Imagine classification python project

#TODO: 0: Timing Code

Implement the start_time to measure total program runtime.

Coding within the *check_images.py*

Code to Edit within Project Workspace - Timing

- The comments in the program header indicated by #TODO: 0
- Add your name
- The date you started working on the project
- Check the timing code within the main() function indicated by #TODO: 0
- We have added all the code you need to time your program. Here we expect that you will test our timing code by adding different values of seconds in the sleep() function to check how the timing and formatting of time works within the code we have provided.

Expected Outcome

When completed this code will calculate the runtime of the program. Specifically, this code will measure how long each of the three algorithms will take to classify all the images in the pet_images folder.

Checking your code

Use the sleep() function to test that your timing code is working correctly. Test the following:

- Set different values of seconds in the sleep() function to check the timing and formatting of time.

Project Workspace - Timing

- The next concept will have your workspace to work on #TODO: 0
- Editing of *check_images.py* can be done within the Project Workspace - Timing

For additional information and help on #TODO: 0, please look at the information below:

Importing Time Module

Timing your program or a portion of your program's code, allows one to compare the *time* costs associated with using different algorithms to solve a problem. Additionally, timing your code allows one to know the *time* costs associated with running a program using given computing resources.

To time your code In python requires the import of the **time() function** from the **python time module**. To simulate our program running for a certain period of time, we are going to use the time module's **sleep() function**. It will pause the program execution for a set number of seconds.

Since we only need to use the time() and sleep() functions, we will only import these two functions and not the entire time module. Only importing the functions from the module that you need saves on memory (RAM) your program requires to execute.

Such an import would look like the following:

```
# Imports time() and sleep() functions from time module
from time import time, sleep
```

`#TODO: 1`: Command Line Arguments

Fill code in the get_input_args() function to create & retrieve the command line arguments

Code to Edit

This section will help you code the function **get_input_args** within ***get_input_args.py***. With this function you will use argparse to retrieve three command line arguments from the user. (Argparse makes it easy to write user-friendly command-line interfaces).

- Code for the function definition `def get_input_args():` as indicated by `#TODO: 1`within ***get_input_args.py***.

Expected Outcome

When completed this code will input the three command line arguments from the user.

Checking your code

The check_command_line_arguments function within *check_images.py* will check your code.

Test the following:

- Entering ***no*** command line arguments when you run ***check_image.py*** from the terminal window. This should result in the *default* values being printed.
- Entering in values of your choosing for the command line arguments when you run ***check_image.py*** from the terminal window. This should result in the values you entered being printed.

Project Workspace - Command Line Arguments

- The next concept will have your workspace to work on `#TODO: 1`
- Editing of *check_image.py* and *get_input_args.py* can be done within the Project Workspace - Command Line Arguments

For additional information and help on `#TODO: 1`, please look at the information below:

Purpose

The purpose of command line arguments is to provide a way for your programs to be more flexible by allowing external inputs (command line arguments) to be input into a program. The key is that these external arguments can change as to allow more flexibility in the program.

For example, imagine you wrote a program that simply counts the number of lines in a file and prints out that number to the screen. To allow the user to enter in *any* file without having to change the program, one would want to pass in the file location as a command line argument. In this way, the program could be used on *any* file since the value is passed as an external input at runtime.

Usage of Argparse:

We will be using the **argparse** module to input the following external inputs into our program **check_image.py**. We recommend writing the **get_input_args** function to get the command line arguments using **argparse**.

Below are the three external inputs your **check_image.py** program will need to retrieve from the user along with the suggested *default* values each should have.

- Folder that contains the pet images
- pet_images/
- The CNN model architecture to use
- *resnet*, *alexnet*, or *vgg* (pick one as the default). You will find them in **classifier.py**.
- The file that contains the list of valid dognames
- dognames.txt

The **get_input_args** function will need to create an argument parser object using **argparse.ArgumentParser** and then use the **add_argument method** to allow the users to enter in these three external inputs from above.

Below is an example of creating an argument parser object and then using **add_argument** to add an argument that's a path to a folder and a second argument that's an integer.

```
# Creates Argument Parser object named parser
parser = argparse.ArgumentParser()

# Argument 1: that's a path to a folder
parser.add_argument('--dir', type = str, default = 'pet_images/',
                    help = 'path to the folder of pet images')
```

Below you will find an explanation of the inputs into **add_argument**.

- Argument 1:
- --dir = The variable name of the argument (here it's *dir*)
- type = The type of the argument (here it's a string)
- default = The default value (here it's 'pet_images/')
- help = The text that will appear if the user types the program name and then -h or --help. This allows the user to understand the what's expected an argument's value

Accessing Argparse Arguments

To access the arguments passed into the program through your argparse object, you will need to use the **parse_args method**. The code below demonstrates how to access the arguments through the argparse extending the example above.

To begin, you will need to assign a variable to **parse_args** and then use that variable to access the arguments of your argparse object. If you are creating the argparse object within a function, you will need to *return* **parse_args** instead of assigning a variable to it. Also note that the variable **in_args** points to a collection of the command line arguments.

`#TODO: 2`: Creating Pet Image Labels

Fill the get_pet_labels() function to create pet image labels by creating a dictionary with key=filename and value=file label. (We will later use this to to check the accuracy of the classifier function)

Coding within the *check_images.py* and *get_pet_labels.py*

Code to Edit

This section will help you code the function **get_pet_labels** within ***get_pet_labels.py***. With this function you will be creating the labels for the pet images, using the *filenames* of the pet images in the *pet_images* folder. These images filenames represent the identity of the pet in the image. The pet image labels are considered to represent the "*truth*" about the classification of the image. Your function will return the Results Dictionary that will contain the pet image filenames and labels.

- Code for the function definition `def get_pet_labels():` as indicated by `#TODO: 2` within ***get_pet_labels.py***.
- Follow the comments and the docstring within ***get_pet_labels.py*** to implement **get_pet_labels**
- Code within the `main()` function within ***check_images.py*** indicated by `#TODO: 2`
- Replace *None* within the function call with *in_arg.dir* to specify the appropriate directory.

Expected Outcome

When completed this code will return a dictionary with the *pet image filename* as the *key* and a *List* that contains only the *pet image label* as the *value* for all 40 pet images in the *pet_image* folder.

Checking your code

The **check_creating_pet_image_labels** function within ***check_images.py*** will check your code by printing out the number of key-value pairs and the first 10 key-value pairs.
You will need to visually check that the results show:

- The dictionary containing 40 key-value pairs (e.g. dictionary length is 40).

- The pet image labels the following way:
- Lower case letters
- Single space separating each word
- Correct representation of the filenames (from the 10 key-value pairs)

Project Workspace - Pet Image Labels

- The next concept will have your workspace to work on `#TODO: 2`
- Editing of ***check_image.py*** and ***get_pet_labels.py*** can be done within the **Project Workspace - Pet Image Labels**

For additional information and help on `#TODO: 2`, please look at the information below:

How to read *Filenames* from a Folder of Files

The folder **pet_images/** in the lab workspace contains the 40 images you will be testing the classifier algorithms on. The filenames of the images in **pet_images/** identify the animal in each image.

To create the *labels* for pet images you will need to:

- Read all the files' names in the **pet_image/** folder
- Process the filenames to create the pet image labels
- Format the pet image labels such that they can be matched to:
- The classifier function labels
- The dog names in dognames.txt

In the first task the function will need to read the filenames from a folder. To achieve this task you will need to only import the **listdir method** from the **os python module**. The **listdir** method retrieves all filenames from the files within a folder. These filenames are returned from **listdir** as a list. The code below demonstrates how to perform this import and retrieval.

```
# Imports only listdir function from OS module
from os import listdir

# Retrieve the filenames from folder pet_images/
filename_list = listdir("pet_images/")

# Print 10 of the filenames from folder pet_images/
print("\nPrints 10 filenames from folder pet_images/")
for idx in range(0, 10, 1):
```

```
    print("{:2d} file: {:>25}".format(idx + 1,
filename_list[idx]) )
```

How to create a Dictionary of Lists (similar to the Results Dictionary)

The Python **Dictionary** is the data structure you should use for the Pet Image filenames (as **keys**) and a *List* that contains the filenames associated labels (as **values**). The following are reasons for this data structure choice:

- The key-value pairs of a dictionary are a logical choice because the need to process the same filenames (keys) with the **classifier function** and compare its returned labels to those of pet image (values)
- Given an input key, retrieval of the associated value is quicker than retrieval from other data structures (e.g. lists).

Dictionary Usage Review

In the **Data Types and Operators** lesson you first learned about dictionaries . The code below will help you use Python dictionaries.

- Creating an empty dictionary
- Creating a dictionary that contains a *List* as the value.
- Determines number of items in a dictionary
- Adds key-value pairs to dictionary if key doesn't already exist in the dictionary
- Iterates through a dictionary printing all key-value pairs in a dictionary

```
# Creates empty dictionary named results_dic
results_dic = dict()

# Determines number of items in dictionary
items_in_dic = len(results_dic)
print("\nEmpty Dictionary results_dic - n items=",
items_in_dic)

# Adds new key-value pairs to dictionary ONLY when key
doesn't already exist. This dictionary's value is
# a List that contains only one item - the pet image label
filenames = ["beagle_0239.jpg", "Boston_terrier_02259.jpg"]
pet_labels = ["beagle", "boston terrier"]
for idx in range(0, len(filenames), 1):
    if filenames[idx] not in results_dic:
        results_dic[filenames[idx]] = [pet_labels[idx]]
```

```
    else:
        print("** Warning: Key=", filenames[idx],
              "already exists in results_dic with value =",
              results_dic[filenames[idx]])

#Iterating through a dictionary printing all keys & their
associated values
print("\nPrinting all key-value pairs in dictionary
results_dic:")
for key in results_dic:
    print("Filename=", key, "   Pet Label=", results_dic[key]
[0])
```

(For more details on the dictionary of lists see the **Classifying Images** section that's after the **Project Workspace - Pet Image Labels**).

Pet Image File Format for Label Matching

With this project you will need to determine matches between the pet image labels and the classifier labels. To be able to accomplish this matching task with your function, you will need to understand the format of both labels. Below is a detailed description of the format the pet image filenames that you will be using to create the pet image labels.

Pet Image Files

The pet image files are located in the folder *'pet_images'*, in the workspace. Some examples of the filenames you will see are: Basenji_00963.jpg, Boston_terrier_02259.jpg, gecko_80.jpg, fox_squirrel_01.jpg

There are:

- 40 total images pet images
- 30 images of dogs
- 10 images of animals that aren't dogs
- Name (label) of image (**needed for comparison**)
- contains upper and lower case letters
- contains one or more words to describe the image (label)
- words are separated by an underscore (_)

Python functions to create Pet Image Labels

The best format for each pet image name would be:

- Label: with only lower case letters

- Blank space separating each word in a label composed of multiple words
- Whitespace characters stripped from front & end of label

In the Lesson **Data Types and Operators** the sections on *Strings* and *String Methods*, you first learned about the string data type. Based upon the pet image filename format *above*, you can use the following string functions to achieve the label format:

- **lower()** - places letters in lower case only.
- **split()** - returns a list of words from a string, where the words have been separated (split) by the delimiter provided to the split function. If no delimiter is provide, splits on whitespace.
- **strip()** - returns a string with leading & trailing characters removed. If no characters are provided, strips leading & trailing whitespace characters.
- **isalpha()** - returns true only when a string contains only alphabetic characters, returns false otherwise.

Below you will find some code that demonstrates use of the above string functions.

```
# Sets pet_image variable to a filename
pet_image = "Boston_terrier_02259.jpg"

# Sets string to lower case letters
low_pet_image = pet_image.lower()

# Splits lower case string by _ to break into words
word_list_pet_image = low_pet_image.split("_")

# Create pet_name starting as empty string
pet_name = ""

# Loops to check if word in pet name is only
# alphabetic characters - if true append word
# to pet_name separated by trailing space
for word in word_list_pet_image:
    if word.isalpha():
        pet_name += word + " "

# Strip off starting/trailing whitespace characters
pet_name = pet_name.strip()

# Prints resulting pet_name
print("\nFilename=", pet_image, "   Label=", pet_name)
```

`#TODO: 3`: Classifying Images

Implement the classify_images() function to create the classifier labels with the classifier function using in_arg.arch. Compare the labels, and create a dictionary of results (result_dic).

Code within the check_images.py and classify_images.py

Code to Edit

This section will help you code the function **classify_images** within ***classify_images.py***. With this function you will be creating the labels for the images using the classifier function. Additionally, you will be comparing these classifier's labels to the pet image labels. Finally, you will be storing the classifier generated labels, and the comparison of labels in the results dictionary (complex data structure) that was returned by **get_pet_labels** function.

- Code within the function `def classify_images()` as indicated by `#TODO: 3` within ***classify_images.py***
  - Using the comments and docstring within ***classify_images.py*** to define **classify_images**
- Code within the `main()` function within ***check_images.py*** indicated by `#TODO: 3`
  - Replace the first *None* within the function call to **classify_images** with *in_arg.dir* and replace the last *None* in the function call with *in_arg.arch*

Expected Outcome

When completed this code will return a dictionary of lists with the *pet image filename* as the *key* and the *value* will be a list for all 40 pet images in the *pet_image* folder. This list will contain the following items:

- Pet image label (index 0)
- Classifier label (index 1)
- Comparison of labels (index 2)

Checking your code

The **check_classifying_images** function within ***check_images.py*** will check your code. This function will print out all the *matches*

between the classifier and pet image labels and then all the *non-matches* between the labels.
Visually check that the results show:

- Matches between Classifier and Pet Image Labels are *real* matches
- Non-matches between Classifier and Pet Image Labels are *real* non-matches
- The number of *matches* added to the number of *non-matches* totals **40**, to account for all 40 images in *pet_images* folder

Project Workspace - Classifying Images

- The next concept will have your workspace to work on `#TODO: 3`
- Editing of ***check_image.py*** and ***classify_Images.py*** can be done within the **Project Workspace - Classifying Images**

For additional information and help on `#TODO: 3`, please look at the information below:

How to use the Classifier function

Testing the classifier function

Test the environment and the **classifier function** that we will be using to classify the pet images. This function is located in the *classifier.py* program. Test your environment by running the *test_classifier.py* program following the instructions below. You can look at the *test_classifier.py* program to see how to use the **classifier function** within ***classify_images.py***.

1. Go to the **Lab Workspace - Classifying Images** concept.
2. Open a terminal.
3. Type the following on the command-line to test the classifier.py program. The image of the **collie** should be correctly classified as a **collie**. `python test_classifier.py`
4.

Details on *test_classifier.py*

Look at the **test_classifier.py** program you will notice the following:

- The classifier function has to be imported into your program (*this is already done in **classify_images.py***).
- The classifier function takes in two arguments:
  - The *full image path* (includes folder and filename).

        - Folder and filename stored as separate variables, can be **concatenated** together into a single string.
    - The CNN *model* architecture.
        - Must be *resnet*, *vgg* or *alexnet*

To view the code of `test_classifier.py` open the program in the Project Workspace - Classifying Images.

Classifier Label Format for Label Matching

Your function will need to be able to determine matches between the pet image labels and the labels the classifier function returns. To be able to accomplish this matching task with your function, you will need to understand the format of the classifier labels. Below is a detailed description of the format of the classifier labels.

Classifier Labels

Labels are located in file *imagenet1000_clsid_to_human.txt* that you will see in the project workspace.

Label Information:

- 1000 total labels
    - associated to 118 different dog breeds
        - dog breeds are associated to Ids 151: Chihuahua to 268: Mexican hairless
    - associated to 882 images that aren't dogs
- label format:
    - a mixture of upper and lower case letters
    - a single word that identifies the image
        - Ex. *beagle*
    - multiple words separated by spaces that identify the image
        - Ex. *German shorthaired pointer*
    - a number of different terms separated by a comma (,) that all identify the same image
        - Ex. *cocker spaniel, English cocker spaniel, cocker*

Comparing Pet Image Labels to Classifier Labels

In the **Creating Pet Image Labels** concept we formatted the pet image labels to be:

- Label composed of all lower case letters

- Blank space separating each word in label composed of multiple words
- Whitespace characters stripped from front & end of label
- Examples:
  - beagle
  - cocker spaniel
  - polar bear

Looking at the classifier label format above, the only processing you will need to do is to put all the letters in lower case and to strip off any leading/trailing whitespace characters. The **in operation** can be used to determine if the pet image label matches one of the terms that compose the classifier label. The pet image labels are always only one term (even if multiple words make up that term). Therefore, if you discover (using the **in operation**) that your pet image label matches one term in the term (or terms) that make up a classifier label - then there's a match.

In the Lesson **Data Types and Operators** (of the Python lesson) you first learned about the string data type. To accomplish these formatting and matching tasks use the following string functions:

- **lower()** - places letters in lower case only.
- **strip()** - returns string with leading & trailing characters removed. If no characters are provided strips leading & trailing whitespace characters.
- **in operation** - returns *True* when a string exists within another string, otherwise returns *False*.

Data Structure for Results

Compound Data Structures

In the lesson **Data Types and Operators** you first learned about dictionaries. The **get_pet_labels** function returns a dictionary with the filenames as the *keys* and a list that contains only the pet image labels as the *values*. For the **classify_images** function, you can use the **extend** list function to add both the classifier label and the comparison to the results dictionary simultaneously.

In the lesson **Data Types and Operators**, you first learned about compound data structures. You created and used a nested dictionary to hold information about the elements. For the compound data structure we recommend using a dictionary of

lists(*values*). If you choose to use a different compound data structure the the following check functions will not work:

- **check_creating_pet_image_labels**
- **check_classifying_images**
- **check_classifying_labels_as_dogs**
- **check_calculating_results**

The reason behind this choice of data structures is:

- It's easier to access elements of the list using index values
- You can use the **sum()** function with **slicing** to quickly classify the results

Computing the Results

For this function you will be inputing the *results_dic* dictionary that contains:

- The *filenames* as *keys*
- A *list* whose only item is the *pet image label*.

You will need to:

1) Iterate through this dictionary (*results_dic*) processing each pet image (filename) with the **classifier function** to get the classifier label.

2) Compare the pet image and classifier labels to determine if they match.

3) Add the results to the results dictionary (*results_dic*).

(In **Mutable Data Types and Functions** you learned that because the results dictionary is a mutable data type you don't need to return it from the **classify_images** function).

Coding recommendations as to prevent issues with **classify_images**:

- With the **classifier function**, be certain to concatenate the *images_dir* with the *filename* to represent the *full* path to each pet image file.
- Put classifier labels in *all* lower case letters, stripping leading and trailing whitespace from the label.
- ***results_dic*** will have the following format:
  - *key* = pet image filename (ex: Beagle_01141.jpg)
  - *value* = List with:
    - index 0 = Pet Image Label (ex: beagle)
    - index 1 = Classifier Label (ex: english foxhound)

        - index 2 = 0/1 where 1 = labels match , 0 = labels don't match (ex: 0)
    - example_dictionary = {'Beagle_01141.jpg': ['beagle', 'english foxhound', 0]}
- To *initialize* a key in *results_dic*, use the *assignment* operator (=) to assigning the value of a list.
- To add a single item to the list of an **existing** *key* in *results_dic*, append to the list using either *+=* operator or the *append* function.
- To add multiple items simultaneously to the list of an **existing** *key* in *results_dic*, use the **extend** list function.

For more details on using a dictionary of lists see the example code below. This demonstrates the difference between *initializing* a key-value pair and adding to the list of an *existing* key-value pair. The code also demonstrates how to iterate through a dictionary of lists to access each element of the list.

```
# Defining lists to populate dictionary
filenames = ["Beagle_01141.jpg", "Beagle_01125.jpg",
"skunk_029.jpg" ]
pet_labels = ["beagle", "beagle", "skunk"]
classifier_labels = ["walker hound, walker foxhound",
"beagle",
                     "skunk, polecat, wood pussy"]
pet_label_is_dog = [1, 1, 0]
classifier_label_is_dog = [1, 1, 0]

# Defining empty dictionary
results_dic = dict()

# Populates empty dictionary with both labels &indicates if
they match (idx 2)
for idx in range (0, len(filenames), 1):
    # If first time key is assigned initialize the list with
pet &
    # classifier labels
    if filenames[idx] not in results_dic:
        results_dic[filenames[idx]] = [ pet_labels[idx],
classifier_labels[idx] ]

    # Determine if pet_labels matches classifier_labels using
in operator
    # - so if pet label is 'in' classifier label it's a match
```

```
    # ALSO since Key already exists because labels were
added, append
    # value to end of list for idx 2
    # if pet image label was FOUND then there is a match
    if pet_labels[idx] in classifier_labels[idx]:
        results_dic[filenames[idx]].append(1)

    # if pet image label was NOT found then there is no match
    else:
        results_dic[filenames[idx]].append(0)

# Populates dictionary with whether or not labels indicate a
dog image (idx 3&4)
for idx in range (0, len(filenames), 1):
    # Key already exists, extend values to end of list for
idx 3 & 4

results_dic[filenames[idx]].extend([pet_label_is_dog[idx],

classifier_label_is_dog[idx]])

# Iterates through the list to print the results for each
filename
for key in results_dic:
    print("\nFilename=", key, "\npet_image Label=",
results_dic[key][0],
          "\nClassifier Label=", results_dic[key][1],
"\nmatch=",
          results_dic[key][2], "\nImage is dog=",
results_dic[key][3],
          "\nClassifier is dog=", results_dic[key][4])

    # Provides classifications of the results
    if sum(results_dic[key][2:]) == 3:
        print("*Breed Match*")
    if sum(results_dic[key][3:]) == 2:
        print("*Is-a-Dog Match*")
    if sum(results_dic[key][3:]) == 0 and results_dic[key][2]
== 1:
        print("*NOT-a-Dog Match*”)
```

#TODO: 4: Classifying Labels as Dogs

Implement the adjust_results4_isadog() function to adjust the results of dictionary(result_dic) to determine if the classifier correctly classified images as 'a dog' or 'not a dog'.

Coding within the check_images.py and adjust_results4_isadog.py

Code to Edit

This section will help you code the *undefined* function **adjust_results4_isadog** within ***adjust_results4_isadog.py***.
With this function you will:

- Read in the dog names from *dognames.txt* file into a data structure (like a dictionary)
- Compare the dog names to the classifier and pet image labels in the results dictionary
- Adjust the results dictionary to indicate whether or not these labels indicate the image is 'of-a-dog'.
  Note that the **adjust_results4_isadog** function will change the results dictionary, but because dictionaries are mutable you won't have to return this dictionary (review the section *Mutable Data Types and Functions* if you want a more detailed explanation).
- Code for the function definition `def adjust_results4_isadog():` as indicated by `#TODO: 4` within ***adjust_results4_isadog.py***.
- Using the comments and docstring within ***adjust_results4_isadog.py*** to define **adjust_results4_isadog**
- Code within the `main()` function within ***check_images.py*** indicated by `#TODO: 4`
- Replace *None* within the function call to **adjust_results4_isadog** with *in_arg.dogfile*.

Expected Outcome

When completed this code will have altered the results dictionary (a dictionary of lists) with the *pet image filename* as the *key* and the *value* will be a list for all 40 pet images in the *pet_image* folder. This list for each *key* will now contain two additional items: whether the

pet image label is of-a-dog (index 3) and whether the classifier label is of-a-dog (index 4).

Checking your code

The **check_classifying_labels_as_dogs** function within ***check_images.py*** will check your code. This function will print out all the *matches* between the classifier and the pet image labels and all the *non-matches* between the labels.

Visually check that the results show:

- Matches between Classifier and Pet Image Labels have both labels classified as "dogs" or "not dogs" as appropriate for the labels.
- Non-matches between Classifier and Pet Image Labels correctly classify each label as "dogs" or "not dogs"
- The number of *matches* added to the number of *non-matches* totals **40**, to account for all 40 images in *pet_images* folder.

Project Workspace - Adjusting Results

- The next concept will have your workspace to work on `#TODO: 4`
- Editing of ***check_image.py*** and ***adjust_results4_isadog.py*** can be done within the **Project Workspace - Adjusting Results**

For additional information and help on `#TODO: 4`, please look at the information below:

Define: "Is a Dog" / "Is NOT a Dog" by Reviewing Dog Names File

Lets remember the principle objectives _1_ and _2_:

1. Correctly identify which pet images are of dogs (even if breed is misclassified) and which pet images aren't of dogs.
2. Correctly classify the breed of dog for the images that are of dogs.

To achieve objectives _1_ and _2_, your program will have to be able to identify if labels from both the classifier function and the pet images are of "*a dog*" or "*not a dog*". To be able to classify the labels as "*dogs*" or "*not dogs*", your program will need to compare

the labels to the list of dogs contained in the **dognames.txt** file in the workspace.
This **dognames.txt** file was created from the formatted labels (lower case, whitespace trimmed, etc.). Therefore, when comparing *dog names* (from dognames.txt) to your *labels*:

- if there is a match between *dog name* and *label*, the label "**is a dog**"
- if there isn't a match between *dog name* and *label*, the label "**is *not* a dog**"

Details about *dognames.txt*:
There are:

- One dog breed name per line
- 223 dog breeds named
- All possible dog breeds that can come from the classifier function and pet image labels
- *Classifier Function Labels:*
- Lines **1** (chihuahua) to **118** (mexican hairless), these should match classifier returned labels as long as those labels are in *all* lower case letters and have whitespace characters trimmed from the ends.
- *Pet Image Labels:*
- Should match the following lines as long as the labels are in *all* lower case letters, whitespace has been trimmed from the ends, and there is a single space separating the words of a dogs name (if it's composed of multiple words).

Reading In Dogsname.txt
The first task of the **adjust_results4_isadog** will be to read in all the *dog names* and store them in a data structure. Given the details above, the most ideal data structure for our *dog names* is a dictionary with the key as the *dog name* and a value of 1 (arbitrary value). The reasoning behind this choice is because the speed of the look-up of a dictionary. Since we know the labels should match the *dog name* exactly if there is a match; we can simply *look* for the label as a key within the dog names dictionary to discover all labels that are *dogs*. If the label isn't found as a key in the dog names dictionary, then we know that label is *not* a dog.

In the lesson **Scripting** we demonstrated how to open and read information from a file. Please review this section if you are having difficulty reading the dog names from *dognames.txt*.
Coding recommendation as to prevent issues with **adjust_results4_isadog**:

- Define the *dognames_dic* prior to opening the *dognames.txt* file for reading
- Use **rstrip()** to strip off newline characters from each line read from *dognames.txt*
- If a *dog name* already exists in *dognames_dic* print a **Warning** statement because you shouldn't find any duplicate *dog names* in *dognames.txt*

Adjusting Results Dictionary

Once you have read in the *dog names* into *dognames_dic*, you will need to adjust the results dictionary (*results_dic*) to account for when labels were correctly or incorrectly classified as dogs.
Review section *Classifying Labels as Dogs* to review how to iterate through the results dictionary to append values onto the *list* that is the *value* for each *key* in the results dictionary. If you want to append both values at the same time, you will need to use the **extend** list function to add both **index 3** and **index 4** to the *results_dic* simultaneously.
results_dic will have the following ***adjusted*** format:

- *key* = pet image filename (ex: Beagle_01141.jpg)
- *value* = List with:
- index 0 = Pet Image Label (ex: beagle)
- index 1 = Classifier Label (ex: english foxhound)
- index 2 = 0/1 where 1 = labels match , 0 = labels don't match (ex: 0)
- ***index 3*** = 0/1 where 1= Pet Image Label is a dog, 0 = Pet Image Label isn't a dog (ex: 1)
- ***index 4*** = 0/1 where 1= Classifier Label is a dog, 0 = Classifier Label isn't a dog (ex: 1)
- example_dictionary = {'Beagle_01141.jpg': ['beagle', 'walker hound, walker foxhound', 0, 1, 1]}

`#TODO: 5`: Calculating Results

Implement the calculates_results_stats() function to calculate the results of run and put statistics in a results statistics dictionary (results_stats_dic).

Coding within the check_images.py and calculates_results_stats.py

Code to Edit

This section will help you code the *undefined* function **calculates_results_stats** within ***calculates_results_stats.py***. With this function you will be inputing the results dictionary to create a dictionary of results statistics. This results statistics dictionary will contain the *statistic's name* as the *key* and the *value* will simply be the *statistic's numeric* value.

You will be creating the results statistics dictionary within **calculates_results_stats**. This means we recommend that you create this dictionary as the first line in your function and return it's value with the last line in your function. For a more detailed explanation regarding why this dictionary is returned by the function, review the section *Mutable Data Types and Functions*.

- Code for the function definition `def calculates_results_stats():` as indicated by `#TODO: 5` within ***calculates_results_stats.py***
- Using the comments and the docstring within ***calculates_results_stats.py*** to define **calculates_results_stats**

Expected Outcome

When completed, this code will be able to provide counts and percentages that will be used to answer the objectives of this lab. The percentages provided by the *results_stats* dictionary will answer objectives 1 and 2. The counts will be used to compute the percentages.

Checking your code

The **check_calculating_results** function within ***check_images.py*** will check your code by recalculating the following results statistics (counts and percentages) and comparing them to the results

statistics that you calculated and stored within the results statistics dictionary.
Results Statistics checked:

- **Counts**:
- Number of Images
- Number of Dog Images
- Number of "Not-a" Dog Images
- **Percentages**:
- % Correctly Classified Dog Images
- % Correctly Classified "Not-a" Dog Images
- % Correctly Classified Breeds of Dog Images

Visually check that the results from this *checking* code for these six statistics above *match* the results you computed with **calculates_results_stats** within ***calculates_results_stats.py***.

Project Workspace - Calculating Results

- The next concept will have your workspace to work on `#TODO: 5`
- Editing of ***check_image.py*** and ***calculates_results_stats.py*** can be done within the **Project Workspace - Calculating Results**

For additional information and help on `#TODO: 5`, please look at the information below:

How to Define Percentages for Summarizing the Results

Principle Objectives _1_ and _2_:

1. Correctly identify which pet images are of dogs (even if breed is misclassified) and which pet images aren't of dogs.
2. Correctly classify the breed of dog, for the images that are of dogs.
3.

To achieve objectives _1_ and _2_, your program will need to be able to calculate the following percentages based upon the results of comparing the labels contained within the results dictionary.
The **results dictionary** will have the following format:

- *key* = pet image filename (ex: Beagle_01141.jpg)
- *value* = List with:
- index 0 = Pet Image Label (ex: beagle)
- index 1 = Classifier Label (ex: english foxhound)

- index 2 = 0/1 where 1 = labels match , 0 = labels don't match (ex: 0)
- index 3 = 0/1 where 1= Pet Image Label is a dog, 0 = Pet Image Label isn't a dog (ex: 1)
- index 4 = 0/1 where 1= Classifier Label is a dog, 0 = Classifier Label isn't a dog (ex: 1)
- example_dictionary = {'Beagle_01141.jpg': ['beagle', 'walker hound, walker foxhound', 0, 1, 1]}

You will be storing these calculations (counts & percentages) in the results statistics dictionary. We recommend using the same prefix for all counts (e.g. *n_*) and percentages (e.g. *pct_*) in the statistic's name(*key*) as to make it easier to print all of them for each group. The **results statistics dictionary** will have the following format:

- *key* = statistic's name (e.g. n_correct_dogs, pct_correct_dogs, n_correct_breed, pct_correct_breed)
- *value* = statistic's value (e.g. 30, 100%, 24, 80%)
- example_dictionary = {'n_correct_dogs': 30, 'pct_correct_dogs': 100.0, 'n_correct_breed': 24, 'pct_correct_breed': 80.0}

Counts Computed from the *Results* dictionary for input into the *Results Statistics* dictionary:

- **Z**: Number of Images
- length of *results_dic*, because filenames = key
- **A**: Number of Correct Dog matches
- Both labels are of dogs: *results_dic*[key][3] = 1 and *results_dic*[key][4] = 1
- **B**: Number of Dog Images
- Pet Label is a dog: *results_dic*[key][3] = 1
- **C**: Number of Correct Non-Dog matches
- Both labels are NOT of dogs: *results_dic*[key][3] = 0 and *results_dic*[key][4] = 0
- **D**: Number of Not Dog Images
- number images - number dog images --OR--
- Pet Label is NOT a dog: *results_dic*[key][3] = 0
- **E**: Number of Correct Breed matches

- Pet Label is a dog & Labels match: *results_dic*[key][3] = 1 and *results_dic*[key][2] = 1
- (*Optional*) **Y**: Number of label matches
- Labels match: *results_dic*[key][2] = 1

Compute a Summary of the Percentages from the *Results Statistics* dictionary counts:

- **Objective _1_a**: Percentage of Correctly Classified Dog Images
- **A** Correctly classified *dog* images.
- **B** Number of *dog* images
- Percentage of correctly classified "*dog*" images: **A**/**B** * 100
- **Objective _1_b**: Percentage of Correctly Classified Non-Dog Images
- **C** Correctly classified *NOT a dog* images.
- **D** Number of *NOT a dog* images
- Percentage of correctly classified "*Non-dog*" images: **C**/**D** * 100
- **Objective _2_**: Percentage of Correctly Classified Dog Breeds
- **E** Correctly classified as a particular breed of *dog* images.
- **B** Number of *dog* images
- Percentage of correctly classified *Dog* Breed images: **E**/**B** * 100
- (*Optional*): Percentage Label Matches (regardless if they're a dog)
- **Y** Number of label matches
- **Z** Number of images
- Percentage of correctly Matched Images (regardless if they are a dog): **Y**/**Z** * 100

Important Notes:

- You will need to initialize all the counts to a value of zero before iterating through *results* dictionary. As you iterate through *results* dictionary, if certain criterion are met you will need to increment these counters by 1.
- The percentages (and total number of images) can be generated from the counts (see percentage & count calculations above); therefore, these values should be

calculated *after* counts have been calculated by iterating through the *results* dictionary.

- When calculating the percentage of correctly classified Non-Dog Images, use a conditional statement to check that **D**, the number of "not-a-dog" images, is greater than zero. To avoid division by zero error, only if **D** is greater than zero should **C/D** be computed; otherwise, this should be set to 0.
- Because the *Results Statistics* dictionary is created inside of the function and is a mutable object, you will need to *return* it's value at the end of the function (see section *Mutable Data Types and Functions*).

`#TODO: 6` Printing The Results

Implement the print_results() function to print a summary of the results (as well as incorrect classifications of dogs and breeds if requested).

Coding within the *check_images.py* and *print_results.py*

Code to Edit

This section will help you code the *undefined* function **print_results** within ***print_results.py***. With this function you will be inputting the *results* dictionary and the *results statistics* dictionary to print a summary of the results. Because this function allows one to print a list of incorrectly classified dogs and incorrectly classified breeds of dog, one needs to include the *results* dictionary.

- Code for the function definition `def print_results():` indicated by `#TODO: 6` within ***print_results.py***
- Using the comments and the docstring within ***print_results.py*** to define **print_results**
- Code within the `main()` function within ***check_images.py*** indicated by `#TODO: 6`
- Replace the *None* within the function call to **print_results** with *in_arg.arch*

Expected Outcome

When completed, this code will print the summary of the results that will be used to answer objectives 1 and 2 of this project.

Checking Your Code

For this you will just be running the completed program to visually check the following:

- Running the program results in the statistics and counts being properly printed and formatted. Results from the *code check* for *Calculating Results* should match the values printed for those 6 statistics.
- Leaving *off* the two default arguments in the function call to **print_results**, results in no misclassifications being printed (That's an expected default behavior).
- Adding the values of *True* for the two default arguments in the function call to **print_results**, results in the misclassifications being printed (That's also an expected default behavior).

Final Program Run

Once you are satisfied the program is running properly use batch processing (see section on *Batch Processing* below) to run the program for all **three** CNN model architectures. You will use these results to compare them with our results in the section **Final Results**.

Project Workspace - Printing Results

- The next concept will have your workspace to work on `#TODO: 6`
- Editing of ***check_image.py*** and ***print_results.py*** can be done within the **Project Workspace - Printing Results**.

For additional information and help on `#TODO: 6`, please look at the information below:

Printing Results

The first thing to be printed is a general statement that indicates which of the three CNN model architectures you are using. You can pass the information in the *model* as an input parameter to be able to print it.

Next you will be printing the overall count which will be the same for all three CNN model architectures. This can be done by calling for those counts using the appropriate key within a print statement.

- Number of Images
- Number of Dog Images
- Number of "Not-a" Dog Images

Finally, you will be iterating through the *results_stats* dictionary printing out the statistic's name and value for all of the percentages (e.g. key that starts with the letter "*p*"). Recall that we had recommended that you give the same prefix (e.g. *pct_*) to all of the percentage statistics, so that they could all be printed out as a group.

Percentage Calculations:

- % Correct Dogs
- % Correct Breed
- % Correct "Not-a" Dog
- % Match (*optional* - this includes both dogs and not-a dog)

Printing Misclassifications

This function allows one to *optionally* print cases of *dog* and *breed* misclassifications.

This optional feature is provided to allow improved debugging of the code. Additionally, this feature provides the ability to determine if there are certain breeds of dogs that the algorithms have difficulty identifying.

Default Arguments for Misclassification

The function **print_results** contains two default arguments for printing misclassified *dogs* and *breeds*. (In the lesson **Functions** you first learned about default arguments).

Default Arguments:

- *print_incorrect_dogs* - defaults to *False*
- *print_incorrect_breed* - defaults to *False*

Purpose

The purpose of default arguments can be the following:

- To provide a wider range of behaviors for a function, without having to code multiple (similar) functions.
- To guarantee that certain arguments are always assigned a value within a function.
- To provide a default behavior for a function.

Misclassified Dogs

Labels are misclassified as dogs when both labels aren't in agreement regarding whether or not an image is of a dog.

Prior to iterating through the *results* dictionary to find dog misclassifications, you must first check that the user *wants* to print

misclassified dogs and that dog misclassifications *occurred* with a conditional statement.
This check is done when:

- User wants to print misclassifications:
- *print_incorrect_dogs* == True
- Some dogs were misclassified:
- n_correct_dogs + n_correct_notdogs != n_images

If the check is *True*, then print the pet image and classifier labels for misclassified dogs when:

- The labels disagree on whether or not an image is of a "dog"
- sum(results_dic[key][3:]) == 1

Misclassified Breed's of Dog

Labels have a misclassification of breeds of dog when both labels indicate that the image is a dog; but, labels aren't in agreement regarding the dog's breed.
Prior to iterating through the *results* dictionary to find breed misclassifications, you must first check that the user *wants* to print misclassified breeds and that breed misclassification *occurred* with a conditional statement.
This check is done when:

- User wants to print misclassifications:
- *print_incorrect_breed* == True
- Some breeds were misclassified:
- n_correct_dogs != n_correct_breed

If the check is *True*, then print the pet image and classifier labels for misclassified breeds when:

- When the labels agree that image is of a dog, but disagree on the breed of dog
- sum(results_dic[key][3:]) == 2 and results_dic[key][2] == 0

Batch Processing

Now that you have completed coding check_images.py, you are ready to run it on all 3 models. One way to do this is to call the program from the terminal window for one of the models, wait until it completes running, record it's results, and then repeat for the other two models.

An easier way to handle this task is with batch processing using a shell script. For this exercise, you will find the bash program **run_models_batch.sh** in the workspace. Open that file and you will notice comments use # just like python and the rest look the same as the commands you type into the terminal window to run your program (see code below).

```
#  Code from run_models_batch.sh
python check_images.py --dir pet_images/ --arch resnet  --
dogfile dognames.txt
     > resnet_pet-images.txt
python check_images.py --dir pet_images/ --arch alexnet  --
dogfile dognames.txt
     > alexnet_pet-images.txt
python check_images.py --dir pet_images/ --arch vgg  --
dogfile dognames.txt
     > vgg_pet-images.txt
```

You will also notice that each file ends with *> filename.txt*. The *>* is a pipe and it pipes the output from the console into a file. The file contains the filename of the model being used. This way after each run, the results are automatically stored in your workspace.

To run file **run_models_batch.sh** in the workspace, open a terminal window (in Unix/Linux/OSX/Lab Workspace) and type the following:

```
sh run_models_batch.sh
```

If you want to *batch process* the program on a Windows computer you will need to follow the instructions found **here**.

Once you have ran all three models using `run_models_batch.sh` (`run_models_batch.bat` on Windows) compare your results with those you will find in the section **Final Results**.