TAGGIT – A PYTHON LIBRARY FOR TAGGING FILES

Grant Savage | 2022/03/12

INTRODUCTION TO TAGGIT

Tags are powerful tools to enable semantic searching without compromising the name of a file. Currently, the ability to tag a file in the macOS is restricted to manual tagging via UI functionality. A user can add a single tag to multiple files at the same time, however the user is required to manually select the desired files and add tags one at a time. Manually adding tags is error prone and unsustainable when desired functionality requires applying many tags to many files. Taggit solves this problem by allowing users to add, manage, and search for tags directly in Python. At the time of writing this report, taggit functionality is currently limited to macOS, however it is designed with extensibility to other operating systems in mind.

FUNCTIONAL SPECIFICATIONS

Taggit is a general-use tool with many use cases for use by anyone leveraging Python. While building the project I focused on a few key users and use cases.

USER: DATA SCIENTIST

USE CASE: LOCAL MODEL DEVELOPMENT

A data scientist uses their local machine to develop models. Small-to-medium problems are trained locally, and large problems are built locally with sample data before they are trained on more powerful compute resources. During this model development various documents, python files and data are used and created. Currently, file management is done using directories and by appending prefix or suffix keys to file names. Directories can quickly become too deep to be and prefix and suffix keys can become unruly, both solutions ruin project structure readability. With taggit these problems can be solved by applying the necessary organizational and semantic search desirability via unintrusive tags. This allows the data scientists to keep clean, interpretable names and project structures and will allow the data scientist to quickly retrieve like-files from multiple locations via tag searching.

USE CASE: FEATURE ENGINEERING TAGGING

A Data scientist can use taggit to automatically tag feature engineering modules for specific projects with key features created in that module. Over the course of months or years the scientist will accumulate many feature engineering modules across a variety of projects. By tagging modules with the key feature names the data scientist will be able to quickly search for features from legacy models they want to reuse in new projects. Without these tags the data scientist would have to rely on memory to track down the correct module.

User: Student

Use Case: File Type Organization and Search

A student is in multiple classes and has a folder for each class they are in. The student then must make the imperfect decision to organize the subfolder structure by date/week or by the type of content represented in the file, (lecture notes, homework, presentation, recording, etc.). With taggit the student can tag their files with the information they chose not to represent in the folder naming convention. On my personal machine I organize class files by week and I use tags to label file content type. This allows me to quickly search my machine for all home-works rather than opening up the files week-by-week.

Component Specifications

The purpose of this Component Specification section is to describe the intended behavior and operation of the taggit package's public classes and methods. This document does not contain technical details on how this behavior is achieved, to view this information please refer to the relevant docstrings in the code.

Taggit.py Module

This is the only Python module in the project and contains the primary Tag class as well as all the public and private methods and functions. The taggit module is imported with a “import taggit” command.

taggit ParentTag class

Taggit’s ParentTag is a class object that holds the name and color of a tag. The user creates a ParentTag object by passing two parameters. A name string and a color string. ParentTags are also created using a pure string representation, and by passing a tuple containing string representations of the name and color (str,str). If a color is not provided when the class object is created, then the color is defaulted to None. When printed the ParentTag returns the string representation of the tag name. The ParentTag object representation (__ repr__) is explicit and looks like Tag("Name", "Color"). When checking for equality, two separate tag objects with the same name and color evaluate as equal.

taggit OSManager class

OSManager sets up the available OS managers via the register_manager() method. Once an OS manager is registered the OSManager can create an instance of that manager with the create method. This class is intended to be used to set the intended OS manager to provide the correct methods to edit tags in that OS.

OSManager method register_manager(key, manager)

The register_manager method simply adds a key, value pair {key:manager} to the _managers dictionary.

OSManager method create(key)

The create method looks up a registered tag manager and creates a class object of the manager if it exits.

TAGGIT MACOSTAGMANAGER CLASS

The MacOSTagManager class manages the implementations of Taggit for MacOS. This class contains all the macOS specific parameters as well as all the methods to interact with a masOS tag. The public methods are add_tag(), remove_tag(), remove_all_tags() and get_tags().

MACOSTAGMANAGER NESTED CLASS TAG

The Tag class is nested in the MacOSTagManager class and inherits from the ParentTag class. The Tag class is the macOS representation of a tag. The object holds the name and color of the tag as well as the color_code. The user creates the Tag object by passing two parameters A name string and a color string. The string (__ str__) representation looks like "tag\ncolor_code" so if the tag name = test and the color_code = 4 the string would be "test\n4".

MACOSTAGMANAGER METHOD ADD_TAG(TAG, FILE)

The add_tag method accepts a tag or list of tags and a file path. It applies the tags to the file without adding duplicate tags or removing existing tags.

MACOSTAGMANAGER METHOD REMOVE_TAG(TAG, FILE)

The remove_tag method accepts a tag or list of tags and a file path. It removes the tags from the file without generating an error if the passed tag(s) don't exist. It only removes the tags that are specified.

MACOSTAGMANAGER METHOD REMOVE_ALL_TAGS(FILE)

The remove_all_tags method accepts a file path and removes all existing tags from that file.

MACOSTAGMANAGER METHOD GET_TAGS(FILE)

The get_tags method accepts a file path and returns a list of tags that are on the file.

DESIGN DECISIONS

This section of the report defines the high-level design decisions of the project as well as the lower-level design style and standards that are followed and enforced.

HIGH-LEVEL DESIGN DECISIONS

My primary design consideration has been to focus on flexibility. What I mean by flexibility is that whenever possible, methods should “just work”. Users should be able to create a tag successfully by passing a variety of interpretable values (strings, lists, tuples). I do not want to enforce a single acceptable way of creating a tag. Tags should be able to be added without explicit creation of a tag object. This decision is made to allow for flexibility and ease of use.

Another import design consideration is allowing for future expansion to multiple OS environments. Nothing passed by the user should be required to be OS specific for functionality to work. This is achieved with a factory method pattern design. One drawback to this pattern is that when functionality is expanded to other operating systems the the developer building out the new OS tag manager class will be responsible for reusing method names implemented in the MacOSTagManager if they desire method calls to be consistent across platforms.

Cognitive load to the end user should be reduced whenever possible. Each method and function should clearly state its functionality via naming convention. Docstrings are non-negotiable. Class docstrings should provide information about the attributes, methods, and final class object. Function and method docstrings should provide a one-liner as well as parameter and return object information. This is an important design decision to make and enforce to increase ease of use.

STYLES AND STANDARDS

PEP 8 is followed and loosely enforced via Flake8. The Zen of Python (PEP 20) is channeled and often reflected upon.

Docstrings are used and required for all public and private classes, methods, and functions. Multi-line docstrings are expected, single line docstrings should only be used when functionality is remarkably obvious (developer's discretion). Comments should be kept to a minimum and only used when functionality is non-obvious. In-line comments preferred. The README contains high level information and examples for getting started with this package. Other documentation files are stored in the '/docs' directory.

Tests are stored in the '/tests' directory and test code-coverage is maintained at 100%. Pytest is used in tandem with Coverage via the pytest-cov plugin.

Taggit specifies build dependencies with pyproject.toml in accordance with PEP 518 and uses Poetry to help orchestrate the project and its dependencies.

COMPARISON TO MACOS-TAGS

Macos-tags is a library built for managing macOS tags in Python. The functionality of the library is quite similar to what I developed, and many of the methods have overlapping names and behavior. In fact, there are several methods in macos-tags that I still aspire to implement in taggit.

The macos-tags package does have less free methods for creating and applying tags. Taggit allows for adding multiple tags to a file via a list and can create tags with various string, tuple, and class objects. Macos-tags requires either a single Tag object or a single, perfect string representation of the tag for a user to add a tag to a file.

The larger limitations of the library become apparent when we consider allowing additional contributors or if we want to extend the library to multiple operating systems. The docstrings in mac-ostags are lacking detail, making it harder to contribute to the project. In my opinion, the heavy use of Python's typing library makes the code more difficult to read and adds a non-trivial cognitive load to developers trying to contribute. Finally, the primary reason taggit stands out as an improvement over mac-ostags, is that it is easily extensible to other platforms while mac-ostags was built only for macOS. Many methods and parameters in the mac-ostags are specific to the macOS environment. Taggit leverages a factory method design so that the OS is handled automatically as part of the design consideration.

Software Extensibility

The taggit package was designed to be extensible to include additional operating systems and I believe it does a good job at meeting this design requirement. The first steps in making a package extensible are to ensure it is well documented and well tested. All classes, methods, and functions in taggit have docstrings, and test coverage is 100%. The other major consideration made was the use of a factory method design pattern. This pattern is specifically designed to allow for extensibility to multiple systems and platforms, so it was a natural fit for this project. To extend the taggit project to another operating system the user does not need to modify any existing code. They simply need to create a new OS tag manager class, include the functionality in that class, and then in the __init__.py they can add a single line to register the new tag manager. If a developer wants to extend the functionality for an existing manager, it is intuitive where code should be added, and it is clear what functionality exists. There will be some overhead while the user learns the OS specific parameters and tag representations, but I do not believe this can be simplified.