0) Imports
1) Configuration
    a) Contains all the adaptable information besides the model itself
        i) Paths to our data
            (1) Image folder
            (2) csv with labels
            (3) Path where metadata is saved to
        ii) Data to train on
            (1) Train cases, test cases, validation cases
        iii) Hyperparameters
            (1) Batch size
                (a) # images that go into the model as a group before the next backpropagation is run and the model params change
            (2) Learning rate
                (a) Controls step size in gradient descent
            (3) # epochs
                (a) # times we learn on all training data
        iv) Filtering parameters
            (1) Min and max image sizes
            (2) Stain
            (3) Resizing (if needed for bucketed adaptive pooling)
        v) Other Parameters
            (1) Force metadata rebuild
                (a) Only set to false if you have not changed any other parameters
                (b) Will remake metadata each time
            (2) Device
                (a) Controls what hardware (GPU vs. CPU etc) we use
                (b) Default GPU
            (3) Num_workers
                (a) Param for the data loader
                (b) If num_workers > 0, we load + preprocess data in parallel
                (c) Data is loaded + preprocessed WHILE the model is training (in the background)
                (d) Set = 4x the # GPU

2) Metadata
    a) Why do we need to build metadata?
        i) We need to filter out images that aren't in our training criteria (stain and size)
        ii) Because we do not have unique identifiers (id codes) for each image, we build a metadata data frame that finds the image size and stain for each image path

        iii) We can then filter based on this metadata

    b) Fast scan function
        i) Goal: Quickly and safely find all of the png files from google drive
        ii) Google Drive I/O (info transfer system) is buggy -> needs some retrying
    c) Metadata builder function
        i) Uses fast scan function to navigate through image folder and create the metadata on files that fit our filtering criteria (stain and size).
        ii) Stain included in png file name, height x width included in png metadata
        iii) Default is to force rebuilding of metadata each time we run it so that if we change any inputs (stain, image size, image path, images inside the folder update), the metadata will be safely up to date
            (1) This is slow. If you decide no inputs (even new images added to folder) have actually changed, you can set `force_rebuild_metadata_full to false`

3) Labels Map

- Loads in our class labels as a dictionary with {case:label} format
- A labels map is needed when labels and images aren't stored together

4) Define Transforms

- Why do we need a transform?
    - Get all the images in the right format (tensors), and right colors (normalized)
    - Please adjust the transformation functions as per the requirements of your specific model
- Why separate train and eval?
    - We add .RandomHorizontalFlip(p = 0.5) to augment image data for train, but we want original (non-augmented data for validation and training data which both use the eval_transform)
    - .RandomHorizontalFlip(p = 0.5) has 0.5 probability of flipping image horizontally
    - We also have other sample data augmentation inputs in the skeleton code
- to_Tensor
    - Converts PIL images to pytorch tensor w/ normalized [0,1] values for each pixel
    - Pytorch uses tensors and cannot use PIL images (Python imaging library)
        - Why load as a PIL first if we need to convert?
            - Simple and common bc PIL works with transforms and pytorch and it is easy to read images as PIL
- .normalize()
    - We normalize bc neural nets learn more easily with images that have standard normal distribution (mean = 0, variance = 1)
    - Why these specific values? (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        - ImageNet images are already normalized with these values, so we do that to match what this model expects (the model has learned to see colors like this)
        - Only use this normalization if you are working with networks trained on ImageNet data

5) Creating a Dataset Object for Train/Test/Val

- Why do we need a dataset class?
    - This is our funnel that brings images, metadata, and labels together and prepares them to be fed into the model
    - We have a unique dataset with images, metadata, and labels in 3 different places -> This helps bring them together neatly for the model
    - Defined so that the DataLoader can quickly and easily get image data
    - Applies the transformations
    - Keeps training, test, and val datasets separate

- What does the Dataset class do?
    - 1) Brings labels, metadata, and images together
    - 2) Tells length of dataset
    - 3) Gets individual images
        - We load the image from the path
            - Read as a PIL image
        - Run the transformation on that image
        - Return transformed image, label, and case ID altogether
    - 2 and 3 are designed to help the data loader

6) Dataloader and helper functions

- What does a data loader do?
    - Batching
        - Creates groups of size = batch_size that are all fed into the model at once (before each new backpropagation update)

There are 3 code chunks for 3 different helper functions in the skeleton code:

- BucketSampler: function behind our bucketing logic. It will group indices of similar-sized images together
- make_boundaries_for_buckets: will calculate and define the actual size boundaries for our buckets
- collate_fn_pad: collate functions are used by the DataLoader object to group individual samples into a batch. Our custom function also performs the padding operation within each batch

Bucketing logic: Because we have to pad images up to the size of the largest image within each batch, we want to make sure that images within the same batch are of similar sizes. To do this, we will first calculate the total_number_of_batches that will be used per epoch (total images/batch size). Then we will separate the images by width into total_number_of_batches number of bins (e.g. if total_number_of_batches = 40, then there will be 40 bins). Then each of these bins will be a batch and sent separately to be trained on — thus images within the same batch are of very similar sizes and the amount of padding needed is very negligible).

Note - be careful about the n_buckets input if you are using non-adaptive pooling layers in your model. The n_buckets parameter directly influences sizes of images being fed into the network, since the amount of padding varies based on distribution of image sizes in each bucket (and hence the resultant batches). If your model architecture has pooling layers with fixed kernel sizes, such as MaxPooling, you will have to be careful about what image sizes are fed into the network. This is not a problem with Adaptive Pooling since, as the name suggests, this layer adapts features such as kernel size in accordance to the desired output size.

7) Data Visualization and summary statistics
- So that we can see some example images and see our class balance in train, test, and validation
8) Model definition and training

Training Function
- What does it do?
    - 1) Picks the hardware (GPU if possible) and picks up from last checkpoint if it exists
        - Needs to move images to GPU if we have one bc PyTorch doesn't do it by default
    - 2) Training Loop
        - Loops through # epochs number of times
        - In each loop
            - Loops through all batches (created by data loader)
                - For each batch
                    - Forward pass -> loss calculation -> backprop (finds gradients) -> optimizer step (uses gradients to update params)
            - Calculates validation score
            - The best model and the model from every epoch are saved
    - 3) Visualizes training and validation performance

Define your model
- 5 Things must be specified
    - 1) Model object
    - 2) Unfreeze layers that you want to unfreeze
        - These are the layers for which params are actually updated
    - 3) Replace the classifier head if you want and modify feature extractor to include Adaptive Pooling
        - We do this if the model we pretrained on has more than 2 output classes (e.g. Imagenet has 1000 classes, we have 2 so we need to update the classifier head)
    - 4) Define the model optimizer and pass your trainable params
        - We tell the Adam optimizer to only update the trainable params
        - We give the optimizer the learning rate
        - Optimizers control how we navigate the loss function based on the gradients
    - 5) Define class weighting and update criterion
        - To correct for class imbalance
        - Criterion is a loss function

            - E.g. cross_entopy_loss

Train the model

- Calls the training function

9) Evaluation

Model evaluation

- Runs model on our test dataset and visualizes the performance