mapmakers

Release

Erik Rose

Jan 07, 2025

List of Tables

Chapter 1

Introduction

1.1 Why Use Webmaps as Templates?

When the GIS team wants to share spatial data with a general audience, we publish the layers as a feature service, either on AGOL or our Enterprise SDE instance. Within ArcGIS Pro, we can tweak the symbology, the labeling, and the popup info, so why does a Webmap become necessary? As it turns out, many of the settings in ArcGIS Pro are not compatible with viewing on the web. Complex feature symbology may fail to build, and will fall back to a default color and symbol type that is probably not what you want (this is particularly common when using hatching). ArcGIS Pro uses the MapLex labeling engine, whereas the online web server does not. Since we want our users to consume our published services via Webmaps, using a Webmap as a template ensures that the symbol and label definitions we select will appear and function correctly in their destination environment.

In addition, there are several "server-side" settings that can only be adjusted once the service is imported into a Webmap. Enabling popups, and making the fields of individual layers searchable, are examples of settings that can only be set after importing into a Webmap. Saving these settings in a template Webmap makes it easy to reference the implementation and reproduce the build when making a new map. The use of templates reduces the need for manually copying maps and ajusting settings using the AGOL GUI interface, saving time and clicks.

Chapter 2

Installation

2.1 Using `pip`

```
(.venv) $ pip install mapmakers
```

2.2 Using Github

```
(.venv) $ pip install pip@git+https://github.com/grantspassoregon/mapmakers
```

Chapter 3

Logging into ArcGIS

In order to commit updates to Webmaps, either on AGOL or the internal portal, the `mapmakers` package must be logged in to your user account. The City of Grants Pass requires multi-factor authentication, so the sign-in process cannot be fully automated. Instead, you will prepare your log-in credentials within a `.env` file, and log in using custom login script, then you will be able to load and use `mapmakers` normally. The steps are as follows:

- Obtain a client id.
- Save the client id to a `.env` file in your working directory.
- Run the login script in "/exmaples/grants_pass/login.py".
- Import `mapmakers` into a build script for your new Webmap.
- Run the build script.

3.1 Obtain a Client ID

The instructions to obtain a client ID come from the "User authentication with OAuth 2.0" section from this page.

The steps below show how a client id can be obtained by registering a new application with you GIS. Only one client id is required, so if your GIS has one already, you may use it instead of creating a new application.

- Log in to your ArcGIS Online or ArcGIS Enterprise organization.
- Go to the Content tab.
- Click 'New item', then select 'Application'.
- On the 'New item' dialog, select 'Other application' and click 'Next'.
- Type in a title for the application item. Optionally, specify the folder, tags and summary.
- Click 'Save' to add the item.
- On the item details page of this newly created application, browse to the Credentials section and find the Client ID. The string listed is the value that you will pass in as the `client_id` parameter when creating the GIS object.

3.2 Setting Up the `.env` File

The client ID is an example of a form of credentials that is not easy to memorize. However, entering the client ID directly into our code would be insecure. Anybody on Github could steal your credentials. *Environmental variables* are a way to store these sensitive credentials on your local machine, and

load them into the workspace later when needed. The convention is to keep these variables stored in a file called `.env`. The dot at the beginning of the file name denotes that it is a hidden file, and will not appear by default within File Explorer on Windows.

At the start of the login scripts at "/examples/grants_pass/login.py", we use the following import statement and function call:

```
from dotenv import load_dotenv

load_dotenv()
```

The *load_dotenv* function from the `dotenv` package will read the contents of the `.env` file and allow you to access the variables that it defines. In this case, we are storing the client id as an environmental variable named `CLIENT_ID`. We specify the value of an environmental variable using the variable name, followed by an equal sign, followed by the value wrapped in quotes.

Here is an example of how to define the `CLIENT_ID` variable in a `.env` file:

```
CLIENT_ID="my_client_id"
```

Replace *"my_client_id"* with the actual value of your client ID.

The `.env` file will not exist in your project until you create it. Use the text editor of your choice, and make sure to include the dot before *env* for the title. If the text editor sneakily adds a ".txt" or some other extension on the end of the file name, rename the file as `.env` and ignore any cautionary warnings about changing the file type.

3.3 Running the Login Script

In it's entirety, the login script consists of:

```
from arcgis.gis import GIS
from dotenv import load_dotenv
import os
import logging

logging.basicConfig(
    format="%(asctime)s %(message)s",
    datefmt="%m/%d/%Y %I:%M:%S %p",
    level=logging.INFO,
)

load_dotenv()
PORTAL = "https://grantspassoregon.maps.arcgis.com/"
CLIENT_ID = os.getenv("CLIENT_ID")
logging.info("Environmental variables loaded.")
GIS_CONN = GIS(PORTAL, client_id=CLIENT_ID)
```

The import statements tell us that we are using the *GIS* class from the *arcgis* Python API, as well as *load_dotenv*, the *os* package, and the *logging* package. The next block configures logging to include the date and time of the message.

After loading our environmental variables with *load_dotenv()*, we define the portal path as *PORTAL*. If you instead use the path for our internal portal ("https://gis.grantspassoregon.gov/arcgis/home"), make sure that you obtain your client ID from the internal portal and not AGOL. This example is configured to use AGOL.

Note that *load_dotenv* does not do the whole job of reading your environmental variables into the workspace, you will still need to access the value by calling *os.getenv()*. As an argument, *os.getenv* takes the name of the variable that you have defined in your `.env`, in this case "CLIENT_ID".

When the last line of the script executes, it will attempt to open a GIS connection and save that connection as the variable *GIS_CONN*. ESRI describes this process as an "interactive login experi-

ence". First the terminal will prompt you to authenticate using SAML, reading:

```
Enter code obtained on signing in using SAML:
```

A browser window will then open leading to a plain-Jane web page with the heading **OAuth2 Approval**. The instructions will read "Please copy this code, switch to you application and paste it there:" and below will be a text field with a long string of text. Copy this text, switch back to the terminal, and paste the authentication string there. If you are using Windows Terminal, pressing `Ctrl` + `Shift` + `v` will paste the text.

Note that at the time of writing, after pressing `Enter` you will receive an *InsecureRequestWarning* advising you to add certificate verification. You can safely ignore this warning, as we are logging in using the recommended method from ESRI. Addressing this warning is a concern for the maintainers of the *arcgis* Python API.

Upon successful completion, this script will add the variable *gis* to your Python environment, which holds a reference to the authenticated GIS connection. The `mapmakers` library assumes the existence of the *GIS_CONN* variable. If the GIS connection is not named *GIS_CONN*, or if you forget to run the login script prior to using the library, you will receive an error message that *GIS_CONN* is unassigned. Admittedly, this device is a bit a crude hack to get around the restrictions of multi-factor authentication, and it is not best practice to use a variable in a library that is not necessarily assigned. I am open to ideas for a better way to do this.

To execute the login script, navigate to the `mapmakers` package location on your machine and open a session of Python from the terminal:

```
cd path/to/mapmakers
python
```

In the Python interactive shell, enter the following command:

```
> exec(open("examples/grants_pass/login.py")).read())
```

If your working directory is different than the location for the `mapmakers` package, you will need to adjust the path to the login script. The *open* command specifies for Python to open the file at the indicated path, and the *read* methods reads the contents of the file into memory. The *exec* command executes any commands contained in the contents of the file. We make use of this pattern to read and execute scripts from the Python shell.

3.4 Import `mapmakers` Into Your Build Script

The build script is a python file that contains the instructions for building your Webmap. This script is where you will import and use the `mapmakers` package. To learn more about how to use the classes and methods in the `mapmakers` package, see Making Maps.

Import `mapmakers` into your script the same way you would any other package:

```
import mapmakers as m
```

Place the import statement at the top of the file. Here we have assigned the alias *m* to the package name using *as*.

3.5 Running Your Build Script

Run your build script using the same pattern that we used for the login script. From the Python interactive shell, enter the command:

```
> exec(open("path/to/my/script.py").read())
```

Replace *"path/to/my/script.py"* with the absolute or relative path to your build script. Make sure that

you have logged in first using the Section 3.3, or the `mapmakers` package will throw an error that *GIS_CONN* is undefined.

Chapter 4

Template Creation and Use

Any Webmap can serve as a template. To use a Webmap as a template, you only need the item ID. On the Overview page of AGOL or the internal portal, the item ID is in the *Details* section of *Item Information*, on the right side of the page. The ID is also listed in the address string of the browser, following "id=". Because an item ID is not a very memorable string of characters, we will assign a nickname to each ID. First we create an empty dictionary, then add the desired templates using the nickname as a key and the item ID as the value.

```
# remember to import the mapmakers package
import mapmakers as m

templates = {}
templates.update({"aerials": "553b8d97bbeb415c81e41bbb0649d66a"})
templates.update({"boundaries": "c65bd751b50441cd9e07f9607f151b34"})
templates.update({"plss": "f1e719ca11d248178d2d886654b38f44"})
templates.update({"property": "c984c06eaf6f4869aab5ec3eaac999df"})
```

Here the *templates* variable defines an empty dictionary, and we add ids for four Webmaps. These Webmaps are copies of templates that we use for current city services. The keys that you use will not appear on the final map product, they are human-readable nicknames intended to make your code easier to read and understand.

4.1 The *from_obj* Method

The `mapmakers` package has a *Templates* class for managing template data. We can create an instance of the *Templates* class from the dictionary of names and ids using the *from_obj* method in the *Templates* class. The *from_obj* method is static, meaning that we must call it using the abstract class name. The method takes a dictionary as the first argument, and it assumes that the dictionary is structured as above, using nicknames for keys and item IDs for values. The second required argument is a reference to the authenticated GIS connection, which in the `mapmakers` package is named *GIS_CONN*. You must run the Section 3.3 to create the *GIS_CONN* variable.

Continuing our example, we can instantiate a *Templates* object using the following command:

```
# here we alias GIS_CONN as gis
gis = GIS_CONN
# load templates from their item IDS
tmp = m.Templates.from_obj(templates, gis)
```

While the templates are loading, a progress bar will appear and increment as each new template is loaded. In this case, there are only four templates, but the web viewer requires more than a dozen, and the operation can take some time to complete, so we use the progress bar to indicate that the program is still working.

The *Templates* class has only a single property, *template*. The *template* property contains a dictionary, using the same nicknames as keys that were specified in the *templates* variable we created earlier.

Instead of using item IDs as values, however, the *Templates* class has loaded all of the mapping information associated with the item ID, and it stores this information in a *Template* object. A *Template* is another class unique to the `mapmakers` package. Each *Template* object corresponds to a single Webmap, and the *Templates* object holds references to a number of *Template* objects. Since the *template* field of *Templates* is a dictionary, we can check the contents using the *keys*, *values* or *items* methods.

```
tmp.template.keys()
```

Output:

```
dict_keys(['aerials', 'boundaries', 'plss', 'property'])
```

The *Template* class has the properties *name* and *id*, and these correspond to the nickname and item ID that we used to create the template above. The *Template* class also has a field called *items*, and this contains another dictionary. Each entry in the dictionary represents a layer on the target Webmap. The value of the dictionary is a class called *TemplateItem*. The *TemplateItem* class holds all information related to a specific layer, including properties for *title*, *group_name* (the template name), *item_name* (the layer name), *layer_definition*, *popup_info*, *url* and *search*. We can select an individual template from a *Templates* object by including the dictionary key in brackets after the *template* field:

```
prop = tmp.template["property"]
prop.items.keys()
```

Output:

```
dict_keys(['property_0', 'property_1', 'property_2', 'property_3'])
```

4.2 The *with_names* Method

From the output, we can infer that there are four layers on the "property" Webmap. When creating a *Templates* instance using the *from_obj* method, the *item_name* of each *TemplateItem* is automatically assigned. If you want to use more memorable or human readable names for each layer, there are a couple methods for doing this. The first approach is to use the *with_names* method from the *Template* class.

The *with_names* method assumes that you know beforehand the number and order of layers in a template Webmap. Since this is a method of the *Template* class, make sure that you are calling the method from a *Template* object. Pass in as an argument a vector of names, with the requirement that the number of layers in the Webmap and the number of names in the vector must match.

```
prop = prop.with_names(["taxlots", "footprints", "subdivisions", "addresses"])
prop.items.keys()
```

Output:

```
dict_keys(['taxlots', 'footprints', 'subdivisions', 'addresses'])
```

When reading layers from a Webmap on AGOL or our internal portal, the ordering is the inverse of its order on the Webmap. This is an artifact of the how ESRI creates Webmaps, where the first layer assigned to a map goes to the bottom of the rendering stack, and subsequent layers go on top (as seen in the image below). In a future update we may elect to reverse the ordering here, to match the "display order" in the map. Let us know if this is a feature you would like to use.

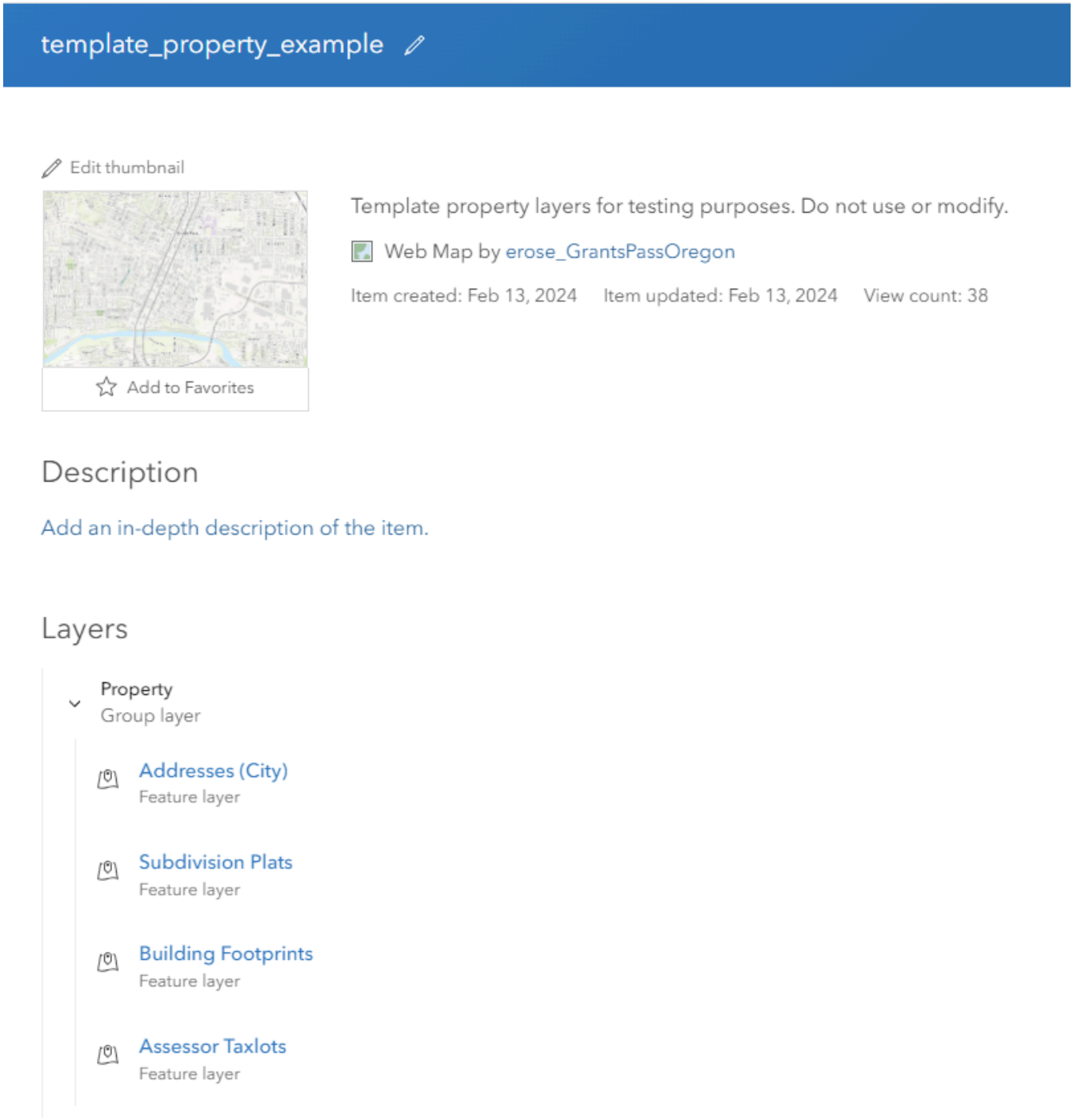

Figure 4.1. Order of layers in the "property" template on AGOL.

4.3 The *workbook* Method

The second way to customize the names of individual layers on a map template is to use a workbook. The *workbook* method allows you to save the relevant map information from a template onto a .csv file. Within this .csv file, you can customize the layer names, and save your configuration for later use. When reading template information from a .csv file, the `mapmakers` package does not need to

re-open and read the reference templates on AGOL, so this method is faster than using *from_obj* for repeat build calls. If you change the settings on one of your template maps on AGOL or our internal portal, you will still need to run the *from_obj* method to read in the new change. Once these changes are exported to a workbook, you can load the map information directly from the workbook using the *from_workbook* method.

From *demo.py*:

```
def workbook():
    tmp = m.Templates.from_obj(templates, gis)
    logging.info("Templates: %s", len(tmp.template))
    tmp.workbook(gis, "examples/demo", True)
    logging.info("Workbook printed.")
```

We have wrapped all of the potential actions in *demo.py* within functions so that you can load the file into memory and experiment with different features of `mapmakers` by calling the functions defined in the demo from the interactive Python shell. The first line of the *workbook* function above is already familiar to you, and creates a *Templates* object from the dictionary of names and ids using the *from_obj* method. The logging statement prints the number of templates read, in this case four.

The next line calls the *workbook* method. Note that we call this method from the *Templates* object. Because the *Templates* object holds four templates, the workbook will include information from all four Webmaps. The first argument is the GIS connection (*GIS_CONN*), which we have aliased as *gis*. The second argument is the file path for the workbook. If you provide a valid file name with a proper file extension, such as "*my_file.csv*", then the workbook will receive this file name. If you provide a directory, as in the example above, then the file will be called *workbook.csv* and located in the directory provided. If the directory path is invalid, or the file name is invalid, the program will throw an error. If the .csv file is open, perhaps because you were inspecting or editing it in Excel, the program will be unable to overwrite the .csv and will throw an error (I trigger this particular error often).

Note that exporting to *csv* is a "side-effect" that does not modify the *Templates* object, and does not have a return value, so we do not assign a new variable name to the result when we call the *workbook* method.

4.3.1 Alternative Workbook Approaches

Using the *workbook* method on a *Templates* object creates a single workbook that contains information from all the Webmap templates in the project. The *workbook* method is also implemented for the *Template* class, with some minor differences. In the *Template* class, `mapmakers` assumes you have already called *load* on the template, so the GIS connection is not required, just the file path or directory. There are two optional arguments you can provide, *auto* and *named*. If *auto=true*, `mapmakers` will automatically assign names to each layer (e.g. "property_0", "property_1", "property_2"). If *named=true*, `mapmakers` will preserve the names you have set for each layer using the *with_names* method.

```
prop.workbook("examples/demo", named=true)
```

The file name defaults to the nickname used as a key for the template, followed by the *.csv* extension, so in the above case the output file has the path "*examples/demo/property.csv*".

The *Templates* class also provides the *workbooks* method, which is a wrapper around the *workbook* method from the *Template* class. The *workbooks* method iterates through each template in the *template* property field, and calls *workbook* on it, emitting a different .csv file for each template. The name of each template will correspond to the nickname used as the key for the template. In the case of this example, calling *workbooks* would produce four .csv files in the "*examples/demo*" directly, namely "*aerials.csv*", "*boundaries.csv*", "*plss.csv*" and "*property.csv*".

As with the *workbook* method for this class, *workbooks* takes a GIS connection as the first argument. It also takes an optional argument, *auto*, which defualts to *false*. If *auto=true*, the *workbooks* method will automatically assign names to layers.

```
tmp.workbooks(gis, auto=true)
```

These methods provide alternatives that may integrate better with your workflow. For most use cases, the *workbook* method in the *Templates* class is the recommended approach to create a workbook for your project.

4.3.2 Inspecting and Modifying Workbooks

The easiest way to inspect and modify a .csv file is to open it in Excel. The workbook includes seven columns: *name*, *title*, *group*, *id*, *layer_definition*, *popup_info*, *url* and *search*.

Table 4.1. *workbook.csv*

column	description
name	human-readable nickname for the layer
title	display title for the layer
group	human-readable nickname for the template
id	Internal program ID for the layer
layer_definition	JSON dictionary containing the layer definition
popup_info	JSON dictionary containing the popup info
url	URL path for the layer source
search	Searchable fields within the layer

If you are using *auto-naming*, inspecting the workbook can be an easy way to determine the specific name of a layer of interest, using the *title* column to identify the layer.

From *property.csv*:

	A	B	C	D	E	F
1	name	title	group	id	layer_defi	popup_i
2	property_0	Assessor Taxlots	property	e43c56ae-	{"drawing	{"popup
3	property_1	Building Footprints	property	e6e547e2-	null	{"popup
4	property_2	Subdivision Plats	property	0c5688a3-	null	{"popup
5	property_3	Addresses (City)	property	6d06e297-	{"drawing	{"popup

Figure 4.2. The *title* field can help to determine the layer associated with a row of data on the worksheet.

You can assign new nicknames to make the layers easier to reference and use in your code, and you can change the display title here as well. None of the other columns are safe to edit, with the possible exception of the *search* field. **Do not** modify the *group*, *id*, *layer_definition*, *popup_info* or *url* from within the .csv file. If you change the *group* or *id*, the program will not be able to associate the row data correctly with its parent template, and you will not be able to make references to it properly.

The *layer_definition* and *popup_info* are JSON dictionaries copied directly from AGOL, and should not be modified manually because the specifications for what fields and values they can include is very strict and the changes you submit may not be valid. If you want to change the layer definition or popup info associated with the template, then open the template webmap in AGOL or our internal portal, and modify the Webmap using their GUI interface normally. When you have saved the results, load the template into `mapmakers` again and export it to a workbook. You should then see the changes to the template reflected in updated JSON entries in the *layer_definition* and *popup_info* columns.

The *url* field specifies the source path for the layer data in the template. This is not necessarily the same as the url you want to use as the source for your new map. We record this field as a convenience, so that if you want to create a new layer from the template using the same url as the template map, you can. When you create a new map Item from a template, you can choose whether to provide a new url or use the one referenced in the template. For the city web viewer, we can publish the map

from services hosted on AGOL, or on our internal portal. For a given layer, we can use the same template regardless of whether we are pulling the service from our internal portal or AGOL.

The primary intended use of the workbook for editing is to change the nickname associated with the layer, set in the *name* field, for easier reference and use in your own code:

property.csv

	A	B	C	D	E	F
1	name	title	group	id	layer_defi	popup_i
2	taxlots	Assessor Taxlots	property	e43c56ae-	{"drawing	{"popup
3	footprints	Building Footprints	property	e6e547e2-	null	{"popup
4	subdivisions	Subdivision Plats	property	0c5688a3-	null	{"popup
5	addresses	Addresses (City)	property	6d06e297-	{"drawing	{"popup

Figure 4.3. Changing the layers names for use in your own code.

4.4 The *from_workbook* Method

If you create the *Templates* instance using the *from_obj* method, and do not make any changes to the names or titles in the workbook, then you are ready to begin creating a new map using the *Templates* object. If you have made changes to the names or titles using the workbook, then you will need to read these changes back into the `mapmakers` project using the *from_workbook* method. If you have stored template information previously into a workbook, you can also use the *from_workbook* method to read the workbook into your project instead of loading the template information from AGOL or the internal portal using the *from_obj* method, for a slight speed boost in loading time.

The *from_workbook* method takes the path to the workbook as an argument. If the file path does not exist, the program will throw an error. In our *demo.py* examples, we have defined a *load* function as follows:

```
def load():
    return m.Templates.from_workbook("examples/demo/workbook.csv")
```

We can create a new *Templates* instance by assigning a variable name to the output of the function like so:

```
tmp = load()
```

To provide flexibility, we have also implemented the *from_workbook* method for the *Template* class. The different between the two methods is their return types, which matches the class it is called from: *Templates.from_workbook* returns an instance of the *Templates* class, and *Template.from_workbook* returns an instance of the *Template* class. If you use the *workbooks* method to make a separate .csv file for each template, then use *Template.from_workbook* to read each workbook back into a *Template* in your project.

If you accidentally read a workbook with multiple templates into a *Template* class, the operation will still succeed and return a valid *Template* object. You can still select an individual layer by passing the key name into the *items* property, but you will not be able to reference different template key names, because all the information has been merged into a single template. It is not recommended to use this approach, instead use *Templates.from_workbook* if reading a workbook containing multiple templates, and use *Template.from_workbook* for a workbook containing a single template.

Here is a breakdown of the different template classes in `mapmakers`:

Table 4.2. Template Classes in *mapmakers*

Class	Description	*from_obj*	*work-book*	*work-books*	*from_work-book*	*with_-names*
Templates	Multiple reference templates	?	?	?	?	X
Template	A single reference template	X	?	X	?	?
TemplateItem	A single layer from a template webmap	X	X	X	X	X

The package includes multiple ways to create and load templates in order to accommodate different preferences in workflow. When I began, I thought that using a single workbook would be unwieldy because of the large number of layers in use by the city. I also thought that assigning memorable nicknames to individual layers would be an important part of my personal workflow. Now that I am able to experiment with using the package, I find that I almost never use individual template workbooks or the *with_names* method. I prefer loading all the template information into a single workbook using *Templates.from_workbook*, and then looking up the auto-generated layer name in the .csv file when I need to pull out an individual layer. We have tried to provide some flexibility, so you can use whatever approach that works best for you.

Chapter 5

Making Maps

As discussed in Introduction, a published service is often not sufficient for producing a quality web map. Differences in how symbology gets rendered in a web environment, as well as how popups and labels are displayed, can require "server-side" configuration. Perhaps you are using a service published by another agency, and want to alter the symbology or labeling for your site. Generally, we recommend making a template for each service that you want to use. Use the template to configure the web map experience, using the GUI provided in the ESRI Map Viewer. Each service published by the city should have a reference template showing how to render each layer. To build a new web map, we first pull in these reference templates, and then compose our new map from the pieces present in the templates.

Here we present some examples of how to use the `mapmakers` package to prepare a new map from a set of templates.

5.1 From Template to Map

The most simple case is creating a map from an existing template. In this instance we want to make a new map with exactly the same layers and configuration as the original template. As in the previous chapter, we will be working from the *demo.py* project in the *examples* folder. Here we define a function called *property*:

```
def property(template):
    return template.template["property"].into_items().group("Property")
```

This functions takes an argument called *template*, which must be an instance of the *Templates* class. As covered in Template Creation and Use, we can create this template either from a workbook using *Templates.from_workbook* or directly from the item IDs using *Templates.from_obj*.

The syntax ".template['property']" selects the template named "property" from the *Templates* instance, so the resulting object is of type *Template*. The *into_items* method converts the *Template* object into the `mapmakers` *Items* class. The *Items* class has a property field called *items* that holds a list with elements of class *Item*, the `mapmakers` class for map items. Each *Item* contains information related to a particular layer.

The *group* method takes an *Item* or *Items* and places them into a map group. The method takes a single argument specifying the display name of the group, in this case "Property". The line as a whole takes the layers in the "property" template, converts them into map items, and places these layers into a group called "Property", preserving the original order of the layers.

We will use the *Map* class to build a web map from the *Group* object. Instantiating the *Map* class requires three arguments. The first argument is the item ID of the target web map. The *arcgis* Python API does not provide a function for creating a new web map, so we must pass in an existing Item ID. We have created a web map called "test_demo_example" that we will reference for this example.

```
demo_id = "6b81b63a99e74192ac00c3a53b88ca27"
```

For the city web viewer, we use six different web maps. First, we have a map for test builds. We have a different version of the web viewer for public and staff. We also have a separate map that provides an editable version of various layers. Finally, we have a backup map for both the staff and public versions, so that we can build a new map and swap it into the viewer app without experiencing down time. In the *grants_pass* folder of the *examples* directory, the project *web_viewer.py* includes an *Enum* listing the different target Item IDs.

```
class Target(Enum):
    TEST = "2bc59679ad4040f4b7eb9ebec358152b"
    STAFF = "2c76b256e6954677802813291d22a2b9"
    STAFF1 = "199ca9a74d824a2287895cd736394138"
    PUBLIC = "8b104a6cfc774ba29cd873edf2bbef73"
    PUBLIC1 = "df37a2bec8554462a4e33d02bea239dc"
    EDITOR = "54738cddf51648ad99ba0ec5dfff2625"
```

We can then refer to a specific target map, referencing a particular variant's value.

```
mp = m.Map(Target.TEST.value, property(tmp), gis)
```

Within the *demo.py* project, we can use the *demo_id*:

```
mp = m.Map(demo_id, property(tmp), gis)
```

The second argument required to create a new instance of the *Map* class is the layer data for map, in this case the output of the *property* function. The *Map* class is flexible enough to convert objects of class *Item*, *Items*, *Layer*, *Layers* and *Group* into a *Map* object, as well as lists containing these classes. We will provide some examples below.

The final argument required to create an instance of the *Map* class is an authenticated GIS connection. We have used *gis* to alias the *GIS_CONN* connection created by our login script.

The *Map* class includes a *build* method. The *build* method will clear the target web map of any existing layers and then populate the map with the layer information stored in the *Map* object. Executing the *build* method can take a moment, and will complete silently. Add a logging message after the *build* call if you want to confirm completion.

```
mp.build()
logging.info("Target map updated.")
```

Making a new map exactly like the template map may not sound like a practical use case, but keep in mind that this methodology applies to groups within a map as well. If you are only updating a couple layers on a large map like the web viewer, and the majority of groups have not changed, then you can build these group layers directly from their templates with minimal effort.

5.2 Nesting A Group Layer

Group layers are an important tool for organizing layers into sensible categories for easier navigation. One of my original motivations for writing this package came from my frustration at organizing the layers of an ArcGIS Pro project into groups, only to have those group categories disappear when published as a service. The initial "killer feature" of the `mapmakers` package was the ability organize layers into groups programmatically using code, instead of manually through the ESRI Map Viewer GUI.

The *boundaries* function in *demo.py* illustrates the method for nesting a group layer within another group:

demo.py

```
def boundaries(template):
    plss = template.template["plss"].into_items().group("PLSS")
    boundaries = template.template["boundaries"].into_items().layers()
```

```
    logging.info("Appending PLSS")
    boundaries = boundaries.append(plss).group("Boundaries")
    return boundaries
```

First we create a group layer called "PLSS" from the template called "plss". As with the "property" example, we convert the *Template* into an *Items* object using the *into_items* method, and then enclose it in a group layer using the *group* method. The second line of the function creates a variable called *boundaries* by accessing the template "boundaries" and converting it into an *Items* object with the *into_items* method. Instead of placing the results immediately into a group, we want to add the PLSS group to the list of layers included in the "Boundaries" group, so we call the *layers* method. The *layers* method converts members of the *Items* class into the *Layers* class.

Why do we need to covert *boundaries* from the *Items* class into the *Layers* class? The *Item* class provides fields and methods for modifying aspects of the target layer, like the title or layer visibility, and serves as a recipe for how to create the map layer. The *Layer* class converts these instructions into a JSON dictionary that describes the layer details in a format that matches the ESRI specification for web maps. When we create a group layer, first we must convert all the items within the group into *Layer* objects, and then embed this information inside the definition of the group layer. The *Items* class can only contain references to objects of the *Item* class, but the *Layers* class can contain references to either a *Layer* class or a *Group*, so to create a nested group we first covert the *Items* into *Layers*, and then append the *Group* to the list of layers.

We have implemented the *append* and *extend* methods for the *Layers* class, to make it easier to add classes of different type to a list of layers. When calling *append* or *extend* on the layers class directly, the `mapmakers` package will keep the search fields and layer data synchronized when inserting new layers. If you were to append a layer by directly accessing the *layers* property in the *Layers* class, then you would have to remember to also grab the corresponding search terms in the *search* property and append that as well. It is safer and easier to call *append* or *extend* from the *Layers* class when performing operations like nesting a group. In the example above we append the "PLSS" group to the layers in *boundaries*, and then enclose these layers in a new group called "Boundaries".

There is a special case to cover when combining *Group* and *Layer* classes. When working with a *Layers* object, you can use *append* or *extend* to add additional layers to the object, but what if the first layer (the bottom of the render stack) needs to be a group? If you try to *append* or *extend* additional layers to a *Group* object, you will get an error.

```
# won't work because plss is type *Group*
plss = plss.append(boundaries).group("Boundaries")
```

In order to make this work, you must convert the *plss* object into the *Layers* class using the *into_layer* method:

```
# works because plss is type *Layers*
plss = plss.into_layer()
plss = plss.append(boundaries).group("Boundaries")
```

5.3 Raster Layers

Raster layers require special treatment because the JSON representation of the layer used by ESRI has distinct differences from the representation of a vector layer. In order to format these layers correctly, you must convert them into the *Layers* class using the *rasters* method. To illustrate this, let us take a look at the *aerials* function in *demo.py*:

```
def aerials(template):
    imagery = template.template["aerials"].into_items().rasters()
    imagery = imagery.group("Aerials")
    return imagery
```

Once you have transformed raster data from the *Items* class into the *Layers* class using the *rasters*

method, then you can treat it like any other group or layer data. If you have only a single layer of raster data, use the *raster* method to transform the *Item* object into a *Layer* object:

```
imagery = template.template["aerials"].items["aerials_0"].into_item().raster()
```

5.4 Cherry-Picking Layers

This section explains how to compose individual layers into a web map. Use the techniques in this section to select a single layer or a subset of layers from a published service for use in your map. To illustrate the workflow, we have added the *cherry_pick* function to *demo.py*:

```
def cherry_pick(t: m.Templates):
    # cherry pick an item using the item name
    city_limits = t.template["boundaries"].items["boundaries_5"].into_item()
    prop = t.template["property"].into_items()
    # index into *items* to cherry pick multiple layers
    prop.items = prop.items[1:4]
    quarters = t.template["plss"].into_items()
    # the type of the *items* property is a list
    # to cherry pick a single item, pass it in a list
    quarters.items = [quarters.items[0]]
    lyrs = prop
    # use extend to add a list of items to the *items* field
    lyrs.items.extend(quarters.items)
    # use append to add a single item to the *items* field
    lyrs.items.append(city_limits)
    mp = m.Map(demo_id, lyrs, gis)
    mp.build()
    logging.info("Target map updated.")
```

The easiest way to select a single layer from a template is to use the item name associated with the layer. As discussed in Template Creation and Use, you can set the item names of layers in a template either using the *with_names* method, or by editing the *name* column in the template workbook. If you do not manually set item names, then the names will layers will automatically be set to the template name followed by an underscore and the layer index, beginning at zero. In the *cherry_pick* example, we used auto-naming for the *boundaries* template. Examining the workbook in "examples/demo/workbook.csv", we can see that the name for the City Limits layer is "boundaries_5". By inserting the item name in brackets after the property *items*, we can select that particular item from the template.

You can create a new *Items* object from any list where the elements are of class *Item*.

```
county_line = t.template["boundaries"].items["boundaries_0"].into_item()
gpid = t.template["boundaries"].items["boundaries_1"].into_item()
reserve = t.template["boundaries"].items["boundaries_2"].into_item()
wards = t.template["boundaries"].items["boundaries_3"].into_item()
ugb = t.template["boundaries"].items["boundaries_4"].into_item()
city_limits = t.template["boundaries"].items["boundaries_5"].into_item()
# create an *Items* object from a list of *Item* objects
boundaries = m.Items([county_line, gpid, reserve, wards, ugb, city_limits])
# results are identical to this
boundaries_1 = t.template["boundaries"].into_items()
assert boundaries == boundaries_1
```

Since the *items* property in the *Items* class holds a list of *Item* objects, when you slice into the list to create a subset, the result is an *Items* object that holds only the items you have subset.

```
subset = t.template["boundaries"].into_items()
# subset layer at index 0, 1 and 2
subset.items = subset.items[0:3]
county_line = t.template["boundaries"].items["boundaries_0"].into_item()
```

```
gpid = t.template["boundaries"].items["boundaries_1"].into_item()
reserve = t.template["boundaries"].items["boundaries_2"].into_item()
# identical to subset
subset_1 = m.Items([county_line, gpid, reserve])
assert subset == subset_1
```

If you want to pull a single layer out of an *Items* object, you will need to wrap it in a list, because the type of the *items* property is a list.

```
subset = t.template["boundaries"].into_items()
subset.items = [subset.items[0]]
# This is the same as calling into the item by name
county_line = t.template["boundaries"].items["boundaries_0"].into_item()
assert subset == county_line
```

5.5 Customizing Layers

The *Item* class allows you to customize the url, title, visibility or opacity of the resulting layer.

- To change the url source for the layer, set the *url* property to the new target path.
- To change the title, set the *title* property to the desired display title.
- To change the visibility of the layer, set the *visible* property to *True* to make the layer visible, and *False* to make it invisible by default. The `mapmakers` package sets the visiiblity of layers to *False* by default, so you will only need to use the *visible* property to set the visibility to *True*.
- To change the opacity of a layer, set the *opacity* property to a number between zero and one, where the proportion represents the percent of opacity in the resulting layer.

```
county_line = t.template["boundaries"].items["boundaries_0"].into_item()
# change url source to internal portal
county_line.url = "https://gisserver.grantspassoregon.gov/server/rest/services
/city_boundaries/MapServer/7"
county_line.title = "County Line"
# visible defaults to False in mapmakers
county_line.visible = True
# opacity defaults to 0.5 in mapmakers
county_line.opacity = 0.9
```

In the city web viewer, we use the *title* property to change the display title of the assessors taxlots layer from "Assessor Taxlots" to "Taxlots (County)" to emphasis that the data does not belong to the city. We also use the *visible* property to make the city limits and the urban growth boundary visible by default.

Now you are familiar with all of the classes and methods we use to build the city web viewer. You should have sufficient understanding to read and understand the build script at "*examples/grants_pass /web_viewer.py*". Congratulations! If you have questions or suggestions, feel free to contact us at gis@grantspassoregon.gov.

API Reference

This page contains auto-generated API reference documentation [1].

6.1 mapmakers

6.1.1 Submodules

mapmakers.map

Module Contents

Classes

Item	The Item class holds a template reference and configuration information for a map layer.
Items	The Items class holds a list of Item objects.
Layer	The Layer class holds map information formatted in the JSON specification for an ESRI web map.
Layers	The Layers class holds layer data *layers* and search information *search* for multiple layers of a map.
Group	The Group class holds the JSON representation of a web map group layer and its search fields.
Map	The Map class holds layer information and methods for building an ESRI web map.

class mapmakers.map.**Item** (*url: str, template=None, visible=False, title=None, opacity=None*)

The Item class holds a template reference and configuration information for a map layer.

property url

The *url* property is a text representation of the URL for an ArcGIS service.

property template

The *template* property is a TemplateItem associated with an Item.

property visible

The *visible* property sets the default visibility of the map Item.

property title

The *title* property sets the display title of the map Item.

property opacity

The *opacity* field sets the opacity of the map Item.

1 Created with sphinx-autoapi

static check_url_in (*path: str*) → bool

The *check_url_in* static method sends a "GET" request to *path*, returning *True* if the status code is 200 and *False* otherwise. Primarily used by calling *check_url*.

Parameters **path** (*str*) – String representation of the target URL.

Returns Boolean indicating whether a "GET" request returns a status code 200.

Return type bool

check_url ( ) → bool

The *check_url* method wraps the static method *check_url_in* to enable ergonomic checking of `Item` urls.

Returns Boolean indicating whether a "GET" request returns a status code 200.

Return type bool

layer ( )

The *layer* method converts feature class data in a map `Item` into a map `Layer`.

Returns `Layer` instance constructed from self.

Return type Layer

group (*name: str*)

The *group* method converts a map `Item` into a map `Group` with name *name*.

Parameters **name** (*str*) – Display name of the `Group` layer.

Returns `Group` instance constructed from self.

Return type Group

raster ( )

The *raster* method converts raster data in a map `Item` into a map `Layer`.

Returns `Layer` instance constructed from self.

Return type Layer

vector_tile ( )

class `mapmakers.map.`**Items** (*items: list[Item]*)

The `Items` class holds a list of `Item` objects.

property items

The *items* property holds a list of `Item` objects.

static from_names (*urls: list[str], names: list[str], template: mapmakers.template.Template*)

The *from_names* method creates an `Items` instance from a list of urls *urls* and a list of names *names*. Iterates through the urls in *urls* and names in *names*, assigning each to the next layer in the template *template*. The number of urls and names must match the number of layers in the template.

Parameters

- **urls** (*list[str]*) – A list of urls for the target sources of each layer.
- **names** (*list[str]*) – A list of names for each map layer.
- **template** (*mapmakers.template.Template*) – A `Template` object created from a template web map.

Returns An `Items` object with *urls* and *names* mapped to layers in *template*.

Return type Items

static from_template (*urls: list[str], template: mapmakers.template.Template*)

The *from_template* method creates an `Items` object from a list of urls *urls* and a web map *template*. The number of urls and template items must be equal.

Parameters
- **urls** (*list[str]*) – A list of url target sources for the map layers in the template.
- **template** (*mapmakers.template.Template*) – A `Template` object containing layer data for the web map.

Returns An `Items` object with *urls* mapped to layers in *template*.

Return type Items

group (*name: str*)

The *group* method coverts an `Items` object into a map `Group` with name *name*.

Parameters **name** (*str*) – Display name of the `Group` layer.

Returns `Group` instance constructed from self.

Return type Group

layers ( )

The *layers* method converts the *items* in self into a `Layers` object.

Returns `Layers` instance constructed from self.

Return type Layers

rasters ( )

The *rasters* method converts the *items* in self into a `Layers` object.

Returns `Layers` instance constructed from self.

Return type Layers

vector_tiles ( )

append (*item: Item*)

The *append* method appends the `Item` *item* to the list of `Item` objects in the property *items*. Wraps the *append* method for a list.

Parameters **item** (*Item*) – An `Item` object to append to the list of *items* in self.

Returns Modifies and returns self.

Return type Items

extend (*items: list[Item]*)

The *extend* method appends the `Item` objects in *items* to the *items* property of self. Wraps the *extend* method for a list.

Parameters **items** (*list[Item]*) – A list of `Item` objects.

Returns Modifies and returns self.

Return type Items

insert (*idx: int, item: Item*)

The *insert* method inserts the `Item` object in *item* to the *items* property of self. Wraps the *insert* method for a list.

Parameters **items** – A list of `Item` objects.

Returns Modifies and returns self.

Return type Items

class mapmakers.map.**Layer** (*item: Item, raster=False*)

The Layer class holds map information formatted in the JSON specification for an ESRI web map.

property layer

The *layer* property holds a dictionary with the JSON representation for an ESRI web map.

property search

The *search* property holds a dictionary with search instructions for the Layer object.

static from_raster (*raster: Item*)

The *from_raster* method converts the input *raster* from an Item object to a Layer object.

Parameters **raster** (*Item*) – The raster Item to convert.

Returns A Layer object created from *raster*.

Return type Layer

static from_vector_tile (*tile: Item*)

static from_group (*group*)

The *from_group* method converts a Group object into a Layer object. Convert a Group object into a Layer object to nest one group inside another.

Parameters **group** (*Group*) – The Group object to be converted to a Layer.

Returns A Layer object containing *group*.

Return type Layer

group (*name: str*)

The *group* method converts a Layer object into a Group object.

Parameters **name** (*str*) – The display name of the Group.

Returns A Group object constructed from self.

Return type Group

class mapmakers.map.**Layers** (*contents: list, search: list*)

The Layers class holds layer data *layers* and search information *search* for multiple layers of a map.

property layers

The *layers* property holds a list of dictionaries with the JSON representation of each layer in Layers.

property search

The *search* property holds a list of dictionaries with the JSON representation of the search fields for each layer in Layers.

static from_items (*items: Items*)

The *from_items* method converts an Items object *items* into a Layers object.

Parameters **items** (*Items*) – The Items object to convert to a Layers object.

Returns A Layers object constructed from *items*.

Return type Layers

static from_group (*group*)

The *from_group* method converts a Group object *group* to a Layers object.

Parameters **group** (*Group*) – The Group object to convert into Layers.

Returns A Layers object constructed from *group*.

Return type Layers

static from_rasters (*rasters: Items*)
The *from_rasters* method converts an `Items` object containing raster data *rasters* into a `Layers` object.

Parameters **rasters** (*Items*) – The raster data to convert into `Layers`.

Returns A `Layers` object containing the *rasters* map information.

Return type Layers

static from_vector_tiles (*tiles: Items*)

group (*name: str*)
The *group* method converts a `Layers` object into a `Group` object.

Parameters **name** (*str*) – The display name of the `Group` object.

Returns A `Group` object constructed from self.

Return type Group

append (*item*)
The *append* method adds a layer to the *layers* property and any search information for the layer to the *search* property.

Parameters **item** (*Layer | Layers | Group | dict | list*) – The layer data to add to self.

Returns Modifies and returns self.

Return type Layers

extend (*items: list*)
The *extend* method adds the layer and search data from *items* to self. Wraps *Layers.append*.

Parameters **items** (*list[Layer | Layers | Group | dict]*) – A list of layers to add to self.

Returns Modifies and returns self.

Return type Layers

insert (*idx: int, item*)
The *append* method adds a layer to the *layers* property and any search information for the layer to the *search* property.

Parameters **item** (*Layer | Layers | Group | dict | list*) – The layer data to add to self.

Returns Modifies and returns self.

Return type Layers

class `mapmakers.map.`**Group** (*name: str, layers: list, search: list*)
The `Group` class holds the JSON representation of a web map group layer and its search fields.

property group
The *group* property holds the JSON representation of a web map group layer.

property search
The *search* property holds the JSON representation of the search fields for the group layer.

static from_items (*name: str, items: Items*)
The *from_items* method converts an `Items` object *items* into a `Group` object.

Parameters
- **name** (*str*) – The display title of the `Group` object.

- **items** (*Items*) – The `Items` object to convert into a `Group` object.

Returns A `Group` object constructed from *items*.

Return type Group

static from_item (*name: str, item: Item*)

The *from_item* method converts an `Item` object *item* into a `Group` object.

Parameters
- **name** (*str*) – The display title of the `Group` object.
- **item** (*Item*) – The `Item` object to convert into a `Group` object.

Returns A `Group` object constructed from *item*.

Return type Group

static from_layer (*name: str, layer: Layer*)

The *from_layer* method converts a `Layer` object *layer* into a `Group` object.

Parameters **layer** – The `Layer` object to convert into a `Group` object.

Returns A `Group` object constructed from *layer*.

Return type Group

static from_layers (*name: str, layers: Layers*)

The *from_layers* method converts a `Layers` object *layers* into a `Group` object.

Parameters **layers** – The `Layers` object to convert into a `Group` object.

Returns A `Group` object constructed from *layers*.

Return type Group

into_layer ( )

The *into_layer* method converts the `Group` object into a `Layers` object.

Returns A `Layers` object constructed from self.

Return type Layers

class `mapmakers.map.`**Map** (*id: str, layers, gis: arcgis.gis.GIS*)

The `Map` class holds layer information and methods for building an ESRI web map.

property handle

The *handle* property holds the ESRI Item ID of the target web map.

property layers

The *layers* property holds the layer information associated with the web map.

property search

The *search* property holds the search information associated with the web map.

clear ( )

Remove all layers from web map.

Returns Removes layers from web map at *handle* as side effect.

Return type NoneType

build ( )

The *build* method attempts to clear and update the target web map in the *handle* property with the layer information in the *layers* property and the search information in the *search* property.

Returns Modifies the target web map as a side effect.

Return type NoneType

mapmakers.template

Module Contents

Classes

TemplateItem	The TemplateItem class holds display information about a specific layer from an ArcGIS Online web map.
Template	The Template class holds map layer data from an ArcGIS Online web map.
Templates	The Templates class holds references to one or more Template objects.

class mapmakers.template.**TemplateItem** (*title: str*, *group_name: str*, *item_name: str | None*, *layer_definition*, *popup_info*, *url: str | None*, *search: list | None = []*)

The TemplateItem class holds display information about a specific layer from an ArcGIS Online web map.

property title

The *title* property holds a string representation of the display title for the map layer.

property group_name

The *group_name* property holds a human-readable nickname for the parent Template of the TemplateItem.

property item_name

The *item_name* property holds a human-readable nickname for the TemplateItem.

property layer_definition

The *layer_definition* property holds a JSON representation of the layer definition for the TemplateItem.

property popup_info

The *popup_info* property holds a JSON representation of the popup info for the TemplateItem.

property url

The *url* property holds a string representation of the url source for the map layer.

property search

The *search* property holds a list of field names within a map layer that should be searchable.

property id

The *id* property holds a unique ID used internally to track map layers.

static from_layer (*layer: dict*, *group_name: str*, *searches: dict*)

The *from_layer* method converts layer data from a web map into a TemplateItem object. This method is an internal library function called by *Template.read*.

Parameters

- **layer** (*dict*) – Layer data from the *layers* property of an *arcgis.mapping.WebMap* object.
- **group_name** (*str*) – The name of the template web map from which the layer originates.
- **searches** (*dict[str, list[str]]*) – A dictionary with layer IDs as keys and search fields as values, produced by the *Template.get_search* method.

Returns A TemplateItem containing the map data of *layer*.

Return type TemplateItem

static from_parts (*title: str, group_name: str, item_name: str, layer_definition: str, popup_info: str, url: str, search: list, id: uuid.UUID*)

The *from_parts* method is an internal library function that assembles the values in different columns of a template workbook into a `TemplateItem` object. Called by *Template.from_workbook*.

Parameters

- **title** (*str*) – The display title of the map layer.
- **group_name** (*str*) – The name of the parent template for the layer.
- **item_name** (*str*) – The human-readable nickname that serves as the key reference for the layer.
- **layer_definition** (*str*) – The JSON representation of the layer properties.
- **popup_info** (*str*) – The JSON representation of the popup definition for the layer.
- **url** (*str*) – The url source of the layer.
- **search** (*list[str]*) – A list of field names within the layer that should be searchable.
- **id** (*uuid.UUID*) – A unique ID used internally by the library to track layers.

Returns A `TemplateItem` containing the map data of the layer.

Return type TemplateItem

static from_raster (*raster: dict, group_name: str*)

The *from_raster* method is an internal library function that converts raster data from a web map into a `TemplateItem` object. Called by the *Template.read* method when the type of the layer is a raster.

Parameters

- **raster** (*dict*) – Raster data from the *layers* property of an *arcgis.mapping.WebMap* object.
- **group_name** (*str*) – The name of the template web map from which the layer originates.

Returns A `TemplateItem` containing the map data of the layer.

Return type TemplateItem

static from_vector_tile (*tile: dict, group_name: str*)

into_item ( )

The *into_item* method converts a `TemplateItem` into a map `Item`. The url of the `Item` is set to the same url as the `TemplateItem`.

Returns A map `Item` containing the layer data of the `TemplateItem`.

Return type mapmakers.Item

into_search (*id: str*) → list[dict]

The *into_search* method is an internal library function that takes the search fields listed in the *search* field of the `TemplateItem` and converts them into the JSON format recognized by ESRI web maps, so the resulting map can be searchable by the indicated field. Because a layer can have multiple searchable fields or none, the index of elements in the *search* property does not necessarily match the index of layers in the *layers* property of the *Layers* class. To ensure the search fields correspond to the correct map layer, we record the layer ID associated with a search field in the *get_search* method, and refer to the same layer ID when

injecting search into the JSON definition of the map.

Parameters **id** (*str*) – The layer ID of the layer to be made searchable.

Returns A list containing dictionary elements structured to enable search on an ESRI WebMap.

Return type list[dict]

class mapmakers.template.**Template** (*name: str, id: str*)

The Template class holds map layer data from an ArcGIS Online web map.

property name

The *name* property holds the human-readable Template name.

property id

The *id* property holds the Item ID of the template web map.

property items

The *items* property holds a dictionary with TemplateItem objects for values, and the item names of the TemplateItem objects as the keys.

check_template (*gis: arcgis.gis.GIS*) → bool

The *check_template* method attempts to fetch the item using its *id*, returning *True* if the type of content is an arcgis.gis.Item, *False* otherwise. Primarily accessed by *check_template*.

Parameters **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.

Returns Boolean indicating whether the item is of type arcgis.gis.Item.

Return type bool

with_names (*names: list[str]*)

The *with_names* method changes the *item_name* property of the TemplateItem objects within the *items* field to match the names provided in *names*. Fails if the number of names in *names* is not the same as the number of TemplateItem objects in *items*.

Parameters **names** (*list[str]*) – A list of names for items in the Template.

Returns A modified Template with the *item_name* of each TemplateItem set to one of the names in *names*.

Return type Template

read (*layers: list[dict], searches: dict, items=[]*) → list

The *read* method reads map layer information from the *layers* property of an arcgis.mapping.WebMap and returns a list of TemplateItem objects containing the map information. This method is used an in internal library function, called by *Template.load* and *Template.workook_parts*.

Parameters

- **layers** (*list[dict]*) – List returned by the *layers* property of the target arcgis.mapping.WebMap
- **searches** (*dict*) – Search dictionary returned by the *get_search* method.
- **items** (*list*) – A list that holds TemplateItem objects created during the operation.

Returns A list of TemplateItem objects holding the map information from layers in the *Template*.

Return type list[TemplateItem]

load (*gis: arcgis.gis.GIS*)

The *load* method accesses a template web map and reads the layer data into a Template object.

Parameters **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.

Returns A `Template` object containing layer data from the target web map.

Return type Template

static get_search (*item: arcgis.gis.Item*)

The *get_search* method reads the search fields from a web map and returns a dictionary with layer ids as keys and search fields as values. An internal library function called by *Template.load* and *Template.workbook_parts*.

Parameters **item** (*arcgis.gis.Item*) – An ArcGIS Item ID pointing to a non-empty web map.

Returns A dictionary with layer IDs as keys and search fields as values, containing the searchable fields within the provided web map.

Return type dict[str, list(str)]

workbook (*path*, *auto=False*, *named=False*)

The *workbook* method exports data from a `Template` object to a .csv workbook. Called by *Templates.workbooks*.

Parameters

- **path** (*str*) – The directory or file path destination for the workbook.
- **auto** – Assigns names to map layers automatically if true.
- **named** (*bool*) – Indicates that names have been assigned to map layers using the *with_names* method, and should be preserved.

Returns Prints a .csv workbook to the target *path* as a side effect.

Return type NoneType

workbook_parts (*gis: arcgis.gis.GIS*, *names: list[str]*, *title: list[str]*, *group: list[str]*, *id: list[uuid.UUID]*, *layer_def: list[dict]*, *popup_info: list[dict]*, *url: list[str]*, *search: list[dict]*)

The *workbook_parts* method is an internal library function that reads layer data from the target web map and appends the different types of map data to the appropriate list variable passed in as an argument. Called by *Template.workbook*.

Parameters

- **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.
- **names** (*list[str]*) – A list holding data on layer names.
- **title** (*list[str]*) – A list holding the display title of each layer.
- **group** (*list[str]*) – A list holding the group name of each layer.
- **id** (*list[UUID]*) – A list holding the internal unique ID of each layer.
- **layer_def** (*list[dict]*) – A list holding the layer definition of each layer.
- **popup_info** (*list[dict]*) – A list holding the popup info of each layer.
- **url** (*list[str]*) – A list holding the url source of each layer.
- **search** (*list[dict]*) – A list holding the search definition for each layer.

static from_workbook (*path: str*)

The *from_workbook* method loads map data from a .csv workbook at file location *path* into a `Template` object.

Parameters **path** (*str*) – The file path location of the workbook.

Returns A `Template` object containing the layer data in the workbook.

Return type Template

into_items ( )

The *into_items* method converts a `Template` object into an *Items* object.

Returns An `Items` object containing the layer data from the `Template`.

Return type mapmakers.Items

class `mapmakers.template.`**Templates**

The `Templates` class holds references to one or more `Template` objects.

property template

The *template* property holds a dictionary containing `Template` objects as values and using the `Template` names as keys.

add (*template: Template*)

A wrapper around the *update* method for the dictionary in the *template* property. The *add* method takes a *template* as an argument, and updates the *template* property to contain the *template* as a value, using the *template* name as a key.

Parameters **template** (*Template*) – The target `Template` object to add to the `Templates` object.

Returns Modifies the `Templates` object in place.

Return type NoneType

static from_obj (*templates: dict[str, str], gis: arcgis.gis.GIS*)

The *from_obj* method converts template information passed in a dictionary into a `Templates` class object.

Parameters **templates** (*dict[str, str]*) – A dictionary with template names as keys and arcgis.mapping.WebMap Item IDs as values.

Returns A `Templates` object containing layer data from the web maps at the Item IDs stored in *templates*.

Return type Templates

workbooks (*dir: str, auto=False*)

For each `Template` in the *template* property, the *workbooks* method prints a .csv workbook to the directory at *dir* containing the layer data of the template web map.

Parameters

- **dir** (*str*) – The target directory path in which to print the .csv workbooks.
- **auto** (*bool*) – Assigns names to map layers automatically if true.

Returns Writes one or more .csv workbooks to target directory *dir* as a side effect.

Return type NoneType

workbook (*gis: arcgis.gis.GIS, dir: str, auto=False*)

The *workbook* method prints a .csv workbook containing layer data from all of the `Template` objects in the *template* property. If *dir* points to a directory, the workbook will be placed in the directory under the name "workbook.csv". If *dir* provides a file name ending with a ".csv" extension, the workbook will be assigned the given file name.

Parameters

- **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.
- **dir** (*str*) – The path and (optionally) file name at which to print the .csv workbook.
- **auto** – Assigns names to map layers automatically if true.

Returns Writes a .csv workbook to the target directory *dir* as a side effect.

Return type NoneType

static from_workbook (*path: str*)

The *from_workbook* method loads the contents of the .csv workbook at *path* into a `Templates` object.

Parameters **path** (*str*) – The file path to the location of the .csv workbook.

Returns A `Templates` object containing map layer data from the workbook.

Return type Templates

into_items ( )

The *into_items* method converts a `Templates` object into an *Items* object. Iterates through the `Template` objects in the *template* property and calls *Template.into_items* on each object.

Returns An `Items` object containing the layer data from the `Templates`.

Return type mapmakers.Items

mapmakers.utils

Module Contents

Functions

create_layer_id(→ str)	Generate random ids for layers. Copied verbatim from https://community.esri.com/t5/arcgis-api-for-python-questions/python-api-add-group-layer-to-webmap/td-p/1112126.
expand_urls(→ list[str])	Generate list of urls over range index given a service stub.

mapmakers.utils.**create_layer_id** (*layerIndex: int*) → str

Generate random ids for layers. Copied verbatim from https://community.esri.com/t5/arcgis-api-for-python-questions/python-api-add-group-layer-to-webmap/td-p/1112126.
To build a web map from a published service, we generate feature layers pointed to each service. Each feature layer requires a unique layer id, produced by this function.

Parameters **layerIndex** (*int*) – Layer index number.

Returns A randomized string to serve as a unique id.

Return type str

mapmakers.utils.**expand_urls** (*stub: str, rng: range | list[int]*) → list[str]

Generate list of urls over range index given a service stub.

Parameters

- **stub** (*str*) – Url base string for map service.
- **rng** (*range*) – List of numbers generated by *range()* call.

Returns List of urls beginning in *stub* and ending in *rng* values.

Return type list[str]

6.1.2 Package Contents

Classes

Map	The `Map` class holds layer information and methods for building an ESRI web map.
Layer	The `Layer` class holds map information formatted in the JSON specification for an ESRI web map.

Layers	The Layers class holds layer data *layers* and search information *search* for multiple layers of a map.
Group	The Group class holds the JSON representation of a web map group layer and its search fields.
Item	The Item class holds a template reference and configuration information for a map layer.
Items	The Items class holds a list of Item objects.
TemplateItem	The TemplateItem class holds display information about a specific layer from an ArcGIS Online web map.
Template	The Template class holds map layer data from an ArcGIS Online web map.
Templates	The Templates class holds references to one or more Template objects.

Functions

create_layer_id(→ str)	Generate random ids for layers. Copied verbatim from https://community.esri.com/t5/arcgis-api-for-python-questions/python-api-add-group-layer-to-webmap/td-p/1112126.
expand_urls(→ list[str])	Generate list of urls over range index given a service stub.

class mapmakers.**Map** (*id: str, layers, gis: arcgis.gis.GIS*)

The Map class holds layer information and methods for building an ESRI web map.

property handle

The *handle* property holds the ESRI Item ID of the target web map.

property layers

The *layers* property holds the layer information associated with the web map.

property search

The *search* property holds the search information associated with the web map.

clear ( )

Remove all layers from web map.

Returns Removes layers from web map at *handle* as side effect.

Return type NoneType

build ( )

The *build* method attempts to clear and update the target web map in the *handle* property with the layer information in the *layers* property and the search information in the *search* property.

Returns Modifies the target web map as a side effect.

Return type NoneType

class mapmakers.**Layer** (*item: Item, raster=False*)

The Layer class holds map information formatted in the JSON specification for an ESRI web map.

property layer

The *layer* property holds a dictionary with the JSON representation for an ESRI web map.

property search

The *search* property holds a dictionary with search instructions for the Layer object.

static from_raster (*raster: Item*)
The *from_raster* method converts the input *raster* from an `Item` object to a `Layer` object.

Parameters **raster** (*Item*) – The raster `Item` to convert.

Returns A `Layer` object created from *raster*.

Return type Layer

static from_vector_tile (*tile: Item*)

static from_group (*group*)
The *from_group* method converts a `Group` object into a `Layer` object. Convert a `Group` object into a `Layer` object to nest one group inside another.

Parameters **group** (*Group*) – The `Group` object to be converted to a `Layer`.

Returns A `Layer` object containing *group*.

Return type Layer

group (*name: str*)
The *group* method converts a `Layer` object into a `Group` object.

Parameters **name** (*str*) – The display name of the `Group`.

Returns A `Group` object constructed from self.

Return type Group

class mapmakers.**Layers** (*contents: list, search: list*)
The `Layers` class holds layer data *layers* and search information *search* for multiple layers of a map.

property layers
The *layers* property holds a list of dictionaries with the JSON representation of each layer in `Layers`.

property search
The *search* property holds a list of dictionaries with the JSON representation of the search fields for each layer in `Layers`.

static from_items (*items: Items*)
The *from_items* method converts an `Items` object *items* into a `Layers` object.

Parameters **items** (*Items*) – The `Items` object to convert to a `Layers` object.

Returns A `Layers` object constructed from *items*.

Return type Layers

static from_group (*group*)
The *from_group* method converts a `Group` object *group* to a `Layers` object.

Parameters **group** (*Group*) – The `Group` object to convert into `Layers`.

Returns A `Layers` object constructed from *group*.

Return type Layers

static from_rasters (*rasters: Items*)
The *from_rasters* method converts an `Items` object containing raster data *rasters* into a `Layers` object.

Parameters **rasters** (*Items*) – The raster data to convert into `Layers`.

Returns A `Layers` object containing the *rasters* map information.

Return type Layers

static from_vector_tiles (*tiles: Items*)

group (*name: str*)
The *group* method converts a `Layers` object into a `Group` object.

Parameters **name** (*str*) – The display name of the `Group` object.

Returns A `Group` object constructed from self.

Return type Group

append (*item*)
The *append* method adds a layer to the *layers* property and any search information for the layer to the *search* property.

Parameters **item** (*Layer | Layers | Group | dict | list*) – The layer data to add to self.

Returns Modifies and returns self.

Return type Layers

extend (*items: list*)
The *extend* method adds the layer and search data from *items* to self. Wraps *Layers.append*.

Parameters **items** (*list[Layer | Layers | Group | dict]*) – A list of layers to add to self.

Returns Modifies and returns self.

Return type Layers

insert (*idx: int, item*)
The *append* method adds a layer to the *layers* property and any search information for the layer to the *search* property.

Parameters **item** (*Layer | Layers | Group | dict | list*) – The layer data to add to self.

Returns Modifies and returns self.

Return type Layers

class `mapmakers.`**Group** (*name: str, layers: list, search: list*)
The `Group` class holds the JSON representation of a web map group layer and its search fields.

property group
The *group* property holds the JSON representation of a web map group layer.

property search
The *search* property holds the JSON representation of the search fields for the group layer.

static from_items (*name: str, items: Items*)
The *from_items* method converts an `Items` object *items* into a `Group` object.

Parameters
- **name** (*str*) – The display title of the `Group` object.
- **items** (*Items*) – The `Items` object to convert into a `Group` object.

Returns A `Group` object constructed from *items*.

Return type Group

static from_item (*name: str, item: Item*)
The *from_item* method converts an `Item` object *item* into a `Group` object.

Parameters
- **name** (*str*) – The display title of the `Group` object.
- **item** (*Item*) – The `Item` object to convert into a `Group` object.

Returns A `Group` object constructed from *item*.

Return type Group

static from_layer (*name: str, layer: Layer*)
The *from_layer* method converts a `Layer` object *layer* into a `Group` object.

Parameters **layer** – The `Layer` object to convert into a `Group` object.

Returns A `Group` object constructed from *layer*.

Return type Group

static from_layers (*name: str, layers: Layers*)
The *from_layers* method converts a `Layers` object *layers* into a `Group` object.

Parameters **layers** – The `Layers` object to convert into a `Group` object.

Returns A `Group` object constructed from *layers*.

Return type Group

into_layer ( )
The *into_layer* method converts the `Group` object into a `Layers` object.

Returns A `Layers` object constructed from self.

Return type Layers

class mapmakers.**Item** (*url: str, template=None, visible=False, title=None, opacity=None*)
The `Item` class holds a template reference and configuration information for a map layer.

property url
The *url* property is a text representation of the URL for an ArcGIS service.

property template
The *template* property is a `TemplateItem` associated with an `Item`.

property visible
The *visible* property sets the default visibility of the map `Item`.

property title
The *title* property sets the display title of the map `Item`.

property opacity
The *opacity* field sets the opacity of the map `Item`.

static check_url_in (*path: str*) → bool
The *check_url_in* static method sends a "GET" request to *path*, returning *True* if the status code is 200 and *False* otherwise. Primarily used by calling *check_url*.

Parameters **path** (*str*) – String representation of the target URL.

Returns Boolean indicating whether a "GET" request returns a status code 200.

Return type bool

check_url ( ) → bool
The *check_url* method wraps the static method *check_url_in* to enable ergonomic checking of `Item` urls.

Returns Boolean indicating whether a "GET" request returns a status code 200.

Return type bool

layer ( )
The *layer* method converts feature class data in a map `Item` into a map `Layer`.

Returns `Layer` instance constructed from self.

Return type Layer

group (*name: str*)
The *group* method converts a map `Item` into a map `Group` with name *name*.

Parameters **name** (*str*) – Display name of the `Group` layer.

Returns `Group` instance constructed from self.

Return type Group

raster ( )
The *raster* method converts raster data in a map `Item` into a map `Layer`.

Returns `Layer` instance constructed from self.

Return type Layer

vector_tile ( )

class mapmakers.**Items** (*items: list[Item]*)
The `Items` class holds a list of `Item` objects.

property items
The *items* property holds a list of `Item` objects.

static from_names (*urls: list[str], names: list[str], template: mapmakers.template.Template*)
The *from_names* method creates an `Items` instance from a list of urls *urls* and a list of names *names*. Iterates through the urls in *urls* and names in *names*, assigning each to the next layer in the template *template*. The number of urls and names must match the number of layers in the template.

Parameters
- **urls** (*list[str]*) – A list of urls for the target sources of each layer.
- **names** (*list[str]*) – A list of names for each map layer.
- **template** (*mapmakers.template.Template*) – A `Template` object created from a template web map.

Returns An `Items` object with *urls* and *names* mapped to layers in *template*.

Return type Items

static from_template (*urls: list[str], template: mapmakers.template.Template*)
The *from_template* method creates an `Items` object from a list of urls *urls* and a web map *template*. The number of urls and template items must be equal.

Parameters
- **urls** (*list[str]*) – A list of url target sources for the map layers in the template.
- **template** (*mapmakers.template.Template*) – A `Template` object containing layer data for the web map.

Returns An `Items` object with *urls* mapped to layers in *template*.

Return type Items

group (*name: str*)
The *group* method coverts an `Items` object into a map `Group` with name *name*.

Parameters **name** (*str*) – Display name of the `Group` layer.

Returns `Group` instance constructed from self.

Return type Group

layers ( )

The *layers* method converts the *items* in self into a `Layers` object.

Returns `Layers` instance constructed from self.

Return type Layers

rasters ( )

The *rasters* method converts the *items* in self into a `Layers` object.

Returns `Layers` instance constructed from self.

Return type Layers

vector_tiles ( )

append (*item: Item*)

The *append* method appends the `Item` *item* to the list of `Item` objects in the property *items*. Wraps the *append* method for a list.

Parameters **item** (*Item*) – An `Item` object to append to the list of *items* in self.

Returns Modifies and returns self.

Return type Items

extend (*items: list[Item]*)

The *extend* method appends the `Item` objects in *items* to the *items* property of self. Wraps the *extend* method for a list.

Parameters **items** (*list[Item]*) – A list of `Item` objects.

Returns Modifies and returns self.

Return type Items

insert (*idx: int, item: Item*)

The *insert* method inserts the `Item` object in *item* to the *items* property of self. Wraps the *insert* method for a list.

Parameters **items** – A list of `Item` objects.

Returns Modifies and returns self.

Return type Items

class `mapmakers.`**TemplateItem** (*title: str, group_name: str, item_name: str | None, layer_definition, popup_info, url: str | None, search: list | None = []*)

The `TemplateItem` class holds display information about a specific layer from an ArcGIS Online web map.

property title

The *title* property holds a string representation of the display title for the map layer.

property group_name

The *group_name* property holds a human-readable nickname for the parent `Template` of the `TemplateItem`.

property item_name

The *item_name* property holds a human-readable nickname for the `TemplateItem`.

property layer_definition
The *layer_definition* property holds a JSON representation of the layer definition for the `TemplateItem`.

property popup_info
The *popup_info* property holds a JSON representation of the popup info for the `TemplateItem`.

property url
The *url* property holds a string representation of the url source for the map layer.

property search
The *search* property holds a list of field names within a map layer that should be searchable.

property id
The *id* property holds a unique ID used internally to track map layers.

static from_layer (*layer: dict*, *group_name: str*, *searches: dict*)
The *from_layer* method converts layer data from a web map into a `TemplateItem` object. This method is an internal library function called by *Template.read*.

Parameters
- **layer** (*dict*) – Layer data from the *layers* property of an *arcgis.mapping.WebMap* object.
- **group_name** (*str*) – The name of the template web map from which the layer originates.
- **searches** (*dict[str, list[str]]*) – A dictionary with layer IDs as keys and search fields as values, produced by the *Template.get_search* method.

Returns A `TemplateItem` containing the map data of *layer*.

Return type TemplateItem

static from_parts (*title: str*, *group_name: str*, *item_name: str*, *layer_definition: str*, *popup_info: str*, *url: str*, *search: list*, *id: uuid.UUID*)
The *from_parts* method is an internal library function that assembles the values in different columns of a template workbook into a `TemplateItem` object. Called by *Template.from_workbook*.

Parameters
- **title** (*str*) – The display title of the map layer.
- **group_name** (*str*) – The name of the parent template for the layer.
- **item_name** (*str*) – The human-readable nickname that serves as the key reference for the layer.
- **layer_definition** (*str*) – The JSON representation of the layer properties.
- **popup_info** (*str*) – The JSON representation of the popup definition for the layer.
- **url** (*str*) – The url source of the layer.
- **search** (*list[str]*) – A list of field names within the layer that should be searchable.
- **id** (*uuid.UUID*) – A unique ID used internally by the library to track layers.

Returns A `TemplateItem` containing the map data of the layer.

Return type TemplateItem

static from_raster (*raster: dict*, *group_name: str*)

The *from_raster* method is an internal library function that converts raster data from a web map into a `TemplateItem` object. Called by the *Template.read* method when the type of the layer is a raster.

Parameters
- **raster** (*dict*) – Raster data from the *layers* property of an *arcgis.mapping.WebMap* object.
- **group_name** (*str*) – The name of the template web map from which the layer originates.

Returns A `TemplateItem` containing the map data of the layer.

Return type TemplateItem

static from_vector_tile (*tile: dict*, *group_name: str*)

into_item ( )

The *into_item* method converts a `TemplateItem` into a map `Item`. The url of the `Item` is set to the same url as the `TemplateItem`.

Returns A map `Item` containing the layer data of the `TemplateItem`.

Return type mapmakers.Item

into_search (*id: str*) → list[dict]

The *into_search* method is an internal library function that takes the search fields listed in the *search* field of the `TemplateItem` and converts them into the JSON format recognized by ESRI web maps, so the resulting map can be searchable by the indicated field. Because a layer can have multiple searchable fields or none, the index of elements in the *search* property does not necessarily match the index of layers in the *layers* property of the *Layers* class. To ensure the search fields correspond to the correct map layer, we record the layer ID associated with a search field in the *get_search* method, and refer to the same layer ID when injecting search into the JSON definition of the map.

Parameters **id** (*str*) – The layer ID of the layer to be made searchable.

Returns A list containing dictionary elements structured to enable search on an ESRI WebMap.

Return type list[dict]

class `mapmakers.`**Template** (*name: str*, *id: str*)

The `Template` class holds map layer data from an ArcGIS Online web map.

property name

The *name* property holds the human-readable `Template` name.

property id

The *id* property holds the Item ID of the template web map.

property items

The *items* property holds a dictionary with `TemplateItem` objects for values, and the item names of the `TemplateItem` objects as the keys.

check_template (*gis: arcgis.gis.GIS*) → bool

The *check_template* method attempts to fetch the item using its *id*, returning *True* if the type of content is an arcgis.gis.Item, *False* otherwise. Primarily accessed by *check_template*.

Parameters **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.

Returns Boolean indicating whether the item is of type arcgis.gis.Item.

Return type bool

with_names (*names: list[str]*)

The *with_names* method changes the *item_name* property of the `TemplateItem` objects within the *items* field to match the names provided in *names*. Fails if the number of names in *names* is not the same as the number of `TemplateItem` objects in *items*.

Parameters **names** (*list[str]*) – A list of names for items in the `Template`.

Returns A modified `Template` with the *item_name* of each `TemplateItem` set to one of the names in *names*.

Return type Template

read (*layers: list[dict], searches: dict, items=[]*) → list

The *read* method reads map layer information from the *layers* property of an arcgis.mapping.WebMap and returns a list of `TemplateItem` objects containing the map information. This method is used an in internal library function, called by *Template.load* and *Template.workook_parts*.

Parameters

- **layers** (*list[dict]*) – List returned by the *layers* property of the target arcgis.mapping.WebMap
- **searches** (*dict*) – Search dictionary returned by the *get_search* method.
- **items** (*list*) – A list that holds `TemplateItem` objects created during the operation.

Returns A list of `TemplateItem` objects holding the map information from layers in the *Template*.

Return type list[TemplateItem]

load (*gis: arcgis.gis.GIS*)

The *load* method accesses a template web map and reads the layer data into a `Template` object.

Parameters **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.

Returns A `Template` object containing layer data from the target web map.

Return type Template

static **get_search** (*item: arcgis.gis.Item*)

The *get_search* method reads the search fields from a web map and returns a dictionary with layer ids as keys and search fields as values. An internal library function called by *Template.load* and *Template.workbook_parts*.

Parameters **item** (*arcgis.gis.Item*) – An ArcGIS Item ID pointing to a non-empty web map.

Returns A dictionary with layer IDs as keys and search fields as values, containing the searchable fields within the provided web map.

Return type dict[str, list(str)]

workbook (*path, auto=False, named=False*)

The *workbook* method exports data from a `Template` object to a .csv workbook. Called by *Templates.workbooks*.

Parameters

- **path** (*str*) – The directory or file path destination for the workbook.
- **auto** – Assigns names to map layers automatically if true.
- **named** (*bool*) – Indicates that names have been assigned to map layers using the *with_names* method, and should be preserved.

Returns Prints a .csv workbook to the target *path* as a side effect.

Return type NoneType

workbook_parts (*gis: arcgis.gis.GIS, names: list[str], title: list[str], group: list[str], id: list[uuid.UUID], layer_def: list[dict], popup_info: list[dict], url: list[str], search: list[dict]*)

The *workbook_parts* method is an internal library function that reads layer data from the target web map and appends the different types of map data to the appropriate list variable passed in as an argument. Called by *Template.workbook*.

Parameters

- **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.
- **names** (*list[str]*) – A list holding data on layer names.
- **title** (*list[str]*) – A list holding the display title of each layer.
- **group** (*list[str]*) – A list holding the group name of each layer.
- **id** (*list[UUID]*) – A list holding the internal unique ID of each layer.
- **layer_def** (*list[dict]*) – A list holding the layer definition of each layer.
- **popup_info** (*list[dict]*) – A list holding the popup info of each layer.
- **url** (*list[str]*) – A list holding the url source of each layer.
- **search** (*list[dict]*) – A list holding the search definition for each layer.

static **from_workbook** (*path: str*)

The *from_workbook* method loads map data from a .csv workbook at file location *path* into a `Template` object.

Parameters **path** (*str*) – The file path location of the workbook.

Returns A `Template` object containing the layer data in the workbook.

Return type Template

into_items ( )

The *into_items* method converts a `Template` object into an *Items* object.

Returns An `Items` object containing the layer data from the `Template`.

Return type mapmakers.Items

class `mapmakers.`**Templates**

The `Templates` class holds references to one or more `Template` objects.

property template

The *template* property holds a dictionary containing `Template` objects as values and using the `Template` names as keys.

add (*template: Template*)

A wrapper around the *update* method for the dictionary in the *template* property. The *add* method takes a *template* as an argument, and updates the *template* property to contain the *template* as a value, using the *template* name as a key.

Parameters **template** (*Template*) – The target `Template` object to add to the `Templates` object.

Returns Modifies the `Templates` object in place.

Return type NoneType

static **from_obj** (*templates: dict[str, str], gis: arcgis.gis.GIS*)

The *from_obj* method converts template information passed in a dictionary into a `Templates` class object.

Parameters **templates** (*dict[str, str]*) – A dictionary with template names as keys and arcgis.mapping.WebMap Item IDs as values.

Returns A `Templates` object containing layer data from the web maps at the Item IDs stored in *templates*.

Return type Templates

workbooks (*dir: str*, *auto=False*)

For each `Template` in the *template* property, the *workbooks* method prints a .csv workbook to the directory at *dir* containing the layer data of the template web map.

Parameters

- **dir** (*str*) – The target directory path in which to print the .csv workbooks.
- **auto** (*bool*) – Assigns names to map layers automatically if true.

Returns Writes one or more .csv workbooks to target directory *dir* as a side effect.

Return type NoneType

workbook (*gis: arcgis.gis.GIS*, *dir: str*, *auto=False*)

The *workbook* method prints a .csv workbook containing layer data from all of the `Template` objects in the *template* property. If *dir* points to a directory, the workbook will be placed in the directory under the name "workbook.csv". If *dir* provides a file name ending with a ".csv" extension, the workbook will be assigned the given file name.

Parameters

- **gis** (*arcgis.gis.GIS*) – An authenticated GIS connection.
- **dir** (*str*) – The path and (optionally) file name at which to print the .csv workbook.
- **auto** – Assigns names to map layers automatically if true.

Returns Writes a .csv workbook to the target directory *dir* as a side effect.

Return type NoneType

static **from_workbook** (*path: str*)

The *from_workbook* method loads the contents of the .csv workbook at *path* into a `Templates` object.

Parameters **path** (*str*) – The file path to the location of the .csv workbook.

Returns A `Templates` object containing map layer data from the workbook.

Return type Templates

into_items ( )

The *into_items* method converts a `Templates` object into an *Items* object. Iterates through the `Template` objects in the *template* property and calls *Template.into_items* on each object.

Returns An `Items` object containing the layer data from the `Templates`.

Return type mapmakers.Items

mapmakers.**create_layer_id** (*layerIndex: int*) → str

Generate random ids for layers. Copied verbatim from https://community.esri.com/t5/arcgis-api-for-python-questions/python-api-add-group-layer-to-webmap/td-p/1112126.

To build a web map from a published service, we generate feature layers pointed to each service. Each feature layer requires a unique layer id, produced by this function.

Parameters **layerIndex** (*int*) – Layer index number.

Returns A randomized string to serve as a unique id.

Return type str

mapmakers.**expand_urls** (*stub: str, rng: range | list[int]*) → list[str]

Generate list of urls over range index given a service stub.

Parameters
- **stub** (*str*) – Url base string for map service.
- **rng** (*range*) – List of numbers generated by *range()* call.

Returns List of urls beginning in *stub* and ending in *rng* values.

Return type list[str]

- genindex
- modindex
- search

m

A

B

C

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W