be removed in a future
release. Please use the `predicate` parameter instead.
  return self._from_name(state, variables, level, "States", **kwargs)
```

[109]:
```
wa_plumbing.head()
```

[109]:
```
   GEOID                                           geometry  B25047_001E  \
0  53059  POLYGON ((-13608723.150 5728087.660, -13608731…       5865.0
1  53049  POLYGON ((-13821555.220 5895422.380, -13821712…      16213.0
2  53027  POLYGON ((-13837615.950 5982098.020, -13837721…      36273.0
3  53029  POLYGON ((-13677015.870 6150668.720, -13677016…      41902.0
4  53037  POLYGON ((-13507418.850 5986840.900, -13507418…      23801.0

   B25047_002E  B25047_003E                           NAME state county
0       5786.0         79.0      Skamania County, Washington    53    059
1      15931.0        282.0       Pacific County, Washington    53    049
2      35807.0        466.0  Grays Harbor County, Washington    53    027
3      41557.0        345.0        Island County, Washington    53    029
4      23176.0        625.0      Kittitas County, Washington    53    037
```

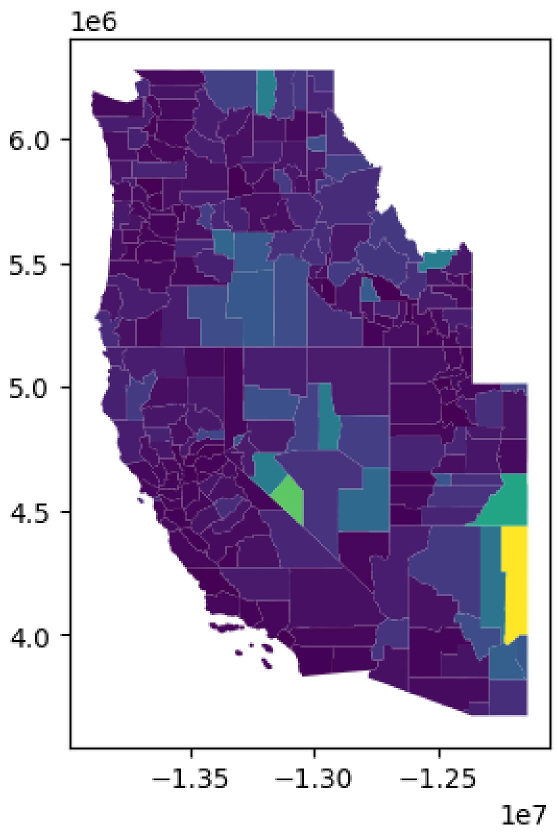
[146]:
```
plumbing = pd.concat([wa_plumbing, or_plumbing, ca_plumbing,
                      az_plumbing, ut_plumbing, id_plumbing,
                      nv_plumbing])

# Compute proportion of household with inadequate plumbing
plumbing['lack_plumbing_percent'] = plumbing['B25047_003E'] /␣
 ↪plumbing['B25047_001E']
```

1.2.3 Simple chloropleth map

GeoPandas provides a high-level interface to the Matplotlib library for making maps. It is as easy as using the plot() method on a GeoDataFrame.

[148]:
```
plumbing.plot('lack_plumbing_percent')
```

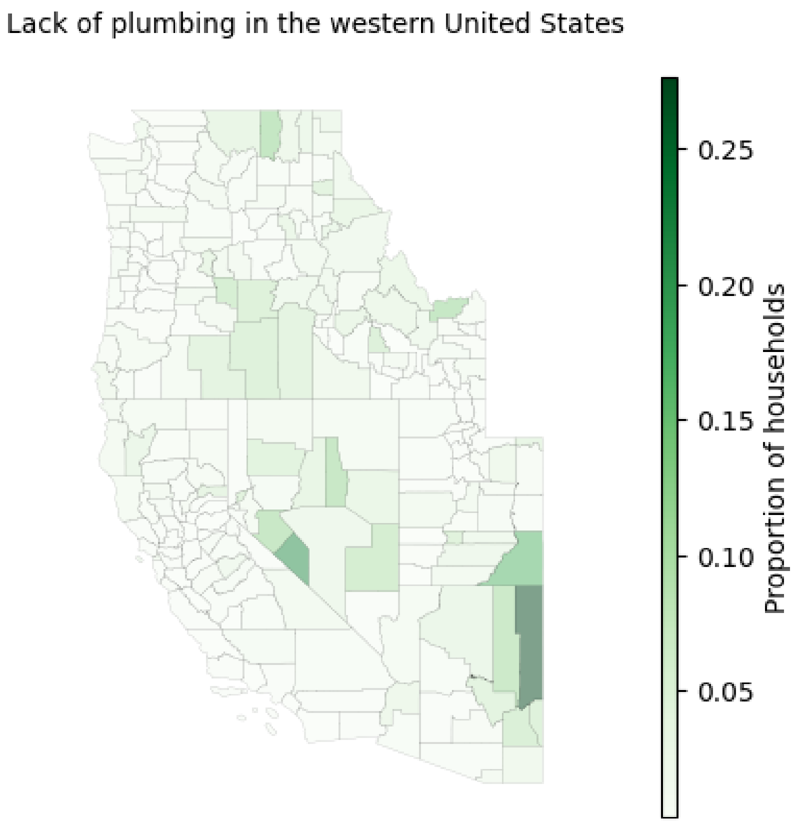
[148]: <Axes: >



1.2.4 Improving the chloropleth map

This is useful for a quick look over the data but we would prefer to customize our plot. For instance, we should add a title, colorbar, and nicer colormap.

[161]:
```python
fig, ax = plt.subplots(figsize=(8,5))

plt.title('Lack of plumbing in the western United States', fontsize=10)
ax.set_axis_off()
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="3%", pad=0.5,alpha=0.5)
plumbing.plot('lack_plumbing_percent', ax=ax, alpha=0.5, cmap='Greens',
              edgecolor='k', legend=True, cax=cax, linewidth=0.1)
plt.ylabel('Proportion of households', fontsize=10)
plt.show()
```



1.2.5 Interactive plots

Alongside static plots, `GeoPandas` can produce interactive maps based on the `Folium` library, which is itself based on `Leaflet`.

[162]:
```
m = plumbing.explore(
     column="lack_plumbing_percent", # make choropleth based on column
     scheme="naturalbreaks",         # use mapclassify's natural breaks scheme
     tooltip="lack_plumbing_percent",# show column value in tooltip (on hover)
     popup=True,                     # show all values in popup (on click)
     tiles="CartoDB positron",       # use "CartoDB positron" tiles
     cmap="Greens",                  # use "Greens" matplotlib colormap
     style_kwds=dict(color="black", weight=0.5) # use black outline with weight␣
 ↪of 1
    )
m
```

[162]:
```
<folium.folium.Map at 0x32769f4f0>
```

1.3 Conclusions

Maps made using programming don't have to be terrible - it is possible to produce visually appealing maps in Python. They may never be quite as good as what professional cartographers can produce in Adobe Illustrator but they have other advantages. Once the code is written, we can quickly make edits or add another dataset. I hope this demo has inspired you to improve the quality of your Python plots!

Content in this lecture was inspired by Adam Symington and this tutorial.