Bluebolt ATD Python Test

Parker Britt

Understanding

This challenge is meant as a facsimile of real ATD work, likely replicating the process of taking production assets and bulk formatting them for delivery using Python. This problem can be split into four parts, project setup, validation, component extraction, and outputting results.

While this may be a one off script, my goal is always to create clean maintainable code within given time constraints. Therefore I prioritized project structure, comments, and general good practices.

Setup

I first needed to set up the environment. This means first picking a python version. Without a target Python version in the spec I decided to go with the VFX reference platform's version 3.11.x for development and testing.

I then set up standard project directories and files including a pyproject and cli.py file. The project is set up as a python package, which can be installed using the python package manager, pip, and can then be executed with the `bb-asset-validation` command. Code segments are split into three separate files based on their purpose.

My initial instinct was to set up a reproducible environment for testing. Using a Docker container I could reproducibly read and manipulate project files. If this were a real project I assume the intent would be to use files from disk, rather than a python list or text file in which case a container would be ideal. However, since this is just a test I shortly abandoned this idea.

Validation

I start by checking the split path has the correct number of subdirectories. While the code is developed and tested only for Linux, as a simple cross compatibility step I split the path based on the OS specific path delimiter.

I then check that the path starts with the correct path prefix (`/shows` in most cases).

Finally I check the version number between the asset and the file number match.

Given the dangerous unpredictability of working with files in a real filesystem more testing would be useful for production, but for the test I kept it brief.

Component Extraction

My initial basic solution involved splitting the path by directory then splitting that into its base components, looking something like this:

```
path_components = {
    "project_name" : split_asset_path[0],
    "asset_type" : split_asset_path[2],
    "asset_name" : split_asset_path[3].split("_")[0],
    "task" : split_asset_path[4],
    "version" : split_asset_path[3].split("_")[1],
    "filename" : os.path.splitext(split_asset_path[6])[0].split("_v")[0],
    "file_extension" : os.path.splitext(split_asset_path[6])[1],
}
```

While this may be okay for the test, I wouldn't want to use it in production as there are several issues.

- It's rigidly mapped to the current directory schema, making future changes to the schema require updating this confusing structure.
- It would require excessive index range (and other error) checking to avoid unexpected exceptions.

Since writing maintanable code is always a priortiy for me, I decided on another solution. That being regex pattern matching. Not only does this simplify the solution above, but by using pattern matching future path schemas could be added by only changing a single string.

To identify the components in the schema I used named capturing groups, then I derive a dictionary of groups from the matches.

It's important to consider that some developers may not be familiar with regex pattern syntax. Future work could involve abstracting the regular expression into a more human readable format. For example the pattern:

```
/shows/(?P<project>\S*?)/\S*?/
```

could become

```
/shows/{project}/*/
```

meaning, the path should start with "/shows", capture the next directory under the name 'project', then the directory after that can contain anything without capturing.

With an abstracted pattern, updating the schema would be possible for even those without development experience. Patterns could be received as a command line argument without requiring any changes to the code.

Outputting results

The output path can be assembled using formating strings then printed. I used ansi escape codes to color the output, making it easy to distinguish errors at a glace, this is especially important if artists are using the script. Normally I would use a cross compatible library like colorama, but to avoid overcomplicating a simple task I just created an enum containing the codes I needed.

Frame Numbers

With the output schema :

```
"/shows/{project}/staging/delivery/assets/{asset_type}_{asset_name}/{task}_{version}/
{filename}{file_extension}"
```

I was unclear whether the frame number of exrs should be included in the filename or not. The given schema doesn't mention it, however leaving it out could lead to exr sequences with the same name overriding eachother.

eg.

```
foo.1001.exr
foo.1002.exr
```

would both evaluate to

```
foo.exr
```

Normally I might seek clarification on this point from my lead, but for the test including it seemed like the most logical decision. Therefore exrs will look like this.

```
"/shows/{project}/staging/delivery/assets/{asset_type}_{asset_name}/{task}_{version}/
{filename}.{frame_number}{file_extension}"
```

Installation & Usage

As previously mentioned the project is formatted as a regular python package, use whatever practice you normally would or choose one of the options below.

Execution

To run it directly you can use:

```
python -m src.bb_asset_validation.cli
```

Or using `uv` you can run:

```
uv run bb-asset-validation
```

Installation

For a proper installation you can use

```
pip install .
```

then the command

```
bb-asset-validation
```

Getting started

Use the `-h` or `--help` arguments to get started.

Then point it to one of the example files provided (or your own) using

```
bb-asset-validation --paths-file example-paths-mixed.txt
```

Please let me know if you have any additional questions at `parker@parkerbritt.com`