This book provides you with the resources to successfully develop your own GIS application in Python. The book begins by walking you through the loading and saving of data structures before you start to build the look and feel of your application and create its interactive map window. You'll then move on to managing, editing, and analyzing spatial data from within the application and finish with instructions for packaging the application for distribution.

By the end of the book, you should be able to use your GIS application as a template for further development, with the potential to expand and customize it to suit your needs.

Who this book is written for
This book is ideal for Python programmers who are tasked with or wish to make a special-purpose GIS application. Analysts, political scientists, geographers, and GIS specialists seeking a creative platform to experiment with cutting-edge spatial analysis, but who are still only beginners in Python, will also find this book beneficial. Familiarity with Tkinter application development in Python is preferable but not mandatory.

What you will learn from this book
- Design and structure a basic GIS application based on existing tools
- Customize Tkinter application widgets to make your very own toolkit of flexible and reusable GIS widgets
- Load, manipulate, and save spatial data, both vector and raster
- Create an interactive and zoomable window for visualizing spatial data
- Manage and organize spatial data
- Create the beginnings of a comprehensive Python library for many types of geospatial tasks, and use this API as the engine behind your application
- Package your final application into a lightweight and portable executable application

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 'Managing and Organizing Geographic Data'
- A synopsis of the book’s content
- More information on Python Geospatial Development Essentials
About the Author

Karim Bahgat holds an MA in peace and conflict transformation from the University of Tromsø in Norway, where he focused on the use of geographic information systems (GIS), opinion survey data, and open source programming tools in conflict studies. Since then, he has been employed as a research assistant for technical and geospatial work at the Peace Research Institute Oslo (PRIO) and the International Law and Policy Institute (ILPI). Karim was part of the early prototyping of the PRIO-GRID unified spatial data structure for social science and conflict research, and is currently helping develop a new updated version (https://www.prio.org/Data/PRIO-GRID/).

His main use of technology, as a developer, has been with Python programming, geospatial tools and mapping, the geocoding of textual data, data visualization, application development, and some web technology. Karim is the author of a journal article publication, numerous data- and GIS-oriented Python programming libraries, the Easy Georeferencer free geocoding software, and several related technical websites, including www.pythongisresources.wordpress.com.
Preface

Python has become the language of choice for many in the geospatial industry. Some use Python as a way to automate their workflows in software, such as ArcGIS or QGIS. Others play around with the nuts and bolts of Python's immense variety of third-party open source geospatial toolkits.

Given all the programming tools available and the people already familiar with geospatial software, there is no reason why you should have to choose either one or the other. Programmers can now develop their own applications from scratch to better suit their needs. Python is, after all, known as a language for rapid development.

By developing your own application, you can have fun with it, experiment with new visual layouts and creative designs, create platforms for specialized workflows, and tailor to the needs of others.

What this book covers

*Chapter 1, Preparing to Build Your Own GIS Application*, talks about the benefits of developing a custom geospatial application and describes how to set up your development environment, and create your application folder structure.

*Chapter 2, Accessing Geodata*, implements the crucial data loading and saving capabilities of your application for both vector and raster data.

*Chapter 3, Designing the Visual Look of Our Application*, creates and puts together the basic building blocks of your application's user interface, giving you a first look at what your application will look like.

*Chapter 4, Rendering Our Geodata*, adds rendering capabilities so that the user can interactively view, zoom, and pan data inside the application.
Preface

Chapter 5, Managing and Organizing Geographic Data, creates a basic functionality for splitting, merging, and cleaning both the vector and raster data.

Chapter 6, Analyzing Geographic Data, develops basic analysis functionality, such as overlay statistics, for vector and raster data.

Chapter 7, Packaging and Distributing Your Application, wraps it all up by showing you how to share and distribute your application, so it is easier for you or others to use it.

Chapter 8, Looking Forward, considers how you may wish to proceed to further build on, customize, and extend your basic application into something more elaborate or specialized in whichever way you want.
Managing and Organizing Geographic Data

Now that we have an up-and-running explorative application, we can move on to developing some more day-to-day practical features. A common task for users of geographic data is to prepare, clean, restructure, and organize data. In this chapter, you will do the following:

• Create a window for inspecting the basic properties of each layer

• Build some convenience functions for commonly needed management tasks, and later add them to the user interface. These functions are as follows:
  ° Operations on individual layers being available when the user right-clicks on each layer in the LayersPane widget (splitting, geometry cleaning, and resampling)
  ° Batch operations on multiple layers available as buttons in the top ribbon area (merging and mosaicking)

• Assign dialogue windows to set the parameters when running each tool

Creating the management module

We start by creating a separate submodule to contain the functionality, one for vector, and one for raster. First, create the `vector/manager.py` file and start it up with the following imports:

```python
import itertools, operator
from .data import *
```
Next, create the file `raster/manager.py` as follows:

```python
import PIL, PIL.Image
```

To make these manager modules available to their respective vector and raster parent package, add the following import statement to both `vector/__init__.py` and `raster/__init__.py`:

```python
import . import manager
```

## Inspecting files

As the most basic way to organize and troubleshoot one's files, one often needs to inspect the properties and details of one's data and loaded layers. This information is usually available in a separate layer options window. Later in this chapter, we will make this window accessible by right-clicking on a layer, and clicking on Properties under the **Layer-specific right-click functions** subheading.

We define a template class for this type of window, with support for tabbed windows using our ribbon class, and create a convenience method for adding information in a nicely formatted way. This is done in the `app/dialogues.py` module. Since we have not yet set up the contents of `app/dialogues.py`, we also have to set up its imports and styling, as shown in the following code snippet:

```python
import Tkinter as tk
import ScrolledText as tkst # a convenience module that ships with Tkinter

from .toolkit.popups import *
from .toolkit.ribbon import *
from .toolkit import theme
from .. import vector, raster

style_layeroptions_info = {"fg": theme.font1["color"],
                           "font": theme.font1["type"],
                           "relief": "flat"}

class LayerOptionsWindow(Window):
    def __init__(self, master, **kwargs):
        # Make this class a subclass of tk.Menu and add to it
        Window.__init__(self, master, **kwargs)

        # Make the top ribbon selector
```
def add_info(self, tab, label, value):
    row = tk.Frame(tab, bg=tab.cget("bg"))
    row.pack(fill="x", anchor="n", pady=5, padx=5)

    # place label
    header = tk.Label(row, text=label, bg=tab.cget("bg"),
                      **style_layeroptions_info)
    header.pack(side="left", anchor="nw", padx=3)

    # place actual info text
    value = str(value)
    info = tk.Entry(row, width=400,
                    disabledbackground="white", justify="right",
                    **style_layeroptions_info)
    info.pack(side="right", anchor="ne", padx=3)
    info.insert(0, value)
    info.config(state="readonly")
    return info

Vector and raster data will typically have very different properties, so we make a separate window for each. First, for vector layers:

Here is the code for the same:

```python
class VectorLayerOptionsWindow(LayerOptionsWindow):
    def __init__(self, master, layeritem, statusbar, **kwargs):
        # Make this class a subclass of tk.Menu and add to it
        LayerOptionsWindow.__init__(self, master, **kwargs)
        self.layeritem = layeritem
```
Managing and Organizing Geographic Data

# GENERAL OPTIONS TAB

general = self.ribbon.add_tab("General")

# add pieces of info
self.source = self.add_info(general, "Source file: ", layeritem.renderlayer.data.filepath)
sself.proj = self.add_info(general, "Projection: ", layeritem.renderlayer.data.crs)
sself.bbox = self.add_info(general, "Bounding box: ", layeritem.renderlayer.data.bbox)
sself.fields = self.add_info(general, "Attribute fields: ", layeritem.renderlayer.data.fields)
sself.rows = self.add_info(general, "Total rows: ", len(layeritem.renderlayer.data))

# Set starting tab
self.ribbon.switch(tabname="General")

Then, for raster layers:

class RasterLayerOptionsWindow(LayerOptionsWindow):
    def __init__(self, master, layeritem, statusbar, **kwargs):
        # Make this class a subclass of tk.Menu and add to it
        LayerOptionsWindow.__init__(self, master, **kwargs)
sself.layeritem = layeritem

Here is the code for the same:
### GENERAL OPTIONS TAB

```python
general = self.ribbon.add_tab("General")

# add pieces of info
self.source = self.add_info(general, "Source file: ",
layeritem.renderlayer.data.filepath)
self.proj = self.add_info(general, "Projection: ",
self.layeritem.renderlayer.data.crs)
self.dims = self.add_info(general, "Dimensions: ", "%i, 
%i"%(self.layeritem.renderlayer.data.width,
self.layeritem.renderlayer.data.height))
self.bands = self.add_info(general, "Raster bands: ",
"%i"%len(self.layeritem.renderlayer.data.bands))
self.transform = self.add_info(general, "Transform: ",
self.layeritem.renderlayer.data.info)
self.bbox = self.add_info(general, "Bounding box: ",
layeritem.renderlayer.data.bbox)
```

### Set starting tab

```python
self.ribbon.switch(tabname="General")
```

**Organizing files**

Traditionally, when working in a GIS application, one first seeks out the data files one wishes to use from various organizational websites. Ideally, one stores these in some logically organized folder structure on the local computer, and from there, one can load the data into the GIS application. In this section, we add functionality to help the user manage their files and access and alter basic file contents.

For some great examples of the varied types and sources of GIS data available online, see the list at [http://freegisdata.rtwilson.com/](http://freegisdata.rtwilson.com/).
Vector data
Vector data is very versatile; its table-like data structure means that it can contain data on a wide variety of concepts in a single file, or contain data for only a very specific concept. For practical usage, it is easier if each file is tailored exactly to the data one needs, since these are represented as layers when loaded in the application. There are therefore many cases where the user may wish to reorganize the data to better fit their needs.

Here, we will implement three specific operations for organizing and maintaining vector data: splitting, merging, and cleaning. The following illustration gives a preview of the inputs and outputs of each:

### Splitting
For instance, the user may have a file that groups a diverse set of concepts, but is only interested in working with certain types separately. In such cases, it will be easier to just split the data for each unique occurrence of a field — known as splitting. In terms of data structure, this means slicing the height of the table into multiple tables along with their associated geometries. We do this conveniently using Python's built-in `sorted()` and `itertools.groupby()` functions. A `splitfields` option defines a list of one or more field names to split on so that each unique value combination defines a new split. So, head to the `manager.py` file for vector data and write the following code:

```python
def split(data, splitfields):
    fieldindexes = [index for index, field in enumerate(data.fields) if field in splitfields]
```


Chapter 5

sortedfeatures = sorted(data,
key=operator.itemgetter(*fieldindexes))
grouped = itertools.groupby(sortedfeatures,
key=operator.itemgetter(*fieldindexes))
for splitid,features in grouped:
    outfile = VectorData()
    outfile.fields = list(data.fields)
    for oldfeat in features:
        outfile.add_feature(oldfeat.row, oldfeat.geometry)
yield outfile

Merging

One can also face the opposite scenario, where one wishes to group together a series
of data files spread across multiple files. This is called a merge operation. A merge
operation stacks the rows from multiple tables into one big one, and generally
increases the spatial coverage, since it leads to a bigger collection of geometries.
The output attribute table from this operation also expands horizontally to include
all of the variables/fields from its input files. Finally, remember that the VectorData
instances can only contain one type of geometry (points, lines, or polygons), so trying
to merge layers of different geometry types will result in an error. We implement it
in the following way:

def merge(*datalist):
    #make empty table
    firstfile = datalist[0]
    outfile = VectorData()
    #combine fields from all files
    outfields = list(firstfile.fields)
    for data in datalist[1:]:
        for field in data.fields:
            if field not in outfields:
                outfields.append(field)
    outfile.fields = outfields
    #add the rest of the files
    for data in datalist:
        for feature in data:
            geometry = feature.geometry.copy()
            row = []
            for field in outfile.fields:
                if field in data.fields:
                    row.append( feature[field] )
                else:
                    row.append( "" )
            yield outfile
Managing and Organizing Geographic Data

outfile.add_feature(row, geometry)
#return merged file
return outfile

Geometry cleaning

Geographic data can come from a very wide variety of sources, and this means that their levels of integrity can vary greatly. For instance, there are many rules that govern what is and is not allowed for each geometry type, but not all data producers (including both software and individuals) use the same rules or follow them to the same degree. This can be a problem for GIS processing, analysis applications, and programming libraries if the data is corrupted or not formatted in the way that is expected. Data may also contain unnecessary junk information that doesn't add anything useful (depending on the level of detail needed), making the file size overly large. Geometry cleaning can therefore be a useful feature as a first step when gathering one's data.

To do this, we make a function that loops the geometries of our features. With the help of the Shapely library, we fix "bowtie" errors (polygons only), remove repeat points, and exclude any remaining geometries deemed to be invalid according to the GeoJSON specification. The tolerance argument can be set to higher than zero to reduce file size, but note that this alters the shape of the geometries and decreases the level of detail and precision in the output. Refer to the following code:

def clean(data, tolerance=0):
    # create new file
    outfile = VectorData()
    outfile.fields = list(data.fields)

    # clean
    for feat in data:
        shapelyobj = feat.get_shapely()

        # try fixing invalid geoms
        if not shapelyobj.is_valid:
            if "Polygon" in shapelyobj.type:
                # fix bowtie polygons
                shapelyobj = shapelyobj.buffer(0.0)

        # remove repeat points (tolerance=0)
        # (and optionally smooth out complex shapes, tolerance > 0)
        shapelyobj = shapelyobj.simplify(tolerance)
# if still invalid, do not add to output
if not shapelyobj.is_valid:
    continue

# write to file
geojson = shapelyobj.__geo_interface__
outfile.add_feature(feat.row, geojson)

return outfile

For more on polygon bowtie errors, visit:
http://stackoverflow.com/questions/20833344/
fix-invalid-polygon-python-shapely

**Raster data**

There are many common raster file management functionalities that you may wish to implement. Here, we will only focus on two of them: mosaicking and resampling, as seen in the following screenshot:

![Mosaick & Resample Diagram]

To implement these functionalities, we will take advantage of the image processing functionality from the PIL library. Since we are using an imaging library not primarily intended for geospatial data, the following code should be considered highly experimental and mostly for demonstration purposes; you may have to troubleshoot and improve on these methods on your own.
Managing and Organizing Geographic Data

If the main purpose of your application is to process satellite, imagery, and raster data, and you don’t have time or feel comfortable finding your own solution using PIL, you may be better off just adding NumPy, GDAL, and related tools as dependencies. For a list of GDAL’s broad range of functionality for handling raster data, see: https://pcjericks.github.io/py-gdalgr-cookbook/

Mosaicking

Similar to how vector data can be merged together, it is also possible to mosaic multiple adjacent raster datasets into a single larger raster data. The way we implement it here is by creating an align_rasters() function, which takes any number of rasters, automatically finds the coordinate bounding box that contains all, as well as the required pixel dimensions (though we should probably allow some user control here), and uses this information to position each raster to their respective location in a region that bounds all of our rasters. We add this as a function in the raster/manager.py file:

```python
def align_rasters(*rasters):
    "Used internally by other functions only, not by user"
    # get coord bbox containing all rasters
    for rast in rasters: print rast.bbox
    xlefts,ytops,xrights,ybottoms = zip(*[rast.bbox for rast in rasters])
    if xlefts[0] < xrights[0]:
        xleft,xright = min(xlefts),max(xrights)
    else: xleft,xright = max(xlefts),min(xrights)
    if ytops[0] > ybottoms[0]:
        ytop,ybottom = max(ytops),min(ybottoms)
    else: ytop,ybottom = min(ytops),max(ybottoms)

    # get the required pixel dimensions (based on first raster, but should probably allow user to specify)
    xs,ys = (xleft,xright),(ytop,ybottom)
    coordwidth,coordheight = max(xs)-min(xs), max(ys)-min(ys)
    rast = rasters[0]
    orig_xs,orig_ys = (rast.bbox[0],rast.bbox[2]),(rast.bbox[1],rast.bbox[3])
    orig_coordwidth,orig_coordheight = max(orig_xs)-min(orig_xs),
    max(orig_ys)-min(orig_ys)
    widthratio,heightratio = coordwidth/orig_coordwidth,
    coordheight/orig_coordheight
```
reqwidth = int(round(rast.width*widthratio))
reqheight = int(round(rast.height*heightratio))

# position into same coordbbox
aligned = []
for rast in rasters:
    coordbbox = [xleft, ytop, xright, ybottom]
    positioned = rast.positioned(reqwidth, reqheight, coordbbox)
    aligned.append(positioned)
return aligned

Since we now have a way to align and properly position the rasters in space, we can easily mosaic them into a new raster by simply creating a new grid with dimensions that bound all rasters and pasting each raster into it:

def mosaic(*rasters):
    """
    Mosaic rasters covering different areas together into one file.
    Parts of the rasters may overlap each other, in which case we use the value from the last listed raster (the "last" overlap rule).
    """
    # align all rasters, ie resampling to the same dimensions as the first raster
    aligned = align_rasters(*rasters)
    # copy the first raster and reset the cached mask for the new raster
    firstalign, firstmask = aligned[0]
    merged = firstalign.copy()
    del merged._cached_mask
    # paste onto each other, ie "last" overlap rule
    for rast, mask in aligned[1:]:
        merged.bands[0].img.paste(rast.bands[0].img, (0, 0), mask)

    return merged

Note that unlike a vector merge, where overlapping geometries are kept in their original form, raster mosaicking needs a rule for choosing a value when there are overlapping cells. In the previous code, we didn't support any customization of the overlap rule, but instead just pasted each raster on top of each other so that any overlapping cells hold the value of the last raster to be pasted—a so-called "last" rule. You may implement other overlap-rules by looking at the tools available in the PIL library, such as average value with PIL.Image.blend(), or min or max with the functions found in the PIL.ImageOps submodule.
Resampling

For raster data, the equivalent to vector cleaning is the removal of unnecessary detail and file size reduction, which can be done by resampling the size and frequency of the grid cells. Such resampling involves algorithms for smoothing out and redistributing the old cell values to the new cell structure. A lot of the same principles apply to resizing an image. Fortunately for us, our raster data values are stored in a PIL Image class, so we simply use its resize method with the nearest neighbor algorithm, which asks for a size in terms of pixels (or number of grid cells in our case). For the convenience of the user, we also give them the alternative to, instead specify the desired geographic width and height of each cell (for instance, degrees or meters, depending on the data's coordinate reference system), with our program calculating the necessary grid resolution behind the scenes. If specified, remember that the $y$ axis of geographic coordinates tend to run in the opposite direction to that of raster coordinates, so cellheight must be given as a negative number. If the user is curious about the grid dimensions or cell size of their existing raster, remember that this can be found in the layer properties window we created earlier this chapter.

An alternative library for raster resampling here will be PyResample. I chose not to use it in our lightweight application due to its NumPy and SciPy dependence.

Other useful libraries for raster management functionality that you may wish to explore are GDAL as mentioned earlier or Rasterio which depends on GDAL.

Take a look at the following code:

```python
def resample(raster, width=None, height=None, cellwidth=None, cellheight=None):
    raster = raster.copy()

    if width and height:
        # calculate new cell dimensions based on the new raster size
        widthfactor = raster.width / float(width)
        heightfactor = raster.height / float(height)
        oldcellwidth, oldcellheight = raster.info["cellwidth"],
        newcellwidth, newcellheight = oldcellwidth * widthfactor,
        oldcellheight * heightfactor
```
Chapter 5

# resample each grid
for band in raster:
    band.img = band.img.resize((width, height), PIL.Image.NEAREST)
    # update cells access
    band.cells = band.img.load()

    # remember new celldimensions
    raster.info["cellwidth"] = newcellwidth
    raster.info["cellheight"] = newcellheight
    return raster

elif cellwidth and cellheight:
    # calculate new raster size based on the new cell dimensions
    widthfactor = raster.info["cellwidth"] / float(cellwidth)
    heightfactor = raster.info["cellheight"] / float(cellheight)
    oldwidth, oldheight = raster.width, raster.height
    newwidth, newheight = int(round(oldwidth * widthfactor)),
                          int(round(oldheight * heightfactor))

    # resample each grid
    for band in raster:
        band.img = band.img.resize((newwidth, newheight), PIL.Image.NEAREST)
        # update cells access
        band.cells = band.img.load()

        # remember new celldimensions
        raster.info["cellwidth"] = cellwidth
        raster.info["cellheight"] = cellheight
        return raster

else:
    raise Exception("To rescale raster, either width and height or cellwidth and cellheight must be specified.")
Weaving functionality into the user interface

Now, we get to the part where we can make the management functionality created earlier accessible to the user in the visual user interface.

Layer-specific right-click functions

Some of the functionality we created in this chapter is intrinsically bound to only one layer, so it makes sense to make these available directly by right-clicking on the desired layer to operate on. Such a feature is only specific to the application we are currently making, so let's define this right-click menu in the app/dialogues.py module. Since Tkinter already has such a nicely formatted popup menu widget, with easy methods for adding items and commands, all we need to do is subclass it. Vector and raster layers will each get their own menu, but both will have in common items called as Rename, Save as, and Properties. To give them better visuals, find the three .png images with the same names as each of these items so we can assign them to the menu items, saving them inside the app/icons folder.

First, we make the options menu for vector layers. We give it the split and clean functions we created earlier, and assign them icons, which you must find and save as app/icons/split.png and app/icons/clean.png. Refer to the following screenshot:

```python
class RightClickMenu_VectorLayer(tk.Menu):
    def __init__(self, master, layerspane, layeritem, statusbar, **kwargs):
        # Make this class a subclass of tk.Menu and add to it
        tk.Menu.__init__(self, master, tearoff=0, **kwargs)
        self.layerspane = layerspane
        self.layeritem = layeritem
```

[116]
self.statusbar = statusbar
self.imgs = dict()

# Renaming
self.imgs["rename"] = icons.get("rename.png", width=32, height=32)
self.add_command(label="Rename", command=self.layeritem.ask_rename, image=self.imgs["rename"], compound="left")

# Saving
def ask_save():
    savepath = asksaveasfilename()
    self.statusbar.task.start("Saving layer to file...")
    pending = dispatch.request_results(self.layeritem.renderlayer.data.save, args=[savepath])
    def finish(result):
        if isinstance(result, Exception):
            popup_message(self, str(result) + "\n\n" + savepath)
        self.statusbar.task.stop()
    dispatch.after_completion(self, pending, finish)
    self.imgs["save"] = icons.get("save.png", width=32, height=32)
    self.add_command(label="Save as", command=ask_save, image=self.imgs["save"], compound="left")

# -(Breakline)-
self.add_separator()

# Splitting
def open_options_window():
    window = VectorSplitOptionWindow(self.layeritem, self.layerspane, self.layeritem, statusbar)
    self.imgs["split"] = icons.get("split.png", width=32, height=32)
    self.add_command(label="Split to layers", command=open_options_window, image=self.imgs["split"], compound="left")

# -(Breakline)-
self.add_separator()

# Cleaning
Managing and Organizing Geographic Data

```python
def open_options_window():
    window = VectorCleanOptionWindow(self.layeritem,
        self.layerspane, self.layeritem, statusbar)
    self.imgs["clean"] = icons.get("clean.png", width=32, height=32)
    self.add_command(label="Clean Geometries",
        command=open_options_window, image=self.imgs["clean"],
        compound="left")

# ---(Breakline)---
    self.add_separator()

# View properties
def view_properties():
    window = VectorLayerOptionsWindow(self.layeritem,
        self.layeritem, statusbar)
    self.imgs["properties"] = icons.get("properties.png",
        width=32, height=32)
    self.add_command(label="Properties",
        command=view_properties, image=self.imgs["properties"],
        compound="left")
```

We then move onto the options menu for rasters layers. The only layer-specific function here is `resample()`, so find and save an icon for it as app/icons/resample.png. You can see an icon named **Resample** in the following screenshot:

![Resample Icon](app/icons/resample.png)

Refer to the following code:

```python
class RightClickMenu_RasterLayer(tk.Menu):
    def __init__(self, master, layerspane, layeritem, statusbar,
        **kwargs):
        # Make this class a subclass of tk.Menu and add to it
        tk.Menu.__init__(self, master, tearoff=0, **kwargs)
        self.layerspane = layerspane
```
self.layeritem = layeritem
self.statusbar = statusbar
self.imgs = dict()

# Renaming
self.imgs["rename"] = icons.get("rename.png", width=32, height=32)
self.add_command(label="Rename", command=self.layeritem.ask_rename, image=self.imgs["rename"], compound="left")

# Saving
def ask_save():
    savepath = asksaveasfilename()
    self.statusbar.task.start("Saving layer to file...")
    pending = dispatch.request_results(self.layeritem.renderlayer.data.save, args=[savepath])
    def finish(result):
        if isinstance(result, Exception):
            popup_message(self, str(result) + "\n\n" + savepath)
        self.statusbar.task.stop()
    dispatch.after_completion(self, pending, finish)
    self.imgs["save"] = icons.get("save.png", width=32, height=32)
    self.add_command(label="Save as", command=ask_save, image=self.imgs["save"], compound="left")

# ---(Breakline)---
self.add_separator()

# Resampling
def open_options_window():
    window = RasterResampleOptionWindow(self.layeritem, self.layerspane, self.layeritem, statusbar)
    self.imgs["resample"] = icons.get("resample.png", width=32, height=32)
    self.add_command(label="Resample", command=open_options_window, image=self.imgs["resample"], compound="left")

# ---(Breakline)---
self.add_separator()
# View properties

def view_properties():
    window = RasterLayerOptionsWindow(self.layeritem,
           self.layeritem, statusbar)
    self.imgs["properties"] = icons.get("properties.png",
           width=32, height=32)
    self.add_command(label="Properties",
           command=view_properties, image=self.imgs["properties"],
           compound="left")

**Defining the tool options windows**

In the preceding code, clicking on an item in the menu opens an options window for a specific tool. We will now create these options windows in app/dialogues.py, utilizing our helpful `RunToolFrame` template to populate the window with appropriate options and widgets. Since these are layer-specific tools, we also remember to set the layer data as a hidden argument. Finally, the results from the processes are added to our LayersPane. The following screenshot shows the options window for vector cleaning:
Here is the code to implement the mentioned functionality:

class VectorCleanOptionWindow(Window):
    def __init__(self, master, layerspane, layeritem, statusbar, **kwargs):
        # Make this class a subclass and add to it
        Window.__init__(self, master, **kwargs)

        # Create runtoolframe
        self.runtool = RunToolFrame(self)
        self.runtool.pack(fill="both", expand=True)
        self.runtool.assign_statusbar(statusbar)

        # Add a hidden option from its associated layeritem data
        self.runtool.add_hidden_option(argname="data",
                                        value=layeritem.renderlayer.data)

        # Set the remaining options
        self.runtool.set_target_method("Cleaning data...",
                                       vector.manager.clean)
        self.runtool.add_option_input(argname="tolerance",
                                        label="Tolerance (in distance units)",
                                        valuetype=float, default=0.0,
                                        minval=0.0, maxval=1.0)

        # Define how to process
        newname = layeritem.namelabel["text"] + "_cleaned"
        def process(result):
            if isinstance(result, Exception):
                popup_message(self, "Failed to clean the data:
                  
                  " + str(result) )
            else:
                layerspane.add_layer(result, name=newname)
                self.destroy()

        self.runtool.set_finished_method(process)
The following screenshot demonstrates the options window for vector splitting populated with a list of field to choose from:

Here is the code to implement the mentioned functionality:

```python
class VectorSplitOptionWindow(Window):
    def __init__(self, master, layerspane, layeritem, statusbar, **kwargs):
        # Make this class a subclass and add to it
        Window.__init__(self, master, **kwargs)

        # Create runtoolframe
        self.runtool = RunToolFrame(self)
        self.runtool.pack(fill="both", expand=True)
        self.runtool.assign_statusbar(statusbar)

        # Add a hidden option from its associated layeritem data
        self.runtool.add_hidden_option(argname="data", value=layeritem.renderlayer.data)

        # Set the remaining options
        self.runtool.set_target_method("Splitting data...",
                                    vector.manager.split)
```

---

[122]
self.runtool.add_option_input(argname="splitfields",
    label="Split by fields",
    multi=True,
    choices=layeritem.renderlayer.data.fields,
    valuetype=str)

    # Define how to process
    def process(result):
        if isinstance(result, Exception):
            popup_message(self, "Failed to split the data:

    
    \n    \n    + str(result) )
        else:
            for splitdata in result:
                layerspane.add_layer(splitdata)
                self.update()
                self.destroy()
                self.runtool.set_finished_method(process)

As shown in the following screenshot of the raster resampling window a user can manually enter input for height and width of raster and cell data:
Here is the code for same:

class RasterResampleOptionWindow(Window):
    def __init__(self, master, layerspane, layeritem, statusbar, **kwargs):
        # Make this class a subclass and add to it
        Window.__init__(self, master, **kwargs)

        # Create runtoolframe
        self.runtool = RunToolFrame(self)
        self.runtool.pack(fill="both", expand=True)
        self.runtool.assign_statusbar(statusbar)

        # Add a hidden option from its associated layeritem data
        self.runtool.add_hidden_option(argname="raster",
                                        value=layeritem.renderlayer.data)

        # Set the remaining options
        self.runtool.set_target_method("Resampling data...",
                                        raster.manager.resample)

    def get_data_from_layername(name):
        data = None
        for layeritem in layerspane:
            if layeritem.name_label["text"] == name:
                data = layeritem.renderlayer.data
                break
        return data

        self.runtool.add_option_input(argname="width",
                                        label="Raster width (in cells)",
                                        valuetype=int)

        self.runtool.add_option_input(argname="height",
                                        label="Raster height (in cells)",
                                        valuetype=int)

        self.runtool.add_option_input(argname="cellwidth",
                                        label="Cell width (in distance units)",
                                        valuetype=float)

        self.runtool.add_option_input(argname="cellheight",
                                        label="Cell height (in distance units)",
                                        valuetype=float)

        # Define how to process after finished
        def process(result):
            if isinstance(result, Exception):
                popup_message(self, "Failed to resample the data:

+ "\n\n" + str(result) )
            else:

[124]
layerspane.add_layer(result)
self.destroy()
self.runtool.set_finished_method(process)

Finally, we need to instruct our application that right-clicking on a layer should open the appropriate menu. We define this in the initialization phase of the class defining our GUI in our app/builder.py module, after creating the LayersPane:

```python
# Bind layeritem right click behavior
def layer_rightclick(event):
    layeritem = event.widget.master.master
    if isinstance(layeritem.renderlayer, pg.renderer.VectorLayer):
        menu = RightClickMenu_VectorLayer(self, self.layerspane, layeritem, self.statusbar)
    elif isinstance(layeritem.renderlayer, pg.renderer.RasterLayer):
        menu = RightClickMenu_RasterLayer(self, self.layerspane, layeritem, self.statusbar)
    # Place and show menu
    menu.post(event.x_root, event.y_root)
    self.layerspane.bind_layer_rightclick(layer_rightclick)
```

### Setting up the management tab

In contrast to the right-click menu of individual layers, the top ribbon tabs should be reserved for more general functionalities that takes multiple layers as input.

All of our data management-related functionality goes in a separate tab called Manage, to which we attach a vector and a raster toolbar, each populated with one or more buttons that open an options window for running related functionalities. Therefore, we add the following to app/builder.py in our GUI class after having created the ribbon and the visualize tab, as shown in the following screenshot:

![Management Tab](image)
Here is the code to set up the **Manage** tab:

```python
## Management tab
managetab = self.ribbon.add_tab("Manage")
### (Vector toolbar)
vectorfiles = managetab.add_toolbar("Vector Files")
def open_merge_window():
    window = VectorMergeOptionWindow(self, self.layerspane, self.statusbar)
    vectorfiles.add_button(text="Merge",
                            icon="vector_merge.png",
                            command=open_merge_window)
### (Raster toolbar)
rasterfiles = managetab.add_toolbar("Raster Files")
def open_mosaic_window():
    window = RasterMosaicOptionWindow(self, self.layerspane, self.statusbar)
    rasterfiles.add_button(text="Mosaic", icon="mosaic.png",
                            command=open_mosaic_window)
```

**Defining the tool options windows**

We define the various tool-specific options windows in `app/dialogues.py`, as we did earlier in text. First for the vector merge tool window, as seen in the following screenshot:

<table>
<thead>
<tr>
<th>User Input</th>
<th>Python GIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layers to be merged</td>
<td>country_triangles.shp</td>
</tr>
<tr>
<td>Merges two or more vector data layers, combining all their rows and fields into a single table.</td>
<td></td>
</tr>
<tr>
<td>Adds the merged data to the layers list.</td>
<td></td>
</tr>
</tbody>
</table>
Here is the code for same:

class VectorMergeOptionWindow(Window):
    def __init__(self, master, layerspane, statusbar, **kwargs):
        # Make this class a subclass and add to it
        Window.__init__(self, master, **kwargs)

        # Create runtool frame
        self.runtool = RunToolFrame(self)
        self.runtool.pack(fill="both", expand=True)
        self.runtool.assign_statusbar(statusbar)

        # Set the remaining options
        self.runtool.set_target_method("Merging data...",
        vector.manager.merge)
        def get_data_from_layername(name):
            data = None
            for layeritem in layerspane:
                if layeritem.namelabel["text"] == name:
                    data = layeritem.renderlayer.data
                    break
            return data
        self.runtool.add_option_input(argname=None,
            label="Layers to be merged",
            multi=True,
            choices=[layeritem.namelabel["text"]
                    for layeritem in layerspane],
            valuetype=get_data_from_layername)

        # Define how to process
        def process(result):
            if isinstance(result, Exception):
                popup_message(self, "Failed to merge the data:
                " + 
                "\n\n" + str(result) )
The options window for the raster mosaicking tool looks as follows:

Here is the code:

```python
class RasterMosaicOptionWindow(Window):
    def __init__(self, master, layerspane, statusbar, **kwargs):
        # Make this class a subclass and add to it
        Window.__init__(self, master, **kwargs)

        # Create runtoolframe
        self.runtool = RunToolFrame(self)
        self.runtool.pack(fill="both", expand=True)
        self.runtool.assign_statusbar(statusbar)

        # Set the remaining options
        self.runtool.set_target_method("Mosaicking data...",
                                     raster.manager.mosaic)

    def get_data_from_layername(name):
        data = None
        for layeritem in layerspane:
            # Process layer item
```

---

[ 128 ]
if layeritem.namelabel["text"] == name:
    data = layeritem.renderlayer.data
    break
return data

self.runtool.add_option_input(argname=None,
    label="Layers to be mosaicked",
    multi=True,
    choices=[layeritem.namelabel["text"]
        for layeritem in layerspane],
    valuetype=get_data_from_layername)

# Define how to process
def process(result):
    if isinstance(result, Exception):
        popup_message(self, "Failed to mosaick the data:
        + \n        + str(result))
    else:
        layerspane.add_layer(result, name="mosaicked")
    self.runtool.set_finished_method(process)

Summary

In this chapter, we created functionalities related to managing and organizing files. This included a window to inspect the basic properties of any data layer. As for operations, we implemented splitting, merging, and geometry cleaning for vector data, and mosaicking and resampling for raster data. These functions were then made available in the application GUI, some by choosing from a pop-up menu when right-clicking on a layer, others by clicking on an icon button in the management tab up by the top ribbon. Each tool got its own window dialogue class with editable options.

By going through this chapter, you should now know the general steps involved in adding a geospatial functionality, making it accessible in the GUI, and adding the results as a new layer if desired. As we move on to the next chapter, where we will build some basic analysis functionality, all we need to do is repeat and follow the same steps and procedures.
Where to buy this book

You can buy Python Geospatial Development Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.