Rahul Rajagopalan
6.170

Assignment 2 Specification

Introduction

The assignment gave the task of writing a Python application that reads a collection of JPEG images, and generates a website in the form of static HTML files that can be served by a web server. As specified, the code generates valid HTML and CSS that degrades gracefully in older browsers, and uses Jinja2 for templating and IPTCInfo for reading JPEG metadata for picture captions. When the program is run, it reads a manifest file and its command line arguments, and leaves its output files in an user-specified directory.

The top-level file is create_photo_gallery.py in the src/ directory. It calls functions from the rest of the application, which lives in the photogallery/ directory as a Python package.

Manifest

JPEG files' IPTC metadata can be of several formats, depending on how the files were originally produced. Therefore, my code can be configured to parse the metadata in the way the user desires. This configuration is done by editing a manifest file. This is a text file containing a JSON string. The object represented by the string should be a simple associative array mapping caption labels to the numbers that IPTCInfo needs to use to look up the metadata for that label.

Here's an example:

```
In [1]: from iptcinfo import IPTCInfo

In [2]: info = IPTCInfo('6170_sample_image_01.jpg')
('WARNING: problems with charset recognition', '"\\x1b"')

In [3]: info.data
Out[3]:
{20: [],
 25: ['Jerusalem'],
 55: '20110417',
 60: '041840-0400',
 62: '20110417',
 63: '041840-0400',
 80: 'Daniel Jackson',
 90: 'Jerusalem',
 101: 'Israel',
 118: [],
 120: 'Hike in Ein Kerem'}
```

When we construct an IPTCInfo object using 6170_sample_image_01.jpg as its input, its data attribute is a dictionary. The keys are numbers and the values are string data that the user presumably wants to display. Unfortunately, this dictionary does not provide semantic information about the caption data – a human would recognize that 'Jerusalem' is the location where the photo was taken, but there's no way for the program to know that. Also, the program can't know on its own which data to present – if it simply displayed it all, then 'Jerusalem' would appear twice, and the website would display the numbers associated with keys 55 and 60, without any context.

To solve this, a user can write a manifest file to clear things up. An example file would be:

```
{
“Photographer”: 80,
“City”: 90,
“Country”: 101,
“Description”: 120
}
```

Note that this must be a valid JSON object that Python's simplejson module can parse without errors. It should map human-readable caption data labels (as double-quoted strings) to the numerical keys that are used to look up the associated caption data using IPTCInfo.

This string tells the program to pull out the data with the numerical keys 80, 90, 101, and 120, and also provides human-readable labels for those data, which will be included in the caption to provide context.

I considered mutltiple options for formatting manifest data. I rejected using them as command-line arguments, as that would cause the resulting commands to invoke the application to become tediously large. I settled on JSON strings because they are relatively human-readable (in this case it is just a set of key-value pairs, surrounded by braces) and can be parsed by modules from the Python standard library.

The manifest file can be anywhere in the user's filesystem. Command-line arguments are used to make the program aware of its presence.

Command-Line Arguments

The program uses command-line arguments to locate the directory containing the input JPEG files, the directory in which the output should be copied (it will be created if it doesn't already exist), and the location of the manifest file. A typical command would be:

```
python create_photo_gallery.py -i /path/to/input/directory -o /path/to/output/directory -m /path/to/manifest/file
```

Users can invoke create_photo_gallery.py with the -h or –help flags to get a more detailed description of command-line flags.

One additional flag exists – the --no-prompt flag. This is used in generating titles for the HTML pages to display in the browser. The program will attempt to infer titles from the directory layout, but it can't read peoples' minds, so by default it will ask the user to accept the inferred name or provide a better one. This flag disables the prompting and could be useful if the create_photo_gallery.py is being invoked by a tool.

Output

All output files are left in the directory specified after the -o flag. The JPEGs are also copied over, so the original input files can be moved or removed without breaking the website. All the user has to do afterwards is copy the directory to a web server.

The program generates one HTML file for every entry in the input directory, including one for the top-level directory. Directory pages list all the photos inside the directory, as well as links to subdirectories. All the photos from the directory are presented on one page without pagination (I personally prefer scrolling vertically to clicking “Next” and “Prev” buttons and wating for the page to load).

Photos are displayed one image per row. Laying out multiple photos in a row isn't feasible for generalized input. First, if the photos have different dimensions, the row will have a “ragged”

appearance where the bottom lines of photos don't line up. Also, we can't predict in advance the sizes of browsers that will be used to see the website. A combination of large enough photos and long rows could force some users to scroll horizontally (typically less acceptable than scrolling vertically).

Each photo has its own detail page, which can be seen by clicking on the photo in the directory page. This detail page shows a single photo in a centered position along with its caption. A “Back” link is provided to return to the directory page.

The problem of how to lay out subdirectories is handled by displaying links to subdirectories in a directory's page. The website's link layout effectively mirrors the directory tree in which the photos were organized in the filesystem. This has the advantage of not forcing the user of the script to learn a new layout scheme; they can reuse their existing knowledge of heirarchical directories in a filesystem.

The output folder contains all the HTML, JPGs, and other files in a flat directory layout, but this is only an implementation detail used to simplify generation of anchor links – the website is presented to the user as if it were heirarchical.

Example

An example page, created using sample-jpegs-with-metadata.zip's JPEGs, can be found at http://web.mit.edu/rahulraj/www/6.170/assignment2/example_album/home-rahulraj-metadata_jpegs.html

(The file name for the HTML derives from the fact that I stored the JPEGs in /home/rahulraj/metadata_jpegs on my computer. Fully qualified names are used for the HTML files to avoid collisions).

I ran the script from the command line, using the same argument structure as described above. I then copied the output directory's contents into the www/ folder on my MIT account. I did not use the --no-prompt flag, and when prompted for a name for the top-level directory, inputted “Example Album”. My manifest file had the following contents:

```
{
 "Photographer": 80,
 "City": 90,
 "Country": 101,
 "Description": 120
 }
```