



## Generating an Interactive Map using Python – How To Guide

Stacey Vernon - B00450960

EGM 722 – Programming for GIS & Remote Sensing

## Table of Contents

|   |    |
|---|----|
| 1. Introduction.....                    | 2  |
| 2. Prerequisites .....                  | 2  |
| 2.1 ArcGIS Online (AGOL) Account.....   | 2  |
| 2.2 GitHub Account .....                | 2  |
| 3. Setup .....                          | 2  |
| 3.1 GitHub / Git Desktop Setup .....    | 3  |
| 3.1.2 GitHub Installation.....          | 3  |
| 3.1.3 GitHub desktop installation ..... | 3  |
| 3.2 Clone the repository .....          | 4  |
| 3.3 Conda/Anaconda Setup .....          | 4  |
| 4. Data and Dependencies .....          | 6  |
| 5. Methodology.....                     | 7  |
| 5.1 Overview .....                      | 7  |
| 5.2 Code Breakdown.....                 | 8  |
| 6. Expected Results .....               | 16 |
| 7. Improvements/Roadmap.....            | 17 |
| 8. Troubleshooting .....                | 17 |
| 9. References .....                     | 18 |

## 1. Introduction

Bleeper is one of the licenced bike-share schemes in the Dublin region (Dublin City Council, n.d.). Smart Dublin (an initiative set up by the four Dublin Councils) recently enhanced its data sharing for the Bleeper Bike Scheme. This data is now accessible through an API and adheres to the General Bikeshare Feed Specification (GFBS) (Smart Dublin, 2024), an open-source data standard widely adopted by shared micro-mobility systems (GFBS, n.d.).

Standardised open such as this significantly enhances the visibility and accessibility of bike locations, allowing for comprehensive insights and analysis. For instance, various statistical analyses such as that conducted by Giuffrida et al. (2023) can be performed to evaluate bike-sharing patterns and trends. Given the environmental and health benefits associated with bike-share schemes, availability and analysis of the data are critical for optimising the system and improving accessibility.

ArcGIS Online (AGOL), a Software as a Service (SaaS) platform distributed by ESRI provides powerful tools for generating, sharing, and collaborating on maps and apps (ESRI, n.d.). By integrating real-time data from the Bleeper Bike Scheme with AGOL, this project demonstrates the potential of combining live data feeds with cloud-based Geographic Information System (GIS) technologies.

This project also highlights the role of Python in streamlining and enhancing geospatial workflows. Python's capabilities alongside ESRI's Rest API make it easier to automate data integration, analysis, and visualisation which in the end allows for more efficient and repeatable processes. The integration can simplify a complex geospatial task while creating new possibilities for real-time analysis and decision-making in urban mobility and planning.

## 2. Prerequisites

Before running the script, ensure that you have the following prerequisites in place:

### 2.1 ArcGIS Online (AGOL) Account

- **Sign Up:** If you do not already have an AGOL account, you will need to create one. Visit [ArcGIS Online](#) to sign up for a free account.
- **Organisation Access:** If you are using an existing account ensure that your account has the necessary privileges to publish Feature Layers and create web maps. If you are part of an organisation, check with your administrator to ensure you have the correct permissions.

### 2.2 GitHub Account

- **Sign Up:** If you do not have a GitHub account, you can [sign up for one](#) as all relevant documentation for this project is stored within a GitHub repository.

## 3. Setup

These setup instructions are largely based on practical material provided by Dr. Robert McNabb as part of the Programming for GIS & Remote Sensing module with Ulster University (McNabb, 2024)

### 3.1 GitHub / Git Desktop Setup

#### 3.1.1 Introduction

GitHub is a web-based platform, focused on development, which allows users to version and collaborate their work simultaneously without overwriting each other's work.

#### 3.1.2 GitHub Installation

Navigate to the [Git website](#) and follow the download and installation instructions for your specific operating system.

#### 3.1.3 GitHub desktop installation

GitHub Desktop is the graphical user interface (GUI) designed to work with Git & GitHub. Download and [install from here](#), then log in to your previously created GitHub account once the installation is complete.

### 3.2 Forking the repository

Creating a fork in the repository is the terminology that describes copying the repository you selected to your own GitHub account. This allows you to run and change the code without affecting the original repository.

To fork the [repository for this script](#):

1. Navigate to the repository linked above.
2. Click the 'Fork' button on the top-right corner of the page (Figure 1).

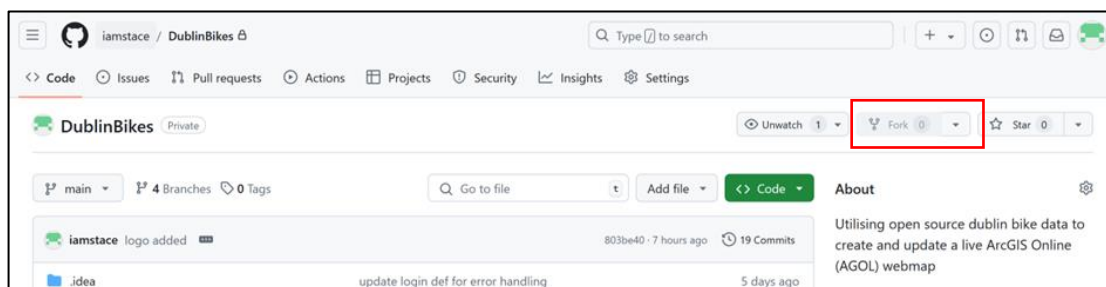


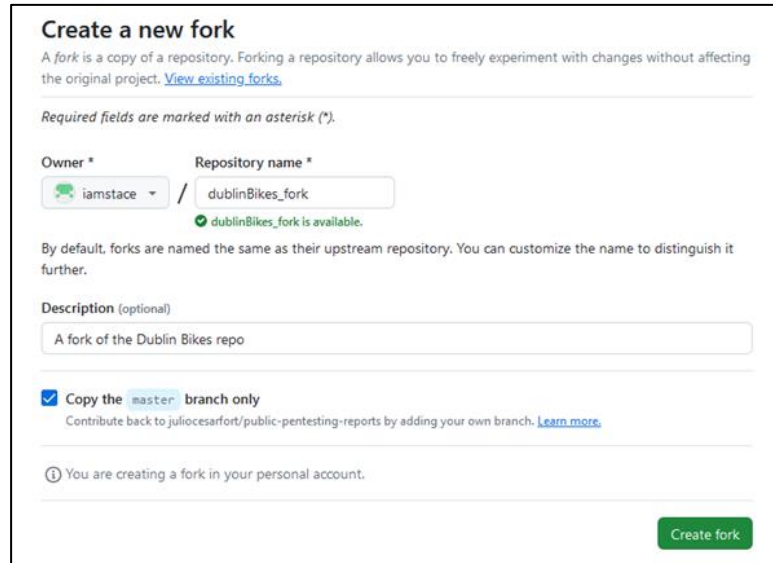
Figure 1: The "fork" option on a GitHub Repository

After selecting Fork, you will receive a pop-up to configure your settings (Figure 2):

1. Set the name and description as desired.
2. Select the "Copy the master branch only" box.



3. Select “Create Fork” to complete the process.



**Create a new fork**

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Required fields are marked with an asterisk (\*).

Owner \* iamstace / Repository name \* dublinBikes\_fork

✓ dublinBikes\_fork is available.

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

☒ Copy the master branch only  
 Contribute back to julioesartort/public-pentesting-reports by adding your own branch. [Learn more.](#)

? You are creating a fork in your personal account.

[Create fork](#)

Figure 2: Options when selecting the fork function on GitHub

### 3.2 Clone the repository

Creating a clone of the repository will create a copy of the repository from your GitHub account to your local drive (Figure 3).

1. Open GitHub desktop and make sure you are logged in to your account.
2. Navigate to File > Clone a repository.
3. You will see all repositories associated with your GitHub Account. Select the one you wish to clone.
4. Choose a suitable file path on your local machine where you would like to store the cloned repository. This will be your working directory.

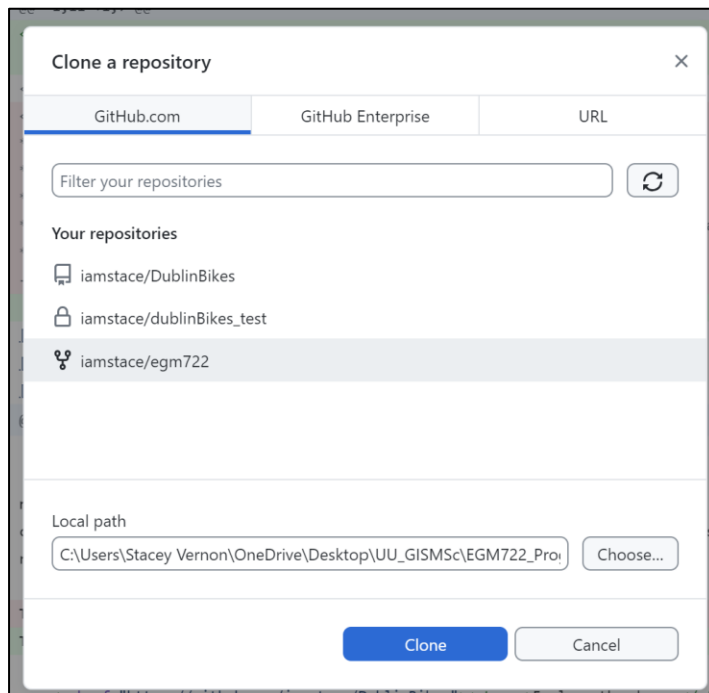


Figure 3: Options to clone your repository using GitHub Desktop.

### 3.3 Conda/Anaconda Setup

### 3.3.1 Introduction

Conda is an open-source package and environment manager. We will be utilising conda through the graphical user interface (GUI), Anaconda Navigator, provided by the Anaconda distribution. If you prefer, Conda can also be used through the command-line interface (CLI) however it will not be documented here.

### 3.3.2 Installation

1. Navigate to the [Anaconda site](#) and follow the download and install instructions for your specific operating system.

### 3.3.3 Environment Setup

1. Launch Anaconda Navigator (Figure 4)
2. Select environments from the left-hand side. If you are using Anaconda Navigator for the first time you will only see the “base (root)” environment.

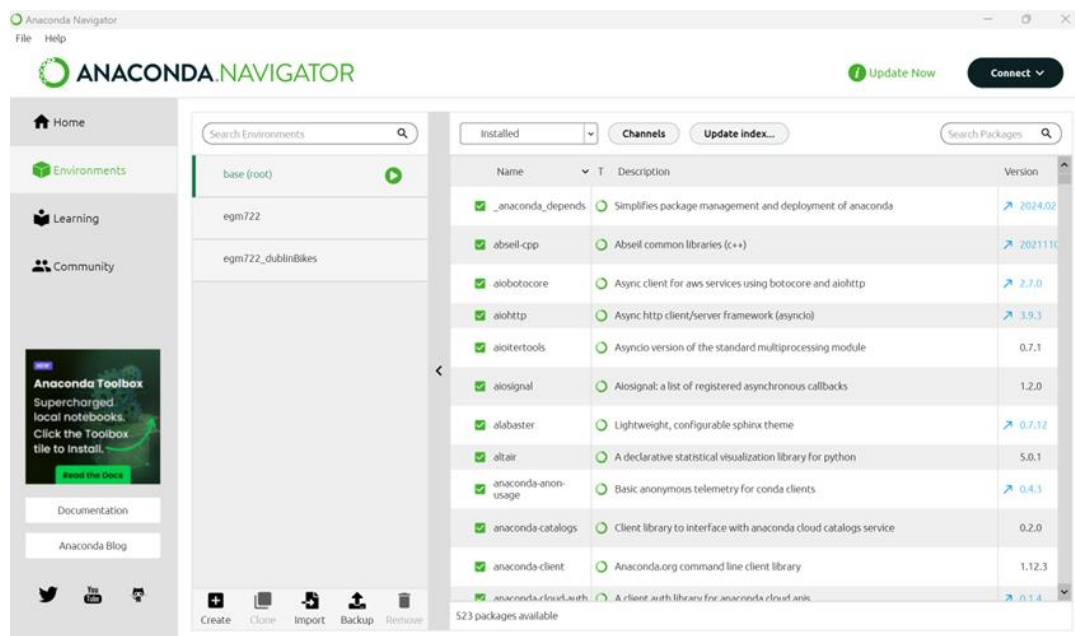


Figure 4: Environments tab in Anaconda Navigator

3. Navigate to your working directory and locate the “dublinBikes\_env.yml” file.
4. Select the import button on your Environments tab in Anaconda Navigator.
5. In the dialogue that appears, input the path to the dublinBikes\_env.yml file in the Local Drive field.
6. If the environment name does not auto-populate, you can enter a name of your choice. If it does auto-populate, you can accept the default name or modify it as needed.
7. Click Import to begin the process.

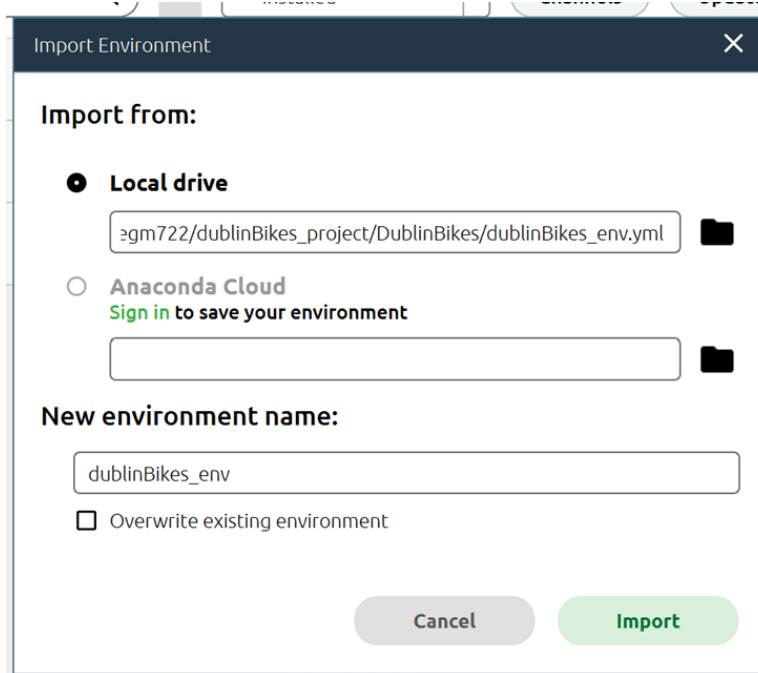


Figure 5: Importing an environment using a '.yml' file on Anaconda Navigator.

8. Once the installation is complete, select “Home” on the left-hand side in Anaconda Navigator. Beside “All Applications” you will see a drop-down list option. This should have automatically set to your new “dublinBikes\_env” but if not, click on the drop-down option and select it (Figure 6).

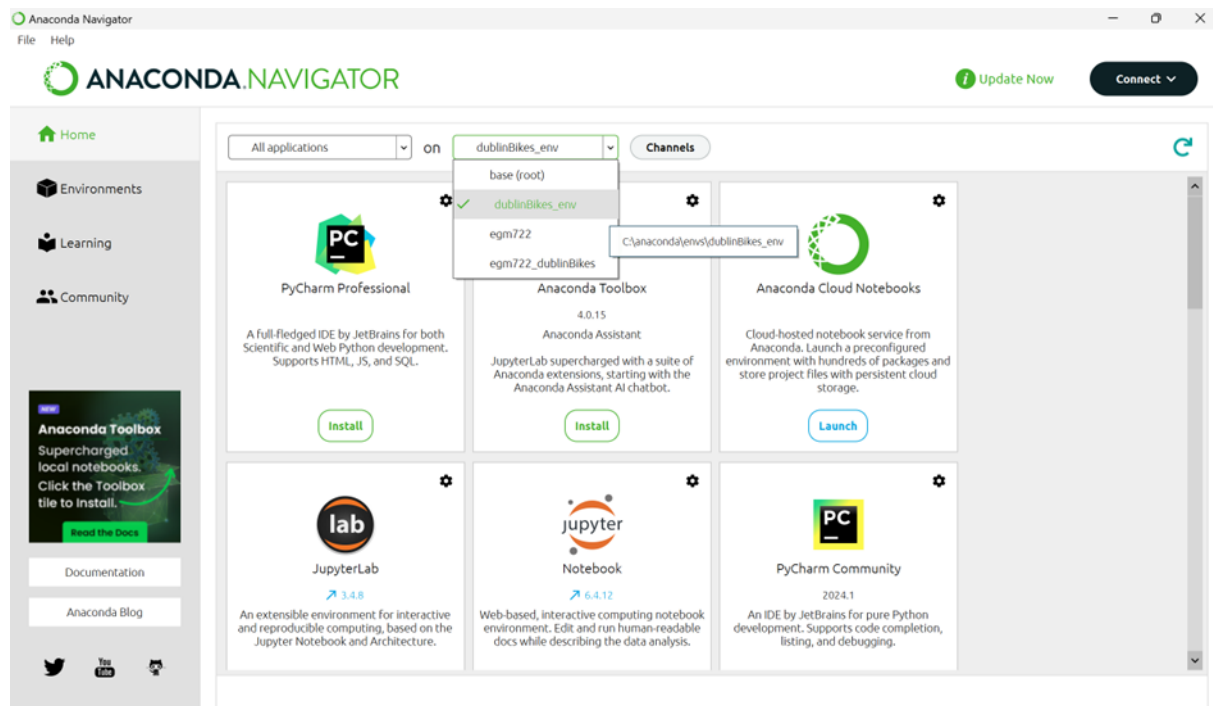


Figure 6: Selecting the environment in Anaconda Navigators 'home' tab.

## 4. Data and Dependencies

All data used in this project is open-source data supplied by Smart Dublin (Table 1).

Table 1: Datasets used in this project.

| Description                            | Data    | Source                     |
|--|---------|----------------------------|
| Bike Stations and Purple Zone Boundary | GeoJSON | Smart Dublin Open Data     |
| Bike Locations                         | GeoJSON | Smart Dublin Open Data API |

To run the script, certain dependencies must be installed and available in the environment. Each of the dependencies listed in the 'dublinBikes\_env.yml' file are outlined in Table 2.

Table 2: Dependencies required for the code and versions where applicable.

| Name       | Version | Description                                |
|------------|---------|--|
| Python     | 3.9     | Programming language and version required  |
| ArcGIS     |         | The ArcGIS API for Python                  |
| Requests   |         | A Python library for sending HTTP requests |
| Pip        |         | Package installer for Python               |
| Jupyterlab |         | Development environment for notebooks      |
| urllib3    | 1.26.5  | Additional HTTP requests library           |

## 5. Methodology

### 5.1 Overview

This tool streamlines the process of working with different data formats by automating key steps. It imports GeoJSON data, converts it into an ArcGIS Online feature service, manages the symbology, and integrates the data into a map. Designed for flexibility, the tool supports straightforward updates to both the data and the map as needed.

The workflow for this process is detailed in Figure XXX, which outlines each step from start to finish.



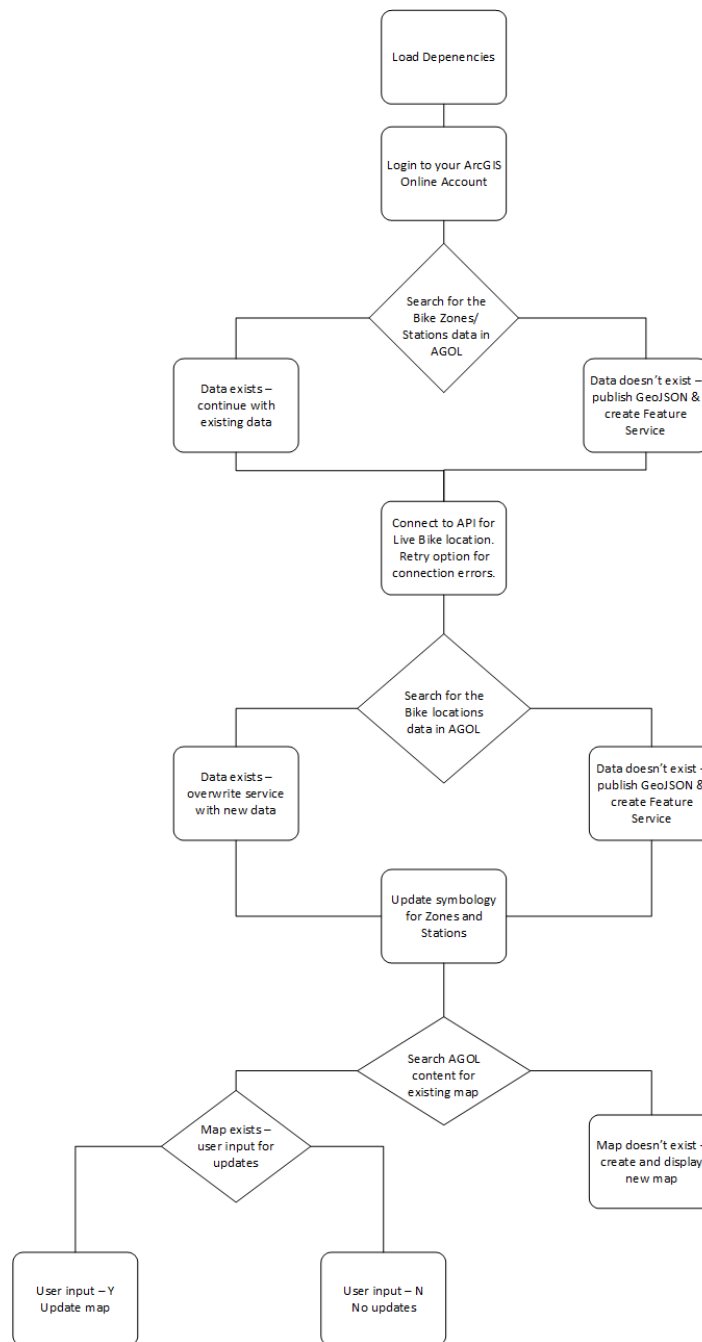


Figure 7: Process workflow

## 5.2 Code Breakdown

The initial section of the code imports the necessary dependencies for the tool's operation, as shown in Code Block 1. Dependencies are loaded within a try-except block to handle any potential errors gracefully.

Error handling is a key feature throughout the code, implemented via try-except blocks to ensure robust and reliable execution of functions. This approach helps in managing exceptions effectively and maintains the tool's stability during operation.

```

1. try:
2.     import time
3.     import getpass
4.     import requests
  
```

```

5.     from arcgis.gis import GIS
6.     from arcgis.features import FeatureLayerCollection
7.     from arcgis.mapping import WebMap
8.
9.     print('Modules loaded successfully')
10.
11. except ImportError as e:
12.     print(f"Error importing module: {e}")
13.

```

Code Block 1: Dependency loading.

Next, you need to log in to ArcGIS Online using your credentials. To enhance security the `getpass` function is employed. This function prompts the user to enter their username and then their password, but the password input is hidden. Once the credentials are provided, the script will confirm whether the login was successful. If the login fails due to reasons other than invalid credentials, the script will return an appropriate error message (see Code Block 2).

```

1. # Login to ArcGIS Online (AGOL) by requesting the user to input a username and password
2. # documentation on getpass: https://docs.python.org/3/library/getpass.html
3.
4. """
5.     Prompt the user to log in to ArcGIS Online (AGOL) by entering their username and
6.     password.
7.
8.     This function uses the `input` and `getpass` modules to securely prompt the user for
9.     their
10.    ArcGIS Online credentials, creates a GIS object, and attempts to log in to the AGOL
11.    portal.
12.
13.    Returns:
14.        GIS: A `GIS` object that represents the authenticated session with ArcGIS Online if
15.        the login is successful.
16.        None: If the login fails (e.g., due to incorrect credentials or an error during the
17.        login process), the function returns `None`.
18.
19.    Raises:
20.        Exception: Any exception encountered during the login process is caught and printed
21.        to
22.        the console, with the function returning `None`.
23.    """
24.
25. def login_AGOL():
26.
27.     try:
28.         # prompt the user to enter their AGOL username
29.         username = input("Enter your ArcGIS Online username:")
30.
31.         # prompt the user to enter their AGOL password (without echoing)
32.         password = getpass.getpass("Enter your ArcGIS Online password:")
33.
34.         # create a GIS object and login to AGOL
35.         gis = GIS("https://www.arcgis.com", username, password)
36.
37.         # confirm the login was successful
38.         if gis.users.me is not None:
39.             print("Login successful. Logged in as:", gis.users.me.username)
40.         else:
41.             print("Log in failed. Please check credentials.")
42.             return None
43.
44.         # return the object
45.         return gis
46.
47.     except Exception as e:
48.         print("An error occurred during login:", e)
49.         return None
50.

```

```

47. # run the login function
48.
49. gis = login_AGOL()
50.

```

Code Block 2: ArcGIS Online login using 'getpass'.

The subsequent steps involve publishing the Bike Stations and Bike Zone area datasets, which are available as "static" datasets from Dublin Smart Bikes (Table 1). The code first searches the users ArcGIS Online (AGOL) content using the specified title to determine whether the dataset already exists. If the dataset is found in the AGOL content, a confirmation message will be printed. If the dataset is not located, the code will proceed to publish the GeoJSON data to AGOL and create a new Feature Layer. In case of any issues beyond these conditions, an error message will be displayed (Code Block 3).

```

1. # Function to publish GeoJSON data to AGOL
2.
3. """
4.     Publishes GeoJSON data related to Dublin Bike Stations and Zones to ArcGIS Online
(AGOL).
5.
6.     This function checks if a feature layer titled "BikeStations_Zones" already exists on
AGOL.
7.     If it does not exist, the function publishes a new feature layer using the provided
GeoJSON data.
8.     If it already exists, the function retrieves the existing layer.
9.
10.    Returns:
11.        FeatureLayer: A `FeatureLayer` object if the GeoJSON data is successfully published
or retrieved.
12.        None: If an error occurs during the process, the function returns `None`.
13.
14.    Raises:
15.        Exception: Catches and prints any exception encountered during the publishing
process.
16.    """
17.
18.
19. def publish_geojson_to_agol(gis):
20.
21.     try:
22.         BikeStationsZone_data = None
23.
24.         # Search for existing feature layers with the title "BikeStations_Zones"
25.         zoneSearch = gis.content.search("title:BikeStations_Zones", item_type="Feature
Layer")
26.
27.         # If no existing feature layer is found, create a new one
28.         if not zoneSearch:
29.             # Set up the feature service properties in AGOL
30.             item_properties = {
31.                 "title": "BikeStations_Zones",
32.                 "description": "Dublin Bike Stations and Zones generated from GeoJSON
provided by Bleeper Bikes",
33.                 "tags": "dublinBikes",
34.                 "type": "GeoJson" # Note: Ensure this is the correct type, usually 'Feature
Layer' is used for publishing
35.             }
36.
37.             # URL link to the GeoJSON file
38.             geojson_url = "https://data.smartdublin.ie/dataset/09870e46-26a3-4dc2-b632-
4d1fba5092f9/resource/40a718a8-cb99-468d-962b-af4fed4b0def/download/bleeperbike_map.geojson"
39.             geojson_item = gis.content.add(item_properties, geojson_url)
40.
41.             # Publish the GeoJSON as a feature layer
42.             BikeStationsZone_data = geojson_item.publish()
43.             print("Data published. New feature layer created.")

```

```

44.
45.     else:
46.         # Handle the case where the feature layer already exists
47.         BikeStationsZone_data = zoneSearch[0]
48.         print("Bike Station and Zone data already exists in AGOL.")
49.
50.         return BikeStationsZone_data
51.
52.     except Exception as e:
53.         print("An error occurred during publishing:", e)
54.         return None
55.
56. # publish the geojson
57. BikeStationsZone_data = publish_geojson_to_agol(gis)
58.

```

Code Block 3: Searching for a service in AGOL content, and publishing a service if it does not exist.

The next dataset to be published is the live bike location data, which differs in handling as Smart Dublin provides an API for accessing real-time information. The initial step involves making a request to the API to retrieve the data. To enhance reliability, the code includes a built-in retry mechanism to handle potential connection issues with the server. This feature allows for a user-defined number of retry attempts (with a default set to 5) and a retry delay (with a default set to 60 seconds), ensuring that data retrieval is successful (Code Block 4).

```

1. # make a request to the rest endpoint and verify the response
2.
3. """
4.     Fetches bike data from the specified API endpoint with retry logic.
5.
6.     This function attempts to retrieve data from a given URL, handling potential
7.     failures by retrying a specified number of times with a delay between attempts.
8.     It parses the JSON response if the request is successful and returns the data.
9.     If all retry attempts fail, it returns `None`.
10.
11.     Args:
12.         url (str): The URL of the API endpoint to fetch the bike data from.
13.         max_retries (int): The maximum number of retry attempts if the
14.                             request fails. Default is 5.
15.         retry_delay (int): The number of seconds to wait between retry
16.                             attempts. Default is 60.
17.
18.     Returns:
19.         dict or None: The parsed JSON data from the API if successful, or `None`
20.                       if all retry attempts fail.
21.
22.     Raises:
23.         requests.RequestException: If a network-related error occurs, the function
24.                                     will handle it and retry according to the
25.                                     retry logic.
26. """
27.
28. # URL to API endpoint for bike locations
29. bike_url = "https://data.smartdublin.ie/bleeperbike-
30. api/bikes/bleeper_bikes/current/bikes.geojson"
31.
32. def fetch_bike_data(url, max_retries=5, retry_delay=60):
33.     attempt = 0
34.     while attempt < max_retries:
35.         try:
36.             response = requests.get(url)
37.             if response.status_code in [200, 201]:
38.                 # Parse the JSON response
39.                 bike_data = response.json()
40.                 return bike_data # Data retrieval successful
41.             else:

```

```

42.             print(f'Failed to retrieve data. Status code: {response.status_code}.
Retrying in {retry_delay} seconds...')
43.         except requests.RequestException as e:
44.             print(f'An error occurred: {e}. Retrying in {retry_delay} seconds...')
45.
46.             # Increment attempt counter and wait before retrying
47.             attempt += 1
48.             time.sleep(retry_delay)
49.
50.         return None # Data retrieval failed after retries
51.
52. # Fetch the bike data
53. bike_data = fetch_bike_data(bike_url)
54.
55. # Output message based on data retrieval success or failure
56. if bike_data:
57.     print('Data retrieved successfully.')
58. else:
59.     print('No data retrieved, cannot proceed with further actions.')
60.

```

Code Block 4: Making a request to the Smart Dublin API.

After successfully retrieving the data from the API, the process for publishing it as an ArcGIS Online (AGOL) feature service mirrors the steps used for the previous dataset. The code first searches for an existing feature service within AGOL. If the service does not already exist, the GeoJSON data is published to AGOL content, and a new feature service is created from this data. If the feature service is found, the code executes an overwrite function to replace the existing data with the new dataset. This ensures that each time the script is run, the dataset in AGOL always reflects the most up-to-date information (Code Block 5).

```

1. # create a FLC from the GeoJSON
2.
3. """
4.     Manages the creation and updating of a Feature Layer Collection (FLC) for live
5.     Dublin Bike data in ArcGIS Online (AGOL) using GeoJSON data.
6.
7.     This script performs the following tasks:
8.     1. Checks if a Feature Layer with a specific title already exists in AGOL.
9.     2. If the Feature Layer does not exist, it uploads a GeoJSON file and publishes
10.    it as a new Feature Service.
11.    3. If the Feature Layer already exists, it overwrites the existing Feature Layer
12.    Collection with new GeoJSON data.
13.
14.    Args:
15.        item_properties (dict): A dictionary containing properties for the item to be
16.        added to AGOL. Must include:
17.            - "title" (str): The title of the item.
18.            - "description" (str): A description of the data.
19.            - "tags" (str): Tags for improving search functionality.
20.            - "type" (str): The type of data ("GeoJson" in this case).
21.        bike_url (str): URL of the GeoJSON file to be uploaded and published.
22.
23.    Returns:
24.        FeatureLayerCollection: A "FeatureLayerCollection" object published if successful.
25.
26.    Exceptions:
27.        - Raises and handles exceptions related to uploading and publishing data.
28.        - Prints messages to indicate success or failure of operations.
29.    """
30.
31. # set up the feature service properties in AGOL for the stations and zone dataset
32. item_properties = {
33.     "title": "Bike_Locations", #provide a title for your item
34.     "description": "Dublin Bike current locations", #add a description of the data
35.     "tags": "dublinBikes", # add tags to data to improve AGOL search functionality
36.     "type": "GeoJson" # define the type of data you are trying to publish

```



```

37.     }
38.
39. # search AGOL contents to check if the layer already exists
40. fl_search = gis.content.search("title:Bike_Locations",item_type="Feature Layer")
41.
42. if not fl_search:
43.     # if search list is empty (meaning a feature layer with that title does not exist),
upload the GeoJSON
44.     # & publish as a new feature service
45.     try:
46.         geojson_item = gis.content.add(item_properties, bike_url)
47.         live_bike_fl = geojson_item.publish()
48.
49.         print("Successfully created new feature service and published live bike location")
50.
51.     except Exception as e:
52.         if "already exists" in str(e):
53.             print("Bike_Locations has been created")
54.         else:
55.             print(f"An error occurred during the process: {e}")
56.
57. else:
58.     # if search list is NOT empty (meaning feature layer with that title already exists),
overwrite the data within the FL
59.     live_bike_fl = fl_search[0]
60.     content = FeatureLayerCollection.fromitem(live_bike_fl)
61.
62.     try:
63.         # overwriting the FLC data with new data
64.         overwrite_result = content.manager.overwrite(bike_url)
65.
66.         # overwrite_result will have {"success":True} or {"success":False} - check the
result
67.         if not overwrite_result['success']:
68.             print("Failed to overwrite Feature Layer Collection with new GeoJSON Data")
69.         else:
70.             print("Successfully updated Feature Layer Collection with new GeoJSON Data")
71.
72.     except Exception as e:
73.         print("An error occurred during the process: {e}")
74.

```

Code Block 5: Generating or overwriting a feature layer.

To enhance visibility on our map, we will customise the symbology for the Dublin Bike Stations and Zone layers, while keeping the live bike locations at their default settings. The code first checks for the presence of the 'drawingInfo' key, which is crucial for updating the symbology. If the layer is newly created, this key will not be present and so the code will add it. For existing layers, the 'drawingInfo' key will only need to be updated (Code Block 6).

The symbology updates are:

- Bike Zone Layer: Outlined in purple with no fill.
- Bike Stations Layer: Displayed as black triangles.

This approach ensures that the map visually distinguishes between different data types, improving overall readability.

```

1. def update_or_add_symbology_for_flc(flc):
2.     """
3.     Updates or adds the symbology of the layers within a FeatureLayerCollection.
4.
5.     :param flc: The FeatureLayerCollection object whose layers' symbology will be updated or
added.
6.     """
7.     try:
8.         # Define the symbology for point and polygon layers
9.         point_symbology = {

```

```

10.         "renderer": {
11.             "type": "simple",
12.             "symbol": {
13.                 "type": "esriSMS",
14.                 "style": "esriSMSTriangle",
15.                 "color": [0, 0, 0, 255], # Black
16.                 "size": 5
17.             }
18.         }
19.     }
20.
21.     polygon_symbology = {
22.         "renderer": {
23.             "type": "simple",
24.             "symbol": {
25.                 "type": "esriSFS",
26.                 "style": "esriSFSolid",
27.                 "color": [0, 0, 0, 0], # No fill
28.                 "outline": {
29.                     "color": [128, 0, 128, 255], # Purple outline
30.                     "width": 2
31.                 }
32.             }
33.         }
34.     }
35.
36.     # Iterate over each layer in the FeatureLayerCollection
37.     for layer in flc.layers:
38.         # Define the symbology to update
39.         symbology_update = None
40.         layer_info = layer.properties
41.         drawing_info_exists = "drawingInfo" in layer_info
42.
43.         if layer_info['geometryType'] == "esriGeometryPoint":
44.             symbology_update = point_symbology
45.             print(f"Updating symbology for point layer: {layer_info['name']}")
46.         elif layer_info['geometryType'] == "esriGeometryPolygon":
47.             symbology_update = polygon_symbology
48.             print(f"Updating symbology for polygon layer: {layer_info['name']}")
49.         else:
50.             print(f"Layer type {layer_info['geometryType']} not supported for symbology
update.")
51.             continue
52.
53.         # Prepare definition to update or add
54.         current_definition = {}
55.         if drawing_info_exists:
56.             # If drawing info exists, only update it
57.             current_definition["drawingInfo"] = symbology_update
58.             layer.manager.update_definition(current_definition)
59.             print(f"Symbology updated for layer: {layer_info['name']}")
60.         else:
61.             # If drawing info does not exist, add it
62.             current_definition["drawingInfo"] = symbology_update
63.             layer.manager.add_to_definition(current_definition)
64.             print(f"Symbology added for layer: {layer_info['name']}")
65.
66.     except Exception as e:
67.         print(f"Failed to update or add symbology for the FeatureLayerCollection: {e}")
68.
69. # Assuming `BikeStationsZone_data` is a FeatureLayerCollection object
70. update_or_add_symbology_for_flc(BikeStationsZone_data)

```

Code Block 6: Updating dataset symbology.

The last step is to assemble the map using the newly created data. As with previous steps, we first search for the map title in our content to check if the map already exists. If the map is found, the script will confirm or it will state that the map is not available in AGOL content (Code Block 7).

```

1. # Search AGOL content to see if a map has already been created
2.
3. """
4.     Searches for an existing web map in ArcGIS Online (AGOL) with a specified title.
5.
6.     This code checks AGOL content to see if a web map using a given title already exists.
7.     If a map with the specified title is found, it retrieves the web map item and creates
8.     a `WebMap` object from it. If no map with the title is found, it prints a message
9.     indicating that no existing map was found.
10.
11.     Args:
12.         map_title (str): The title of the web map to search for in AGOL.
13.
14.     Returns:
15.         Webmap: A `WebMap` object if a web map with the specified title is found in AGOL.
16. """
17.
18. # the title we are searching for
19. map_title = "DublinBike_LocationMap"
20.
21. # searching in AGOL for these maps
22. existing_maps = gis.content.search(query=f"title:'{map_title}'", item_type = "Web Map")
23.
24. if existing_maps:
25.     # retrieve the existing web map item
26.     web_map_item = existing_maps[0]
27.     web_map = WebMap(web_map_item)
28.     print ("Map already exists within AGOL Content")
29.
30. else:
31.     print("No existing map found")
32.

```

Code Block 7: Searching for a map in AGOL.

If the map does not already exist, the script will create a new map and display it to the user. If the map is found, the script will prompt the user with the question: "Do you want to update the map?" This prompt allows the user to include any layers they may have initially forgotten.

- **If the user inputs "Yes"**, the script will call the `layers_to_add` function to identify and add any missing layers. Users should specify any additional layers they wish to include at this stage.
- **If the user inputs "No"**, the script will confirm that no updates are required.

In all cases, the map will be displayed within the script, and a link to the AGOL map will be provided for easy access (Code Block 9).

```

1. def manage_web_map(gis, map_title, live_bike_fl, BikeStationsZone_data):
2.     """
3.     Updates an existing web map or creates a new one in ArcGIS Online (AGOL) based on
4.     user input and layer existence.
5.
6.     Args:
7.         gis (GIS): The authenticated GIS object.
8.         map_title (str): Title for the web map.
9.         live_bike_fl (FeatureLayer): The live bike feature layer to add to the map.
10.        BikeStationsZone_data (FeatureLayer): The bike stations/zone feature layer to add to
the map.
11.    """
12.
13.    # Search AGOL content to see if the map already exists
14.    existing_maps = gis.content.search(query=f"title:'{map_title}'", item_type="Web Map")
15.
16.    if existing_maps:
17.        # Retrieve the existing web map item

```

```

18.     web_map_item = existing_maps[0]
19.     web_map = WebMap(web_map_item)
20.
21.     # Ask user if they want to update the map
22.     user_input = input("Do you wish to update the map? (Y/N): ").strip().upper()
23.
24.     if user_input == 'Y':
25.         # Get existing layer IDs from the web map
26.         existing_layer_ids = {layer['id'] for layer in web_map.layers}
27.
28.         # Add layers if they are not already present
29.         layers_to_add = [live_bike_fl, BikeStationsZone_data]
30.         for layer in layers_to_add:
31.             if layer.id not in existing_layer_ids:
32.                 web_map.add_layer(layer)
33.                 # Debugging: Check the layer attributes
34.                 print(f"Layer added: ID = {layer.id}")
35.             else:
36.                 print(f"Layer with ID '{layer.id}' already exists in the web map.")
37.
38.         # Update the web map
39.         web_map.update()
40.         web_map_url =
f"https://www.arcgis.com/home/webmap/viewer.html?webmap={web_map_item.id}"
41.         print(f"Web map updated successfully. Access the map here: {web_map_url}")
42.
43.     else:
44.         # Construct a URL to access the existing map
45.         web_map_url =
f"https://www.arcgis.com/home/webmap/viewer.html?webmap={web_map_item.id}"
46.         print(f"No updates required. Access the existing map here: {web_map_url}")
47.
48.     else:
49.         # Create a new web map if it does not exist
50.         web_map = WebMap()
51.         web_map.add_layer(layer=live_bike_fl)
52.         web_map.add_layer(BikeStationsZone_data)
53.
54.         web_map_properties = {
55.             "title": map_title,
56.             "description": "A web map of live Dublin Bike locations and their zones/stations
obtained from Smart Dublin API",
57.             "tags": "Dublin Bike, locations, live webmap",
58.             "snippet": "A web map of Dublin Bike locations"
59.         }
60.
61.         # Save the new web map
62.         web_map_item = web_map.save(item_properties=web_map_properties)
63.         web_map_url =
f"https://www.arcgis.com/home/webmap/viewer.html?webmap={web_map_item.id}"
64.         print(f"New web map created and saved. Access it here: {web_map_url}")
65.
66.     try:
67.         display(web_map)
68.     except ImportError:
69.         print("Display functionality not available in this environment.")
70.
71. # Call the function to manage the web map
72. manage_web_map(gis, "DublinBike_LocationMap", live_bike_fl, BikeStationsZone_data)

```

Code Block 9: Creating or updating our AGOL map.

## 6. Expected Results

By following the steps outlined in this project, the expected outcome is a fully functional AGOL map that displays the Dublin Bike Stations, Bike Zones, and live bike locations with customised symbology. The map is interactive and accessible via a provided AGOL link. The layers will be updated or newly

created, ensuring that the most current data is visualised. Figure 8 shows the expected outcome of map production.

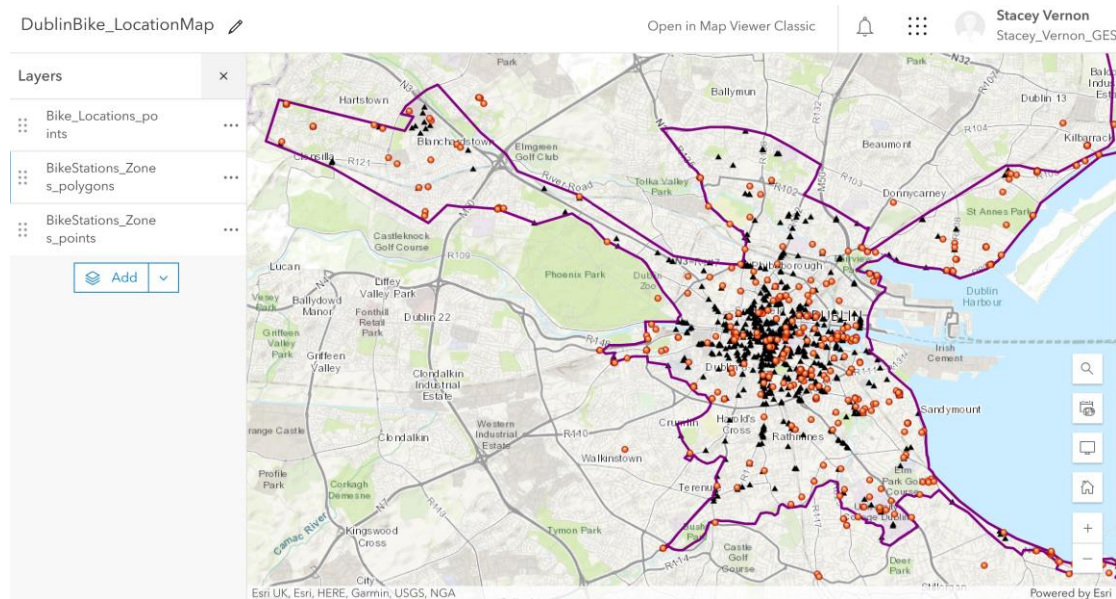


Figure 8: Snapshot of the AGOL map created.

## 7. Improvements/Roadmap

This analysis opens the door to valuable insights that can inform decision-making in urban planning and transportation. Future enhancements to this script will enable more advanced spatial analysis of the data. Firstly, we will generate a heatmap of high-density bike locations and overlaying this data with the current Dublin Bike Lane infrastructure can help assess where accessibility needs to be improved.

Additionally, the script will be enhanced to analyse bike locations outside the designated zone boundaries. A visual inspection shows that several bikes are often located north of Dublin, along the coast. This is known popular tourist destination with nice beaches. Further analysis could quantify the frequency of journeys beyond the current zone, providing data-driven recommendations for extending the parking boundaries to include popular tourist destinations in these areas. This would enhance the accessibility and convenience of the bike-share scheme, catering to user needs more effectively.

## 8. Troubleshooting

ESRI have recently created a “Recycle Bin” feature on ArcGIS Online. This can cause some issues with the code (Specifically block 3 & 4).

You may get an error similar to Figure 9 stating that the service already exists. In this case you can navigate to ArcGIS.com and then to your content. If you see that the Feature Layer dataset exists and you are happy to use it, you can move on to the next code block.

```
return BikeStationsZone_data

except Exception as e:
    print("An error occurred during publishing:", e)
    return None

# publish the geojso
BikeStationsZone_data = publish_geojson_to_agol(gis)

An error occurred during publishing: {'message': "Service name 'BikeStations_Zones' already exists for '0PMINDiTrbaOyMc'"}
```



Figure 9: Potential error message.

However, it may happen that the code has published the GeoJSON but not the Feature Layer. In this case, you can select and delete the GeoJSON. **Next**, you need to navigate to the Recycle Bin on the left-hand side and delete the content from here as well (Figure 10).

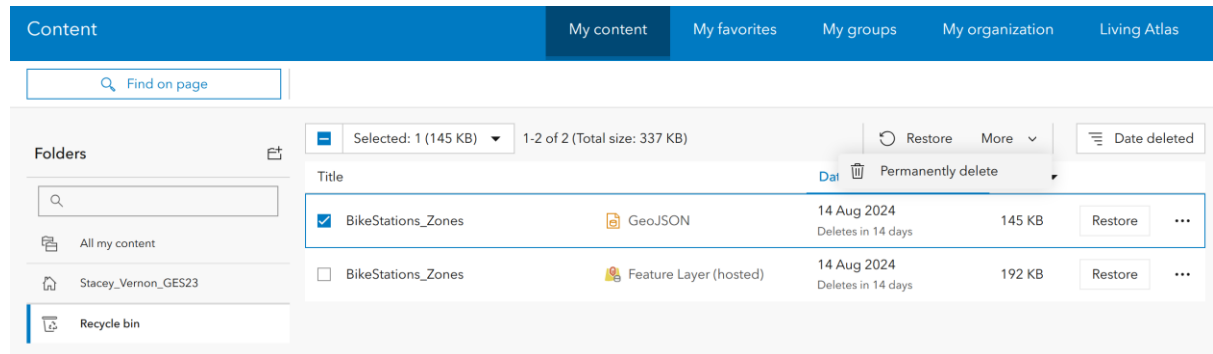


Figure 10: Permanently deleting data from AGOL.

## 9. References

- ESRI. *An Introduction to ArcGIS Online—ArcGIS Online Help | Documentation*. Available at: <https://doc.arcgis.com/en/arcgis-online/get-started/what-is-arcgis.htm> (Accessed: August 13, 2024a).
- Co-pilot, M. (2024) *Image Creator*. Available at: <https://copilot.microsoft.com/images/create/dublin-bikes2farc-gis-online-mapping-artistic-stree/1-66bcaae751794598aa79170742a16341?id=YvZZ%2b5dKSRG5koqx3dJebQ%3d%3d&view=detailv2&idpp=genimg&idpclose=1&thId=OIG2.KpJCY0EjTswBOuxYVMap&skey=LA3ocg8K9OmHJNDyqIKBxHdQHf73ojGjShczsM9dgEM&FORM=SYDBIC> (Accessed: August 14, 2024).
- Dublin City Council. *Bleperbike - Dataset - Data.Gov.ie*. Available at: <https://data.gov.ie/dataset/bleperbike> (Accessed: August 13, 2024).
- Smart Dublin. (2024) *Dublin's Bike Share Schemes: Fuels Interoperability via Open Data - Smart Dublin. Smart Dublin*. Available at: <https://smartdublin.ie/dublins-bike-share-schemes-open-data-interoperability/> (Accessed: August 13, 2024).
- GFBS. *FAQ - General Bikeshare Feed Specification*. Available at: <https://gbfs.org/learn/faq/#> (Accessed: August 13, 2024b).
- Giuffrida, N., Pilla, F. and Carroll, P. (2023) "The Social Sustainability of Cycling: Assessing Equity in the Accessibility of Bike-Sharing Services." *Journal of Transport Geography*, 106, p. 103490. DOI: 10.1016/J.JTRANGE.2022.103490.
- McNabb, R. (2024c) *Setup — Space Cameras and Glaciers*. Available at: <https://iamdonovan.github.io/teaching/egm722/setup/index.html> (Accessed: August 14, 2024).