EGM722 – Programming for GIS and Remote Sensing

Pull requests and merging branches with GitHub

Introduction

This document briefly discusses what to do when you need to incorporate fixes or updates into your code, either because you have developed a new feature that is ready for wider use, or because you have discovered and fixed an error.

You may have noticed that there were two (entirely unintentional) errors in the Week 1 practical. These have been fixed and updated, but illustrating the process of fixing errors and incorporating those fixes into the **main** branch of the repository is still a useful exercise.

The Errors

The two errors (or bugs) were as follows in section 10 of the practical notebook. The first error had to do with the order of the arguments to the **Point** constructor:

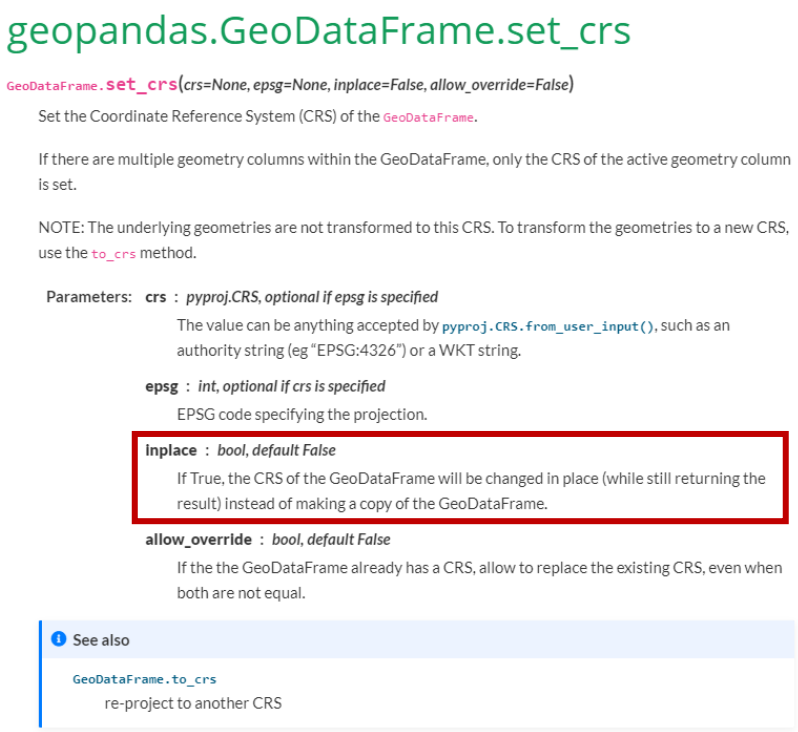
```
In [ ]: df['geometry'] = list(zip(df['lat'], df['lon'])) # zip is an iterator, so we use list to create
                                                        # something that pandas can use.
        df['geometry'] = df['geometry'].apply(Point) # using the 'apply' method of the dataframe,
                                                    # turn the coordinates column
                                                    # into points (instead of a tuple of lat, lon coordinates).
                                                    # NB: Point takes (x, y) coordinates
```

Notice that while the helpful note at the bottom states that Point takes (x, y) coordinates, the first line of the cell has the order as Latitude (y), Longitude (x) – the opposite of what it should be. Hence, when you save the GeoDataFrame to a shapefile, the coordinate order will be reversed, and it will not display properly.

The second error is a bit harder to spot, but it comes a bit further down in the cell:

```
In [ ]: gdf = gpd.GeoDataFrame(df)
        gdf.set_crs("EPSG:4326") # this sets the coordinate reference system to epsg:4326, wgs84 lat/lon
```

Here, you see the line to set the CRS of the GeoDataFrame to EPSG:4326 (WGS84 Latitude/Longitude). But, if we check the Application Programming Interface (API) reference for GeoDataFrame.set_crs, we see the following:



geopandas.GeoDataFrame.set_crs

GeoDataFrame.set_crs(crs=None, epsg=None, inplace=False, allow_override=False)

Set the Coordinate Reference System (CRS) of the GeoDataFrame.

If there are multiple geometry columns within the GeoDataFrame, only the CRS of the active geometry column is set.

NOTE: The underlying geometries are not transformed to this CRS. To transform the geometries to a new CRS, use the to_crs method.

Parameters: **crs** : *pyproj.CRS, optional if epsg is specified*
The value can be anything accepted by pyproj.CRS.from_user_input(), such as an authority string (eg "EPSG:4326") or a WKT string.

epsg : *int, optional if crs is specified*
EPSG code specifying the projection.

inplace : *bool, default False*
If True, the CRS of the GeoDataFrame will be changed in place (while still returning the result) instead of making a copy of the GeoDataFrame.

allow_override : *bool, default False*
If the the GeoDataFrame already has a CRS, allow to replace the existing CRS, even when both are not equal.

See also

GeoDataFrame.to_crs
re-project to another CRS

This shows that set_crs does not automatically change the CRS in place; rather, it returns a copy of the GeoDataFrame that has the CRS set to whatever is passed to set_crs. Because this was done without passing **inplace=True** or assigning the copy of the GeoDataFrame to a variable, the changes weren't saved – and hence, when the GeoDataFrame was saved to the disk, it had no CRS information, and would not load properly in QGIS or ArcGIS Pro.

For now, these error in the Week 1 practical have been fixed – if you go to the course repository on GitHub, you will see that the Week1 notebook has these lines changed to be correct. The next sections of this document describe the process of updating the repository to fix the errors.

Creating a fix branch

To fix the error, I first created a new branch, **week1_fix**, to work on the changes. For this specific case, this is probably overkill, but it's good practice nonetheless to avoid committing changes directly to the **main** branch. From the command line this looks like:

```
> git checkout -b week1_fix
```

The command git checkout will allow you to switch branches; the **-b** flag tells git to create a new branch with the name that you are checking out.

Next, I opened the jupyter-notebook and made the fixes:

```
In [ ]: df['geometry'] = list(zip(df['lon'], df['lat'])) # zip is an iterator, so we use list to create
                                                        # something that pandas can use.
        df['geometry'] = df['geometry'].apply(Point) # using the 'apply' method of the dataframe,
                                                     # turn the coordinates column
                                                     # into points (instead of a tuple of lat, lon coordinates).
                                                     # NB: Point takes (x, y) coordinates
```

```
In [ ]: gdf = gpd.GeoDataFrame(df)
        gdf.set_crs("EPSG:4326", inplace=True) # this sets the coordinate reference system to epsg:4326, wgs84 lat/lon
```

After that, I ran the following command to **stage** the changes:

```
> git add Week1\Practical1.ipynb
```

This tells git to get ready to save the changes made to Week1\Practial1.ipynb. Next, I used the git commit command to actually save the changes:

```
> git commit -m "fix geometry by switching order to lon/lat, add inplace=True to set CRS in place."
```

The **-m** flag allows you to set a message from the command line – if you don't include this, git will open the text editor so that you can type out a message (this is especially useful for longer commit messages).

The last thing to do is to publish these changes to the GitHub repository – so far, all of the steps (creating a fix branch, changing and committing the changes) have been done on the local repository only. To publish the changes (update the GitHub repository), we use **git push**:

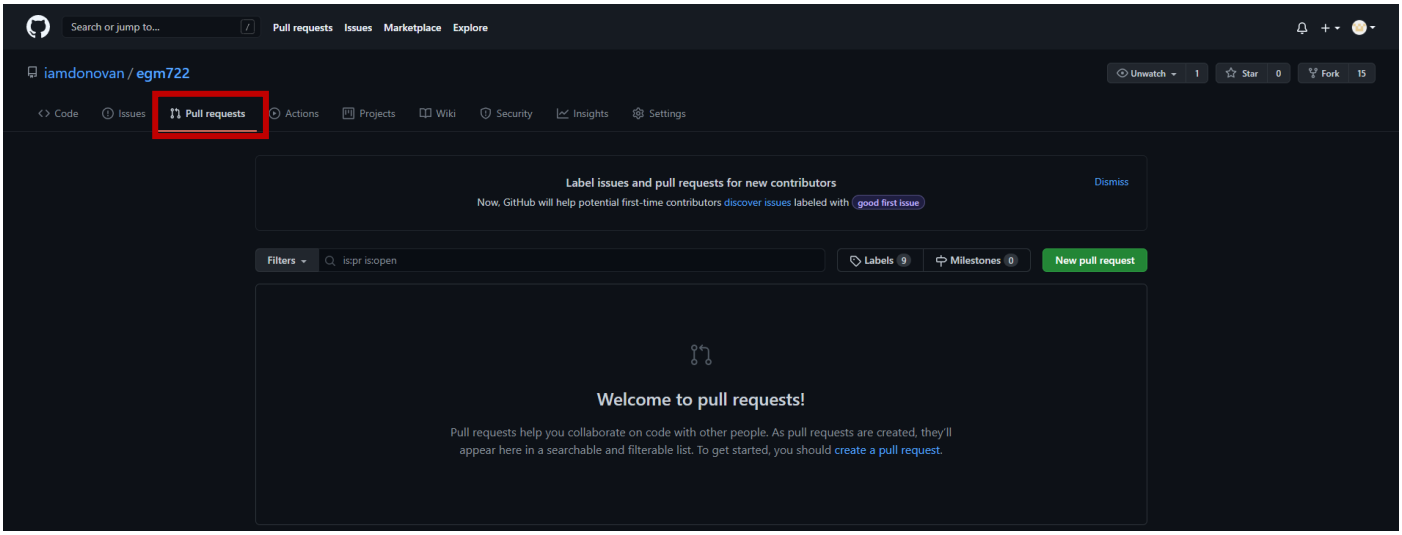
```
> git push -u origin week1_fix
```

This tells git to **push** week1_fix to **origin**, which is an alias for the GitHub repository URL (https://github.com/iamdonovan/egm722). Because this branch doesn't yet have any information about the upstream branch, we use the **-u** flag to tell git where to push the changes, and to save this information for the future.

Submitting a Pull Request

On GitHub, we should now see that there are 3 branches for the EGM722 repository, but the **main** branch has not yet been updated. To do this, we can open a **Pull Request**, which provides a chance for others to review changes before merging branches – when collaborating on projects with others, this is usually the best way to develop new features/fixes, as multiple sets of eyes on a problem are often better than one person working alone.

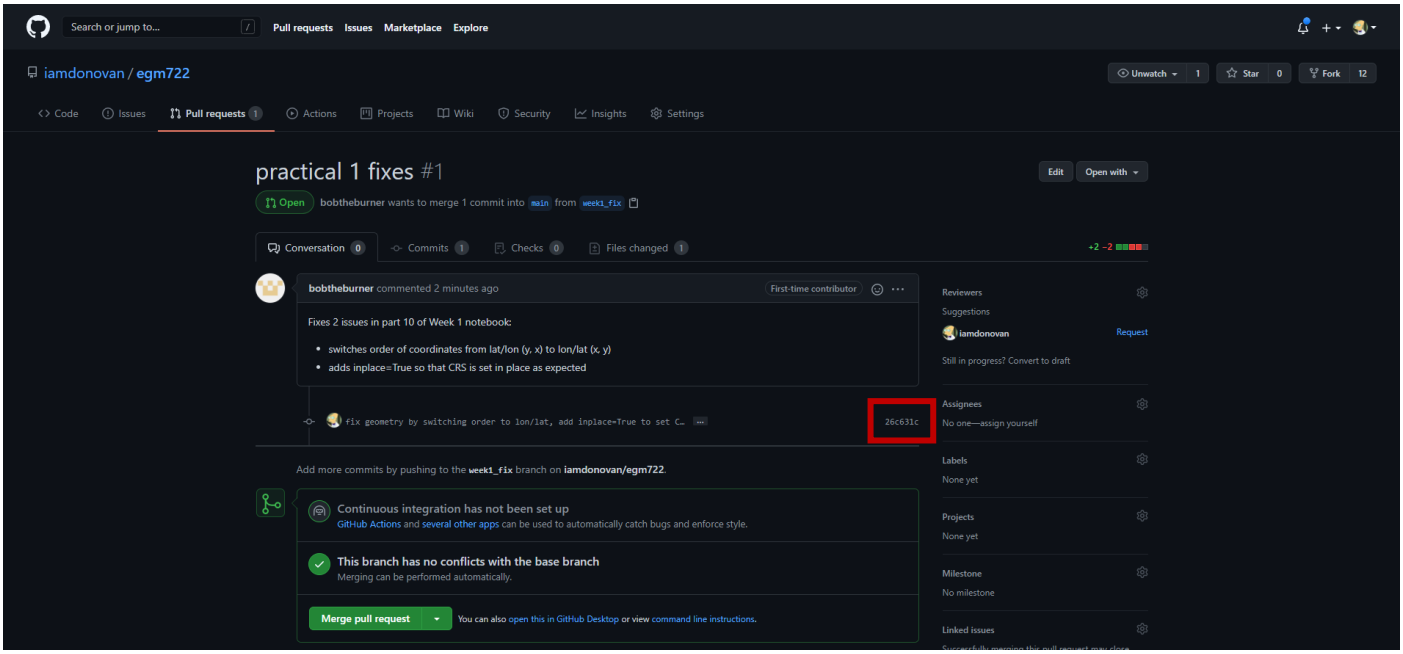
To see any pull requests, you can click the Pull Requests tab at the top of the GitHub page:



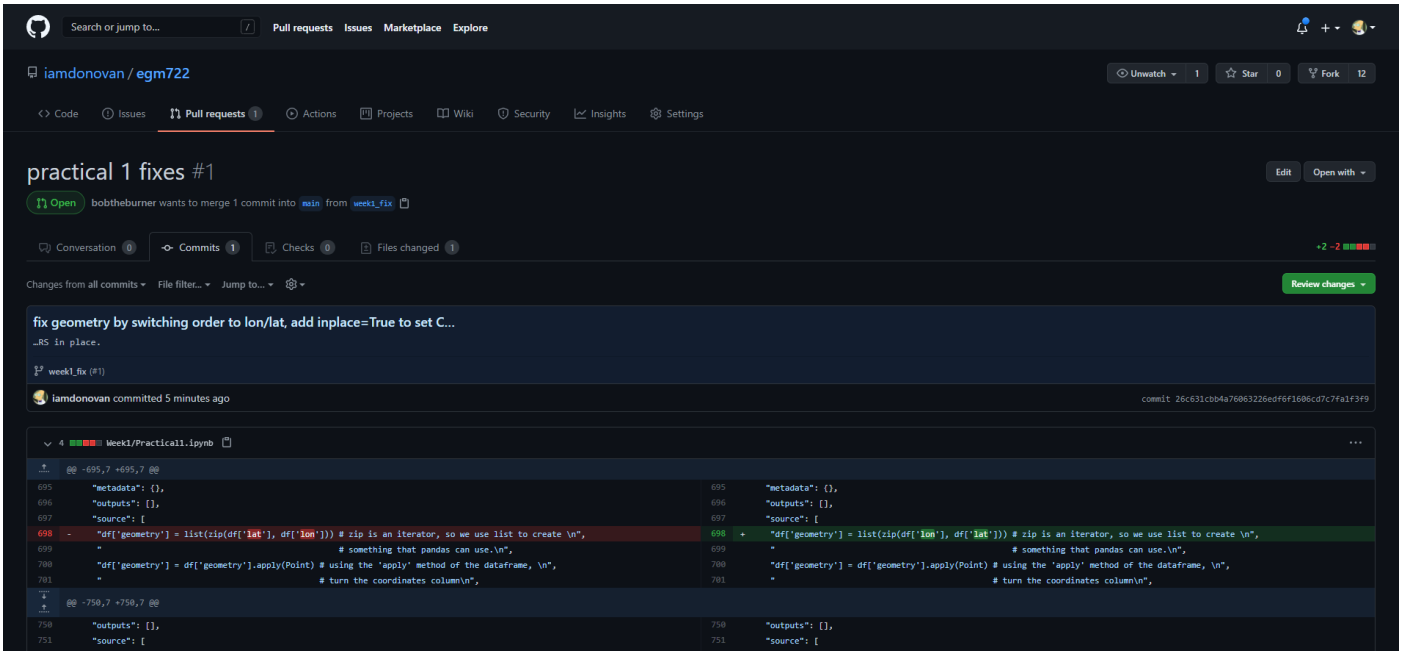
Right now, there aren't any open pull requests. To open a pull request, you can click the green **New pull request** button. From there, you can choose which branch you want to merge into another branch – in this case, we're going to merge the changes from **week1_fix** into **main**. Once you've selected the branches, click "create pull request" to create the pull request.

Reviewing and Merging Pull Requests

Once the pull request has been created, other people are able to review it, comment on it, and "approve" it – but only someone with permission to contribute (change files) to the repository will be able to finish the pull request by merging the branches (note that in the example below, one user, @bobtheburner, has submitted the pull request, but a different user, @iamdonovan, is able to merge the pull request):

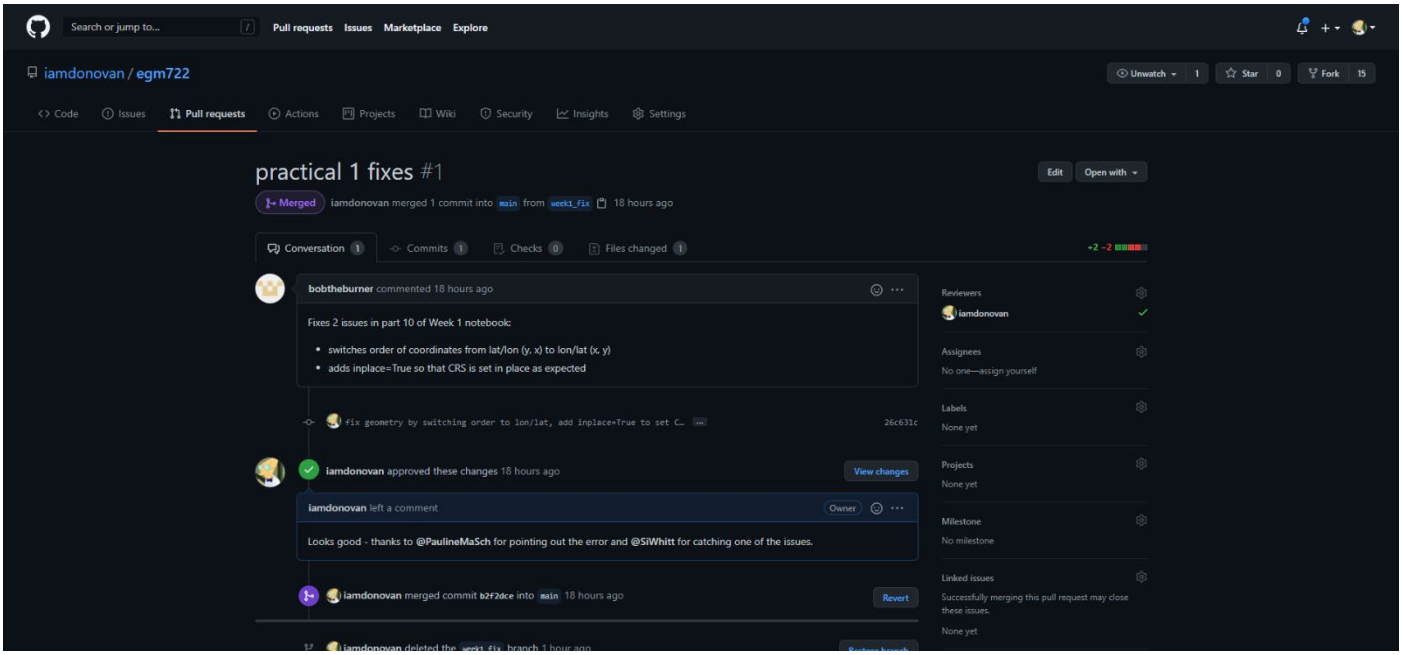


When reviewing the pull request, you can click on the commit hash (highlighted above) for a particular commit to review the changes in more detail:



Here, we can see the previous version on the left, and the new version on the right. From here, you can add comments on individual changes, or provide a review of the commit as a whole. If you think anything needs to be changed, you can also request that those changes are made before approving the pull request.

Once the pull request is approved, the changes will be merged, creating a new commit:



At this point, you can delete the fix branch (if you have the relevant permissions), as all of the changes from the fix branch have been integrated into the main branch.

Note that you can also create pull requests across forks – you do not necessarily need write access to the original repository in order to be able to contribute via pull requests. So, as you are working your way through the rest of the module, if you find any errors that you think should be corrected, you can help fix them by submitting a pull request.