Name:
Instructor:

Midterm Project
Assessing Website Accessibility

Written by: Annie Ross, Bucknell University
*Must be completed **INDIVIDUALLY***

CHECK-IN (Moodle Questionnaire) DUE: 10/18 11pm
PROJECT DUE: 10/25 11pm
Written by: Anne Spencer Ross | modified from previous CSCI 203 projects

Make a copy of this document, you will be uploading it as a PDF to gradescope with your submission.

Grading Overview

100 pt total.
Full Rubric to be released soon.
Grade will be based on

- Code functionality (does it work as intended)
- Code style (comments, variable names, etc)
- Evidence of incremental planning and testing including
    - if DEBUG blocks with print statements
    - documentation captured in this worksheet
- Moodle quizzes

Where should you code?

The final project provides you with the opportunity to show us what you have learned this semester. All coding must be completed on your own in the project replit on replit.com. The progression of your work is then clearly documented.

Can I use the internet?

👍 You **are** allowed to use online resources to help you *understand* a topic in this course.
❗ Any online sources that you use **must be cited as a comment in your code.**

Name:
Instructor:

- 🚫 You are **not** allowed to search for *specific* code or solutions to specific problems you're assigned

What kind of collaboration is allowed?

- 👍 You **are** allowed to *talk* to students about approaches to a problem and resources they used to help them solve problems. In fact, you probably should - that's one of the advantages of being at a school like Bucknell.
- 👍 You **are** allowed to get help from professors and TAs.
- 🚫 You are **not** allowed to show another student your code on an individual project or assignment. This means that you should not let another student "borrow" the code. This is the *most common case* that we (regretfully) bring to the Board of Review.
- 🚫 You are also not allowed to publicly post your code to assignments in this class - either on Github or in a publicly-accessible cloud storage folder (like Drive or Dropbox).

Project Introduction

People access information through different modalities including through visual, audio, and tactile feedback. Some people need or prefer to receive information through audio. Audio can be particularly important for people who are blind, have a vision impairment, are in a situation where they must keep their eyes elsewhere (e.g., driving), are in a poor visibility situation, have a learning disability, or otherwise prefer to have content read aloud instead of or in addition to having it shown visually.

Screen readers are an assistive technology that can take visual information and read it aloud. This video demonstrates Android's TalkBack screen reader being used on the Duolingo app. Note how much information you can get from listening to the audio or reading the captions without seeing the screen.

Whether you consider social justice, usability and customer satisfaction, or legal angles, it is imperative we make technology that works for everyone, including people who use assistive technologies like screen readers.

However, not all apps work well with screen readers. If the developer does not provide a description of an image, the screen reader won't know what to say, as we heard with the quit button in the Duolingo demo above. This version of the Freegal audiobook app used by public libraries is another example of a large number of images missing labels

See this blog for more accessibility overview.

In this project you will assess image labeling for screen readers on websites.

HTML Overview

HTML is one of the base programming languages for building web pages. We will not be learning HTML in this class, but understanding a few basics will help you understand the data you're assessing. The building blocks of items on a webpage are defined in tags, denoted with **< >**.

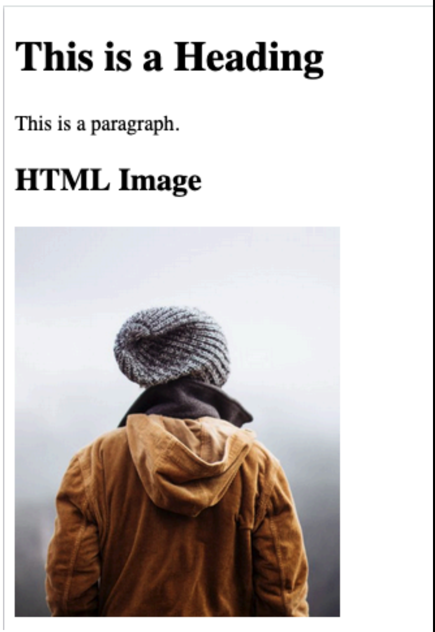
Here is an example of a basic website.

```
<!DOCTYPE html>
<html>
<body>


<h1>This is a Heading</h1>
<p>This is a paragraph.</p>

<h2>HTML Image</h2>
<img src="img_girl.jpg" alt="Girl in a
jacket">

</body>
</html>
```



This is a Heading

This is a paragraph.

HTML Image

Of particular interest to us is the code used to insert the image:

`<img src="img_girl.jpg" alt="Girl in a jacket">`

- `img`: the tagname indicating this will be an image on our website
- `src`: an attribute of the image, the name of the image file to load
- `alt`: an attribute of the image, what a screen reader will announce

If the **alt** attribute of an image element is missing, the screen reader will not know what to call that image or may skip it all together, creating a barrier for someone relying on a screen reader.

Name:
Instructor:

Project Overview

In this project, we will analyze the HTML file of the landing page for Bucknell from 1996-2022 and count the number of unlabeled images. In the `bucknell_web` directory/folder, you will see files labeled `bucknell_<date>.html` for 1996-2022. We gathered historic pages using [The Internet Archive Wayback Machine.]() We will work up to analyzing all these pages. You will not need to edit or understand these files.

Here we've broken that task into smaller steps. We'll break these steps down even further as we go.

Step	
1	write CSV header to file
4	for every HTML file in a folder
2	get date, num images, num unlabeled images, and proportion unlabeled
3	write results to a CSV file
5	read in results from CSV file
6	graph proportion of unlabeled images over time

We're going to practice key steps in programming problem solving. For every step above, we'll:

- Make simple, incremental test cases
- Break the step into sub-steps
- Address smaller problems first

In our goal of breaking down the problem, notice we'll first focus processing one html file (Steps 2-3), then will add code to process multiple files (Step 7).

Step 1: Write CSV header to file

We have provided a write_headers function that writes the header line to our CSV file (column labels). It is called in our `evalute_accessibility()` function.

Moodle Quiz Page 1: File I/O Review

☐ Complete the File I/O questions on PAGE 1 of the Midterm Proj Moodle quiz

Name:
Instructor:

Create Test HTML Files

Our goal centers around counting the number of images and unlabeled images in an HTML file. To help us check our code as we go, we will create simple HTML files where we KNOW how many unlabeled images there are. We want our tests to have good coverage, that is, cover all possible cases!

Worksheet: Plan Test Cases

Complete the below table planning out test cases.

Two test files have been given to you in the `test_html` directory: `test_1.html` and `test_4.html`. Note, for our purposes, the exact content of the image `src` and `alt` attributes does not matter. In the provided tests, we used values that would be easy to identify later and remind us what test case they represent.

Test Case	**Test File**	**Expected Num Images**	**Expected Num Unlabeled**	**Expected Proportion**
Page with mix of labeled and unlabeled	test_1.html	3	1	0.33
<COMPLETE>	test_2.html	<COMPLETE>	<COMPLETE>	<COMPLETE>
<COMPLETE>	test_3.html	<COMPLETE>	<COMPLETE>	<COMPLETE>
Page with no images	test_4.html	0	0	0

Implement: Create Test Files

In replit, create `test_2.html` and `test_3.html files in your test_html` directory that follow the test cases you outlined in the table.

As you progress, you may optionally choose to add additional test files to your test_html directory as needed.

Checkpoint: Test File Coverage

Upload your test_2.html and test_3.html files to gradescope to ensure your tests have sufficient coverage.

Name:
Instructor:

Step 2: Parse One HTML File

Our next step will focus on getting the data for a single HTML file. Let's break this step down into additional sub-steps.

Substeps

2.1	read in all image elements
2.2	get the number of images
2.3	count the number of unlabeled images
2.4	calculate the proportion of images missing labels
2.5	return results

We will be planning, implementing, and testing functionality in the parse_html function, adding helper functions as needed. We will be leveraging our test HTML files as we go.

Remember:

- use comments
- write as little code as possible before testing
- ensure you know your expected test output to check against your program

Moodle Quiz Page 2: parse_html

☐ Complete the parse_html questions on PAGE 2 of the Midterm Proj Moodle quiz

Check your understanding of the given parse_html function using the docstring/starting comment and DEBUG print statements to complete the table.

Note, filepath must be a full path (a path into the directories/folders the file is in), not just the file name.

E.g.,

```
filename = test_1.html
filepath = test_html/test_1.html
```

Step 2.1: Read in all image elements

We've provided the code that turns an HTML filepath into a more useful collection of images. Your task is to ensure you understand the given code.

Uncomment the line in evalute_accessibility() that calls parse HTML on a single file. Note how, if DEBUG = True, it will run on test_1.html. Note, we'll use our test's number (1,2,3,or 4) as a stand in for the file's date.

Name:
Instructor:

The provided code in the parse_html function defines a variable `images`. This code relies on the provided `parser.py` file to turn our `html_filepath` data into a more friendly structure containing information about the images on a given HTML page. Understanding the details of how parser.parse_file works is out of scope for this project, but we still want to make sure we understand how to work with the images variable.

Implement & Moodle Quiz: Explore Images Using Print Statements

The provided code prints the html_filepath and images variable. Answer the questions in Moodle about the images variable. In parallel, add print statements to the if DEBUG block to check your answers.

Using print debugging strategies, in the if DEBUG block, add code that prints information to answer the moodle quiz questions. We've provided you with the values to print for the first part of the quiz. You'll need to add at least three more print statements to answer all the moodle questions.

- ☐ **Upon completion, your if DEBUG block should have at least 6 print statements.**
- ☐ Complete PAGE 3, images, of the moodle quiz

Step 2.2: Get the number of images in a single HTML file

Now that we are familiar with our images variable, we can use it toward our goal of analyzing the webpage represented by the HTML file.

Implement: Calculate the number of image

For each step, ensure you are adding comments to your code.

- ☐ Outside of the if DEBUG statement, write code that creates a variable for the total number of images.

Worksheet: Plan your test:

For each test file, what do you expect the number of elements to be?

File	Num Elements
test_1.html	
test_2.html	
test_3.html	
test_4.html	

Name:
Instructor:

Implement: Check you got the correct values

- ☐ inside the next if DEBUG statement, print the variable's value
- ☐ run for all your test HTML files to assure correctness

Step 2.3: count the number of unlabeled images in an HTML file

We'll break this step into sub-steps to ensure we can check our approach:

- 2.1 print the src attribute of each element in images
- 2.2 print "no label" for each image without an alt attribute
- 2.3 count the number of unlabeled images

We'll incrementally plan and test each step.

Worksheet: Plan your test for going through each image

To check our understanding of working with the elements in the images variable, we are going to start by going through each element and printing its src attribute and "no label" if that image does not have an alt attribute.

Complete the table with what we expect the output to be for each of our files.

File	**Output images src and if it's labeled**
test_1.html	src: labeled_1.jpg src: not_lab_1.jpg no label src: empty_label_1.jpg
test_2.html	<COMPLETE>
test_3.html	<COMPLETE>
test_4.html	<COMPLETE>

Implement: Print the src attribute of each element in images

Outside of the DEBUG conditionals, write code that goes through each element in images and prints out its src attribute.

- ☐ ***FIRST plan out, in comments your steps***
- ☐ Then, implement the code
- ☐ Run the code with each test file and ensure it prints what you expect

Implement: Print "no label" for any image without an alt tag

Outside of the DEBUG conditionals,write code that prints "no label" for any element without an alt tag.

Name:
Instructor:

- ☐ **Run the code with each test file** and ensure it prints what you expect; compared to your plan above.
- ☐ Once you've confirmed correctness, comment out the lines that print src and "no label"

Worksheet: Plan your test for counting number of unlabeled images

Now that we've checked our ability to go through each element in images, we'll move on to our goal, counting the number of unlabeled images. Fill in the below table to remind ourselves of the expected values for each test.

File	Num Unlabeled
test_1.html	
test_2.html	
test_3.html	
test_4.html	

Implement: Count Unlabeled Images

We'll now build on this code to create a variable that will store the total number of unlabeled

- ☐ **Write comments** before you start coding how you will achieve this
- ☐ OUTSIDE the if DEBUG statement, Implement code to count the number of unlabeled images
- ☐ INSIDE the if DEBUG statement, print the variable
- ☐ Run your code with all test files and check the output against your expected results

Step 2.4: calculate the proportion of unlabeled images in a single file

Finally, we want to compute the proportion of unlabeled images in a single HTML file to the total number of images. The proportion will be a number from 0.0-1.0. Remember, if a file has no images, we'll consider it a proportion of 0.

Worksheet: Plan your tests

Remind yourself the expected proportion unlabeled images for each of your test files

File	**Proportion unlabeled**
test_1.html	0.33

Name:
Instructor:

test_2.html	<COMPLETE>
test_3.html	<COMPLETE>
test_4.html	0

Implement & Test: Calculate proportion

- ☐ **Write comments** before you start coding how you will achieve this
- ☐ OUTSIDE the if DEBUG statement, calculate the proportion and store it in a variable
- ☐ INSIDE the if DEBUG statement, print the variable
- ☐ Run your code with ***all test files*** and check the output against your expected results

Step 2.5: return results from single HTML file

The `parse_html` function returns a dictionary containing the values calculated for a single HTML file. For example, for test_1.html it will return {"DATE":1, "TOTAL_IMG":3, "NUM_UNL":1, "PROP_UNL": 0.33}

Implement: create and return results dictionary

- ☐ in parse_html, add code to return dictionary that contains the appropriate information
- ☐ run your code with all test files and ensure the values are as expected

Step 3: Write results to CSV file

Now that we've calculated information about the images in our HTML file, we want to write those results to our results csv file.

You need to:

- create a helper function that will take in the data from parse_html and write it to results_csv_filename
- call your function in evaluate_accessibility()

Notes:

- You will decide the name of your function and what parameters it will have.
- Your function **MUST HAVE a docstring comment detailing**
  - **brief description of the function**
  - **the parameter variable names, data types, and short description**
  - **return value data type and description**
- Your function should not write the header line, it should build on the file started by the write_headers function call.

Name:
Instructor:

Example Output:

After this step, running your program on **test_1.html** will create a file **test_img_count.csv** with the contents:

```
DATE,TOTAL_IMG,NUM_UNL,PROP_UNL
1,3,1,0.33
```

Worksheet: Break down into sub-steps

In Step 2, we broke down the step into sub-steps and guided you through a plan, implement, test process. In this section, you'll be responsible for mapping out a plan and demonstrating incremental testing.

You must **have at least 3 sub-steps** including a plan for how you will incrementally test each step with expected output.

Your testing can involve print statements in if DEBUG blocks, work in another file (e.g., main.py) as you test different parts of code, check file contents. Whatever method you choose, you must document your steps, expected results, and note how you checked your code works as intended.

Sub-Step 3.1: <Description>

<description>

3.1: Plan your test

<describe how you will test if step 3.1 works, including the expected output you will check>

3.1: Implement

- ☐ write your plan in comments
- ☐ implement 3.1
- ☐ run your tests and confirm it works as expected

Sub-Step 3.2: <Description>

<description>

3.2: Plan your test

<describe how you will test if step 3.2 works, including the expected output you will check>

3.2: Implement

- ☐ write your plan in comments
- ☐ implement this step

Name:
Instructor:

- ☐ run your tests and confirm it works as expected

Sub-Step 3.3: <Description>

<description>

3.3: Plan your test

<describe how you will test if this step works, including the expected output you will check>

3.3: Implement

- ☐ write your plan in comments
- ☐ implement this step
- ☐ run your tests and confirm it works as expected

<you may add more steps as needed>

Step 4: Parse Multiple HTML files

Now that we're confident our code can process one HTML file, we want to extend our program to process all HTML files in a given directory/folder. We'll start with processing all test files in our test_html directory. hi

Note: Even when DEBUG = False, we will want our program to print out progress. For example, when we run our program on test_html (which here we say has 4 test files, your directory may have more), we want the following to print to the console:

```
Processing test_html/test_1.html
Processing test_html/test_2.html
Processing test_html/test_3.html
Processing test_html/test_4.html
Finished processing html files
```

Worksheet: Plan your Final Expected Result

When this step is complete, the **test_img_count.csv** file will contain the data for all files in the **test_html** directory. Fill in the missing lines of the CSV file with data from your test files

```
DATE,TOTAL_IMG,NUM_UNL,PROP_UNL
1,3,1,0.33
<COMPLETE THIS BASED ON YOUR TEST FILE DATA>
4,0,0,0.0
```

Name:
Instructor:

Notes:

- you should only change code in the `evaluate_accessibility` function
- you should only need to have one line of code calling `parse_html` in your code. Consider the pattern in how we named our test files. Consider how we created a filepath in the code already given.
- our test files don't have dates like our actual data, but we can use the test file number as our date for testing (e.g., test_1.html would have date 1).

Worksheet: Break down into sub-steps

Break down Step 4, parsing multiple files into **at least 3 sub-steps** including a plan for how you will incrementally test each step with expected output.

Sub-Step 4.1: <Description>

<description>

4.1: Plan your test

<describe how you will test if step 4.1 works, including the expected output you will check>

4.1: Implement

- ☐ write your plan in comments
- ☐ implement the sub-step
- ☐ run your tests and confirm it works as expected

Sub-Step 4.2: <Description>

<description>

4.2: Plan your test

<describe how you will test if the step will work, including the expected output you will check>

4.2: Implement

- ☐ write your plan in comments
- ☐ implement this step
- ☐ run your tests and confirm it works as expected

Sub-Step 4.3: <Description>

<description>

Name:
Instructor:

4.3: Plan your test

<describe how you will test if this step works, including the expected output you will check>

4.3: Implement

- ☐ write your plan in comments
- ☐ implement this step
- ☐ run your tests and confirm it works as expected

<you may add more steps as needed>

Parse Bucknell HTML Data

Once you're certain your code works to parse your HTML files, it's time to parse the data from bucknell_web folder.

After this step is complete, your code should create a file, unl_img_per_year.csv, that has the results from parsing all the HTML files in bucknell_web folder. Inside bucknell_web are html files for the Bucknell homepage from 1996-2022. All files are named bucknell_<year>.html.

Your program should **print** an unabbreviated progress list:

```
Processing bucknell_web/bucknell_1996.html
Processing bucknell_web/bucknell_1997.html
Processing bucknell_web/bucknell_1998.html
<...>
Processing bucknell_web/bucknell_2021.html
Processing bucknell_web/bucknell_2022.html
Finished processing html files
```

The file, unl_img_per_year.csv, should have the contents similar to (with rows for all dates):

```
DATE,TOTAL_IMG,NUM_UNL,PROP_UNL
1996,6,2,0.33
1997,6,1,0.17
1998,29,1,0.03
<...abbreviated for write-up, your final file will have all rows…>
2021,40,0,0.0
2022,28,0,0.0
```

Name:
Instructor:

Implement: data parsing

- ☐ move the code for iterating through the test directory in an if DEBUG block
- ☐ copy the code for iterating through the directory and alter it to run your program and assess the full dataset of HTML files in bucknell_web from 1996-2022
- ☐ change the DEBUG variable at the top of the assess.py file to False
- ☐ run the code
- ☐ check the output file

Note: you should only need to make *very minor* changes to your code to parse every file in the bucknell_web code when DEBUG = False.

Steps 5 & 6: Graph Results

By now we should have two csv files:

- **test_img_count.csv**: contains the data from our 4 test html files
- **unl_img_per_year.csv**: data from our bucknell_web dataset. The CSV has a row for every year from 1996–2022 with the number of images,number of unlabeled images, and proportion of unlabeled images on the Bucknell home page.

We'll use those files to graph the change in the accessibility of Bucknell's homepage accessibility over time.

Understanding graph_prop_unlabeled_by_date

You're given part of the function graph_prop_unlableled_by_date. It takes as a parameter csv_filepath; a string; the filepath of data to read in. We can assume the data has the headers DATE,TOTAL_IMG,NUM_UNL,and PROP_UNL.

The code given in the function creates a scatterplot using matplotlib.

In particular, we will plot dates along the x-axis and the proportion of unlabeled images on the y-axis. The line:

```
ax.scatter(dates,prop_unl)
```

creates our graph based on two lists, dates and prop_unl.

- `dates`; list of integers; the dates of the files we're graphing
- `prop_unl`; list of floats; the proportion of images unlabeled
- `prop_unl[idx]` should be the proportion of unlabeled images for `date[idx]`.

Your task is to create the dates and prop_unl list from the data in the file given by the function's csv_filepath parameter.

Sub-Step 5.1: Read in CSV file as dictionary

In the graphing function, open the csv file at csv_filepath to read in as a dictionary of values.

Name:
Instructor:

Using the csv module, we can easily open our csv file and read in each line as a dictionary that uses the header as a key and maps to the value.

CSV Open Example

Given the file **example.csv**

```
ITEM,COLOR,COST
chair,green,10.00
chair,blue,10.00
desk,white,55.00
```

The following code reads in the file as a dictionary. Note, all values are read in as strings.

Code	**Output**

```
import csv

with open('example.csv', 'r') as f:
  # read the CSV file
  csv_dicts = csv.DictReader(f)
  # go through every row's dictionary
  for entry in csv_dicts:
     print("entry type:", type(entry))
     print("entry:",entry,"\n")
```
	entry type: <class 'dict'>
	entry: {'ITEM': 'chair', 'COLOR': 'green', 'COST': '10.00'}
	
	entry type: <class 'dict'>
	entry: {'ITEM': 'chair', 'COLOR': 'blue', 'COST': '10.00'}
	
	entry type: <class 'dict'>
	entry: {'ITEM': 'desk', 'COLOR': 'white', 'COST': '55.00'}

Worksheet: Plan your test (COMPLETE THE EXPECTED OUTPUT)

- Print the dictionary for each line in our file
- For test_img_count.csv, it will print:
  entry: {<COMPLETE>}
  entry: {<COMPLETE>}
  entry: {<COMPLETE>}
  entry: {<COMPLETE>}
- For unl_img_per_year.csv, it will print (partial expected output):
  entry: {<COMPLETE>}
  entry:{<COMPLETE>}
  …
  entry:{<COMPLETE>}

Name:
Instructor:

Implement

- ☐ add code to graph_prop_unlableled_by_date to open the csv file provided in the parameter as a dictionary
- ☐ print out each entry in that dictionary
- ☐ run the function with your test and full results csv files
- ☐ ensure the output matches your above testing plan

Sub-Step 5.2: Create dates and prop_unl list

Now use the dictionaries produced from reading in the CSV file to create the lists for dates and prop_unl.

Worksheet: Plan your test (COMPLETE THE TABLE)

For **unl_img_per_year.csv**:

Variable	**Value**
dates	[1996, 1997, 1998, 1999, 2000,...,2021,2022]
len(dates)	<COMPLETE>
type(dates[0])	<COMPLETE>
prop_unl	[0.33, 0.17, 0.03,...,0.0,0.0]
len(prop_unl)	<COMPLETE>
type(prop_unl[0])	<COMPLETE>

For **test_img_count.csv**:

Variable	**Value**
dates	[1,2,3,4]
len(dates)	<COMPLETE>
prop_unl	<COMPLETE>
len(prop_unl)	<COMPLETE>

Name:
Instructor:

Implement

- ☐ plan out your approach in comments **before** starting to code
- ☐ extend your code from part 5.1 to create and print the dates and prop_unl list
- ☐ test with both results csv files

Step 6: Graph results

Once you have successfully created the dates and prop_unl lists, the graphing function should display a scatter plot of dates versus proportion unlabeled.

Worksheet: Final Graphs

When the code runs, it will create a file and save the plot.
Paste the final graphs below

Graph for unl_img_per_year.csv:

Graph for test_img_count.csv:

Name:
Instructor:

Helpful Info

These examples can be found on replit examples.

CSV Files

A CSV (comma-separated values) file gives us a standard way to store data. We can think of it like a table. Each line in our file is a row of our table. Each column is separated by a comma.

The table

Item	**Color**	**Cost**
chair	green	10.00
chair	blue	10.00
desk	white	55.00

could be stored in a csv file, **example.csv** as

```
Item,Color,Cost
chair,green,10.00
chair,blue,10.00
desk,white,55.00
```

Test Code Blocks

Adding print statements and running our code on simple test cases help us understand our work, do incremental coding, and fix any bugs as we go. A few tips for more streamlined testing.

Work out what you expect print statements to produce

When adding print statements, it's important to think about what you expect them to produce so we can compare the results your code produces against them.

Make Informative Print Statements

As we add more print statements, it can be tricky to keep track of what each value represents. Adding additional info to our print statements help us keep track of the information we're gathering.

Here's an example to show how we can make our print statements more helpful.

Name:
Instructor:

These print statements give us helpful information but it's easy to lose track of what value is represented in the output.

Code

```
my_list = [ [ 10,15,2],
            [8, 42, 5]]

for r_idx in range(len(my_list)):
   print(r_idx)
   print(type(r_idx))
   print(my_list[r_idx])
   print(type(my_list[r_idx]))
   total = 0
   for val in my_list[r_idx]:
       total += val
       print(val)
       print(total)
   print(total)
print(total)
```

Output

```
0
<class 'int'>
[10, 15, 2]
<class 'list'>
10
10
15
25
2
27
27
1
<class 'int'>
[8, 42, 5]
<class 'list'>
8
8
42
50
5
55
55
55
```

Name:
Instructor:

A better implementation of these print statements:

Code

```
my_list = [ [ 10,15,2],
            [8, 42, 5]]

for r_idx in range(len(my_list)):
  print("\nr_idx:",r_idx)
  print("r_idx type:",type(r_idx))
  print("my_list[r_idx]:", my_list[r_idx])
  print("type of
my_list[r_idx]:",type(my_list[r_idx]))
  total = 0
  for val in my_list[r_idx]:
      print("val:",val)
      total += val
      print("tot bottom inner loop:",total)
  print("tot bottom outer loop:",total)
print("tot after for loops:",total)
```

Output

```

r_idx: 0
r_idx type: <class 'int'>
my_list[r_idx]: [10, 15, 2]
type of my_list[r_idx]: <class 'list'>
val: 10
tot bottom inner loop: 10
val: 15
tot bottom inner loop: 25
val: 2
tot bottom inner loop: 27
tot bottom outer loop: 27

r_idx: 1
r_idx type: <class 'int'>
my_list[r_idx]: [8, 42, 5]
type of my_list[r_idx]: <class 'list'>
val: 8
tot bottom inner loop: 8
val: 42
tot bottom inner loop: 50
val: 5
tot bottom inner loop: 55
tot bottom outer loop: 55
tot after for loops: 55
```

Name:
Instructor:

Debug test blocks

We may want to switch back and forth between debugging mode (where we have ample print statements and use test data) and running our code in its final mode (only necessary output, on real data). Commenting out lines of code is one technique, although can be time consuming. Another technique is to use a conditional statement and a boolean to easily toggle testing print statements.

Example of using boolean variable DEBUG and if statements to change data to test data and print debugging statements.

Code

```python
DEBUG = True
# this is our "real data"
print("Summing all values")
my_list =  [
  [10,4,203,501,2,4,482,0,19],
  [1,40,3,5,2,40,82,0,20],
  [3,4,20,77,2,4,100,0,5],
  [14,4,203,501,2,4,482,0,19]
]
# in DEBUG case,
# change data to a simpler test case
if DEBUG:
  # expect sum to be 21
  print("Debugging")
  my_list = [ [1,2,3],
            [4,5,6]]

for r_idx in range(len(my_list)):
  if DEBUG:
    print("\nr_idx:",r_idx)
    print("r_idx type:",type(r_idx))
    print("my_list[r_idx]:", my_list[r_idx])
    print("type my_list[r_idx]:",end="")
    print(type(my_list[r_idx]))
  total = 0
  for val in my_list[r_idx]:
      total += val
      if DEBUG:
        print("val:",val)
        print("tot bottom inner loop:",total)
  if DEBUG:
      print("tot bottom outer loop:",total)
print("List Total:",total)
```

Output

```
Summing all values
Debugging

r_idx: 0
r_idx type: <class 'int'>
my_list[r_idx]: [1, 2, 3]
type my_list[r_idx]:<class 'list'>
val: 1
tot bottom inner loop: 1
val: 2
tot bottom inner loop: 3
val: 3
tot bottom inner loop: 6
tot bottom outer loop: 6

r_idx: 1
r_idx type: <class 'int'>
my_list[r_idx]: [4, 5, 6]
type my_list[r_idx]:<class 'list'>
val: 4
tot bottom inner loop: 4
val: 5
tot bottom inner loop: 9
val: 6
tot bottom inner loop: 15
tot bottom outer loop: 15
List Total: 15
```

Name:
Instructor:

When DEBUG is set to false, it runs with the "real" data and fewer print statements. Depending on your needs, you may still need to comment some lines out as you go.

Code

```
DEBUG = False
# this is our "real data"
print("Summing all values")
my_list =  [
  [10,4,203,501,2,4,482,0,19],
  [1,40,3,5,2,40,82,0,20],
  [3,4,20,77,2,4,100,0,5],
  [14,4,203,501,2,4,482,0,19]
]
# in DEBUG case, use change data to a simpler
test case
if DEBUG:
  # expect sum to be 21
  print("Debugging")
  my_list = [ [1,2,3],
            [4,5,6]]

for r_idx in range(len(my_list)):
  if DEBUG:
    print("\nr_idx:",r_idx)
    print("r_idx type:",type(r_idx))
    print("my_list[r_idx]:", my_list[r_idx])
    print("type my_list[r_idx]:",end="")
    print(type(my_list[r_idx]))

  total = 0
  for val in my_list[r_idx]:
    total += val
    if DEBUG:
      print("val:",val)
      print("tot bottom inner loop:",total)
  if DEBUG:
    print("tot bottom outer loop:",total)
print("List Total:",total)
```

Output

```
Summing all values
List Total: 1229
```