PROG5000 Assignment 1 Documentation

Reuben Walker

Purpose & Datasets Used

The purpose of this Python script is to provide summarized data about all forest stands of a given leading species. The user is able to select what leading species they want to see a summary of. The forest stand data is from the Nova Scotia Department of Natural Resources (NSDNR). This data is represented on the map by multiple polygons, with each polygon corresponding to a forest stand. Each polygon has data associated with it to describe the forest stand. This includes the perimeter and area, height, crown coverage, and info about the species composition of the stand. This program will be mostly working with the leading species and the area of the stand. The leading species (listed in the attribute table as SP1) refers to the tree species that is most dominant in the stand, and is represented by a one or two letter species code. There is also the variable "SP1P" which refers to the percentage of that stand that is composed of the leading species.

Summary of Program

The script will perform the following actions:

1. Build a list consisting of each unique species represented in the dataset.
2. Prompt the user to select a species that they want to see information for. If the user enters an invalid input, the script will handle it and prompt the user to enter a valid input without crashing the program.
3. Based on the user's choice, calculate the following:
    a. Number of polygons (stands) of selected species
    b. Smallest area stand of selected species
    c. Largest area stand of selected species
    d. Average leading species percentage within stands of selected species
    e. Average area of stands of selected species
    f. Total area of stand of selected species
4. Display the calculated data.
5. Prompt the user to run the program again.

1- Build a List of Unique Species

We want to provide the user with a list of species to choose from. We know there are 9 species, so we could hard-code them in. However, this would make the program very inflexible and only useful for this specific snipped of this dataset. Instead, the program finds out how many unique species there are in the dataset and builds a list out of them:

```
7  # Main function definition (makes looping easier)
8  def main():
9      # Select all forest polygons in the layer and create an empty list of species
10     layer = iface.activeLayer()
11     layer.selectByExpression('"SP1" IS NOT NULL', QgsVectorLayer.SetSelection)
12     selection = layer.selectedFeatures()
13     speciesList = []
14
15     # Go through each entry and add unique species to the list
16     for feature in selection:
17         if feature['SP1'] not in speciesList:
18             speciesList.append(feature['SP1'])
19
20     # Create a list of numeric options for user to choose from
21     optionList = ''
22     for i in range(len(speciesList)):
23         optionList = optionList + ("%d. %s\n" % (i+1,speciesList[i]))
24
```

This part of the code selects the active layer in the interface, which we assume to be the forest layer. Then, it will select all polygons that do not have a NULL value for leading species (SP1). We pass our selection to a list called *selection*. We create an empty list called *speciesList* to add all the unique species values to.

The for-loop starting on line 16 goes through every feature in the selection. For each feature, it checks if the value for SP1 is contained in *speciesList.* If not, it will append that value to *speciesList*. If *speciesList* already contains that value, it will not be added. This results in a list that is populated with one instance of each unique species code.

Starting on line 21 we make an empty string called *optionList* and loop through *speciesList*, adding to the string an increasing series of numbers and each species code we gathered in *speciesList*. This results in a string that is properly formatted for the QInputDialog function that we will use next.

2- Prompt the User for Input

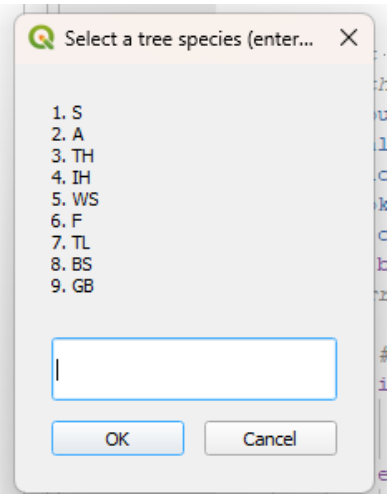
```
25     # Set up QInputDialog
26     qid = QInputDialog()
27     title = "Select a tree species (enter the number):"
28     label = optionList
29     mode = QLineEdit.Normal
30     default = ""
31
```

This part of the code sets up the format for the input dialog box. We have passed *optionList* to the label for this input dialog, so the input dialog should display the list that we built.

The following code is used to prompt the user for input and handle any invalid or erroneous inputs:

```
32      # Prompt user for input, returning an error and re-prompting the user
33      #   if they enter a non-numeric input or out of range value
34      validInput = False
35      while validInput == False:
36          optionSelect, ok = QInputDialog.getText(qid,title,label,mode,default)
37          if ok == False:
38              optionSelect = '1'
39              break
40          # Error handler if user enters a non-numeric input
41          try:
42              # Check if user input is in range
43              if int(optionSelect) <= 0 or int(optionSelect) > len(speciesList):
44                  iface.messageBar().pushMessage("Error", "You entered an invalid
45                  validInput = False
46              else:
47                  validInput = True
48                  iface.messageBar().clearWidgets()
49          except:
50              iface.messageBar().pushMessage("Error", "You entered an invalid inpu
51              validInput = False
52
```

The variable *validInput* is used to keep the while-loop going until an acceptable input is given. At the beginning of the while-loop, the QInputDialog.getText function displays the following dialog box to prompt the user for an input:



If the Cancel button is pressed, then the variable *ok* is set to False. We assume the user does not want to go through with making a selection – we use a trick here to allow for this. On line 38 we just set the user's chosen option to "1" (Note: this is a string, not an integer) and then break out of the loop. This forced "1" selection was the easiest solution I could find to allow the user to cancel out of the program without causing an error.

Starting on line 41, we use a Try block. When writing this program, I found that the QInputDialog function seemed to accept 0 and numbers from -1 to -8 without any problems – I did not want this behavior in my program. So, I cast *optionSelect* (which is a string) to an integer and then checked if it's either below zero or

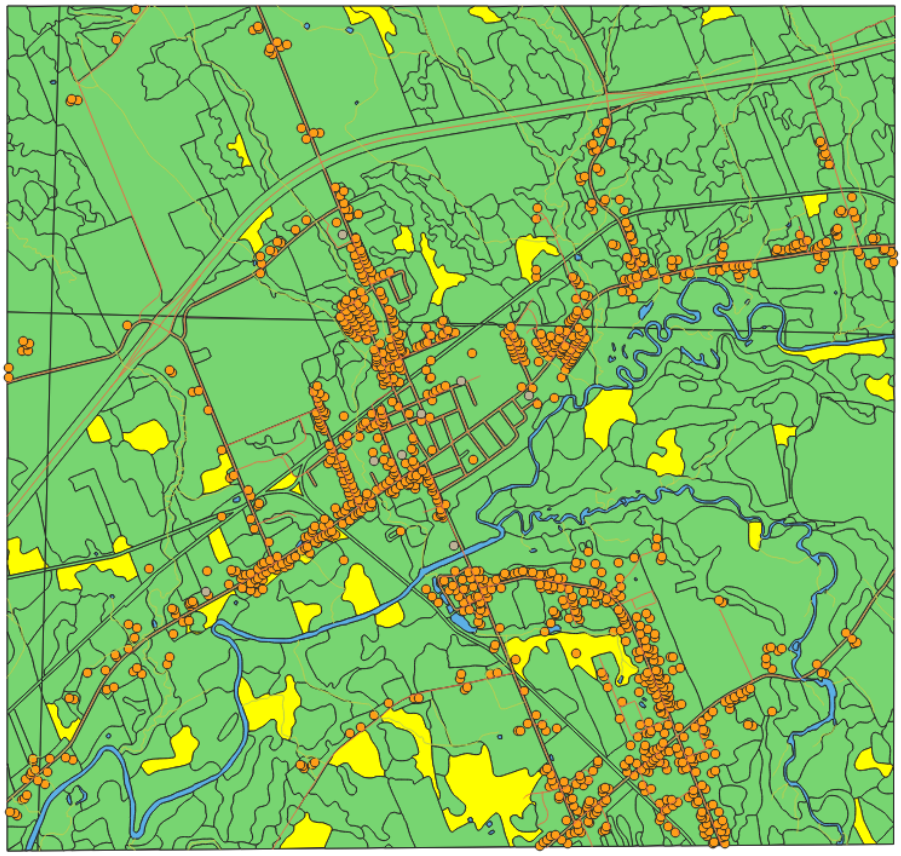
higher than the length of the species list. If it is, then the input is considered invalid and the program will prompt the user to try again If the user entered invalid characters (such as letters or special characters), then casting *optionSelect* to an integer will fail and the code will move to the Except block on line 49, which will also prompt the user to try again. If the input is valid, that is, an integer between 1 and number of unique species (9 in this case), then the *validInput* will be set to True and the program will break out of the loop.

3- Calculate Forest Stand Statistics

Now that a valid selection has been made, the program knows which polygons to select. First, we will build an expression that will be accepted by the selectByExpression function. To do this we will cast *optionSelect* to an integer and subtract 1 from it, since Python list indices start at 0 and the list we displayed to the user started at 1. Then we use this value to get the appropriate species code for SP1 from *speciesList.* This code is inserted into the expression using string formatting and assigned to a newly created string called *expr*.

```
53      # Select appropriate forest stand polygons based on user input
54      expr = "\"SP1\" = '%s'" % (speciesList[int(optionSelect)-1])
55      layer.selectByExpression(expr, QgsVectorLayer.SetSelection)
56      selection = layer.selectedFeatures()
57
```

The string *expr* is passed to the layer.selectByExpression function, which selects the forest stands we want. The map display will highlight the selected polygons. This is what it looks like if I select option #2, “A”:



Now, we can begin calculating statistics. As a reminder, we are looking for:

1. Number of polygons (stands) of selected species
2. Smallest area stand of selected species
3. Largest area stand of selected species
4. Average leading species percentage within stands of selected species
5. Average area of stands of selected species
6. Total area of stand of selected species

All of these stats can be derived from just these 2 items:

- A list containing all the areas (SHAPE_Area) of each polygon for the selected species
- The sum of the all leading species percentages (SP1P) for the selected species

```
58      # Create a list containing all areas of the selected polygons
59      # as well as the average leading species percentage for polygons
60      areas = []
61      leadSpeciesPercentage = 0
62      for feature in selection:
63          areas.append(feature['SHAPE_Area'])
64          leadSpeciesPercentage += feature['SP1P']
65      leadSpeciesPercentage = leadSpeciesPercentage/float(len(areas))
66
```

Empty list *areas* will contain all the areas and *leadSpeciesPercentage* will contain the sum of the percentages.

The for-loop will step through each of the selected features and append each of the values for the polygon areas to list *areas*. It will also add each leading species percentage to our running total of percentages.

After the loop has been run, we can find the average leading species percentage by dividing the total by the length of the list of areas. The length of the list has been cast to a floating-point number to prevent errors, and because the result will also be a floating-point number.

4- Display the Calculated Data

To display the calculated data, I used formatted print statements. I did most of the calculations within the print statements to avoid making a lot of new variables. If the program was expanded upon, I might want to have given each result its own variable instead.

```
67      # Print final results
68      if ok == True:
69          print("\n========== RESULTS FOR SELECTED SPECIES: %2s ==========" % (speciesList[int(optionSelect)-1]))
70          print("Number of polygons of this species: %16i" % len(areas))
71          print("Smallest area stand of this species: %18.2f" % min(areas)+" m\u00B2")
72          print("Largest area stand of this species: %19.2f" % max(areas)+" m\u00B2")
73          print("Avg. leading species percentage within stands: %8.2f" % (leadSpeciesPercentage) + '%')
74          print("Avg. area of stands of this species: %18.2f" % (sum(areas)/(float(len(areas))))+" m\u00B2")
75          print("Total area of stands of this species: %17.2f" % sum(areas) +" m\u00B2")
76
```

The number of polygons found will be the length of the *areas* list we created. The functions min() and max() can easily find the smallest and largest values in that list. *leadSpeciesPercentage* was calculated in the previous step, so it's displayed as-is. The average area is derived by using sum() on the *areas* list, and then dividing it by the length of the list of *areas*. The total area is found by just using sum() on the *areas* list.

Note that this entire block of code does not run if *ok* is equal to False. This is a trick I used to allow the user to cancel out of making a selection – if the user pressed "cancel" when prompted for the selection, the if-statement on line 68 skips over the entire block of print statements, making it appear as if no selection was made.

The output in the terminal looks like this:

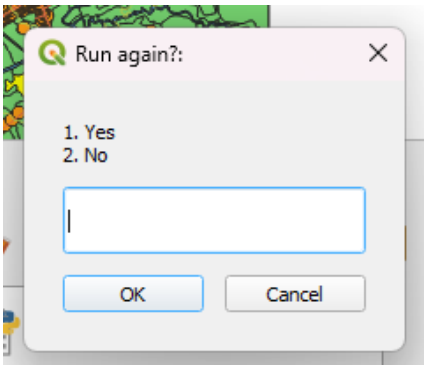
```
14 ========== RESULTS FOR SELECTED SPECIES:  A ==========
15 Number of polygons of this species:                42
16 Smallest area stand of this species:            617.38 m²
17 Largest area stand of this species:          210845.18 m²
18 Avg. leading species percentage within stands:    49.52 %
19 Avg. area of stands of this species:          36108.04 m²
20 Total area of stands of this species:        1516537.87 m²
```

5- Prompt the User to Run the Program Again

Note that all of the code from the previous sections was contained within a function titled *main().* This is the code that lets the user run the program over and over again if they want. As long as *userContinue* is equal to '1', the program will run the main function, then prompt the user with another QInputDialog box if they want to run the program again.

```
77  # Loop to run the main function as many times as the user wants.
78  #   User will be asked if they want to continue. If they enter
79  #   an invalid value, the program will also end
80  userContinue = '1'
81  while userContinue == '1':
82      main()
83      qid = QInputDialog()
84      title = "Run again?: "
85      label = '1. Yes\n2. No'
86      mode = QLineEdit.Normal
87      default = ""
88      userContinue, ok = QInputDialog.getText(qid,title,label,mode,default)
89      if userContinue.lower() == 'y' or userContinue.lower() == 'yes':
90          userContinue = '1'
91
```

The dialog box looks like this:



I programmed a bit of flexibility in how the user can respond – if they enter “1”, “y”, or “yes” (“y” and “yes” are not case sensitive because I used the lower() function), the program will run again. If they enter anything else or press “cancel” then the program will end. I figured it might make the user frustrated if they want to end the program but are told to re-enter their answer for a simple yes-or-no question if they enter an invalid input. Also note that for *userContinue* I am using a string, when a Boolean value would be more appropriate. This is because QInputDialog returns a string value so I figured it would make more sense to check for the appropriate string instead of writing a method to convert the string into a bool.

Limitations of Script

The script has the following limitations:

1. It only works if the “Forest” layer is the active layer in QGIS. In pervious versions of my program, if I ran the code without making the “Forest” layer active first, it would get stuck in an infinite loop. Fortunately, now it just returns an error and terminates.
2. The script only works on Forestry data that is formatted in a certain way (for example, with columns SP1, SP1P, SHAPE_Area, etc)
3. The interface for selecting a species is clunky and unintuitive, making the user enter a number displayed on a list. A drop-down menu or list of buttons would be more user-friendly. Same for the prompt asking the user if they want to re-run the code.
4. The list of species that the program displays only shows the species code, which can be cryptic to users. At first glance, nobody would know that species codes such as “TH” “WS” and “A” are supposed to represent. However, the dataset does not contain any full-length species names in it. I would need to either perform a table join/relate or hard-code in the species names (not desirable).