Lab2:

Name1: Saffa Gbondo ([gbondos@berea.edu](mailto:gbondos@berea.edu))

Name2: Andrew Marima Mambondiumwe ([mambondiumwem@berea.edu](mailto:mambondiumwem@berea.edu))

# **INITIAL DESIGN PLAN**

Design for the modulo check character

1. Check if the length of the string is 12
2. If the length of the string is 12 continue with the program
3. If the length of the string is not 12 display error
4. Identify the numbers in the odd position
5. Add the numbers in the odd position
6. Multiply the result by 3
7. Store the result in a variable R1 or any other variable
8. Identify the numbers in the even position
9. Add the numbers in the even position except the 12th number
10. Store the result in variable R2 or any other variable
11. Sum R1 and R2
12. Store the value of R1 and R2 in T1
13. Divide T1 by 10 and find the remainder
14. If the remainder is 0, return that value
15. If the remainder is not 0,

Purpose: The primary purpose of this software is to take a string and determine if it is a valid UPC code and display the UPC code if it is valid

We will complete the task with the following functions:

* First, we need to determine if the length of the string is 12 digits. We will make a function called is\_twelve\_numbers(digit). Takes a string of numbers as input and checks that the length of the string is 12 and returns True if it is 12, but returns False otherwise. We should not continue if this fails
* We need to put the numbers in a list. The input will be the 12 digits or 12 numbers and the output will be the same numbers arranged in a list. This will be represented by the separate\_num(digit) function.
* Next, we will identify the numbers in the odd position that is the 1st, 3rd, 5th, 7th, 9th and 11th. Then we will add these numbers together and store them in a variable R1(result 1). We will return R1(result 1). The function will be sum\_odd\_num(digits). The input here is a list of 12 numbers and the output is sum of the numbers in the odd position
* We will identify the numbers in the even position of the 12 digit number. That is the 2nd, 4th, 6th, 8th, and 10th number. We will not include the 12th number. Then we will add these numbers in the even position together and store in a variable R2(result 2). We will return R2(result 2). The function will be sum\_even\_num(digits). The input here is a list of 12 numbers and the output is the sum of the numbers in the even position.
* The next function will store the value of the variables R1(result 1) and R2(result 2). We will call this function sum\_total(R1, R2). We will add R1 and R2 and store it in a variable T1. We will return T1(total result). The input will be  2 numbers and the output will be T1 which is the sum of the two numbers.
* We will create the check\_remainder function(T1)(total result).It takes a number  as input, divides it by 10 and checks if the remainder is 0. If the remainder is 0,it returns True, otherwise it Returns False.
* check\_number(T1)(total result). This function takes a number as an input. It calls the check\_remainder(T1)(total result) function.If it is True, this function returns the value 0. If it is False, subtract 10 from the remainder of T1/10  and return the answer. The output is either 0 or the value of 10-(T1%10).
* is\_valid(number). This takes a 12 digit number and checks if it is a valid UPC code by calling all the other functions. This will call the other functions. After the last function check\_number(T1) is called, it will return a check digit. This function uses the check number to determine if it is a valid UPC code.  If it is a valid UPC code, it returns True. If it is invalid it returns False. So, the output will either be True or False. It will check if it is valid by calling these functions:
* is\_twelve\_numbers(digit)
* separate\_num(digit)
* sum\_odd\_num(digits)
* sum\_even\_num(digits)
* sum\_total(R1, R2)
* check\_remainder function(T1)
* check\_number(T1)
* convert\_to\_binary(number). This function takes a 12 digit number and converts it to binary numbers. We will take numbers from 0-9 and store the binary equivalent of the left and right digits in a dictionary. The input will be 12 digit numbers and the output will be the binary equivalent of the 12 digit numbers.
* draw\_code(t, sz): This function takes binary numbers and draws them in black and white lines using the turtle module. If the number is 0, it draws a white line, if the number is 1, it draws a black line. Afterwards, it writes the corresponding 12 digit that is readable by humans under the bars.  The input will be the name of the turtle instance and the size.The output will be the barcode.
* write\_error\_message (t, sz). This function takes a turtle instance and size as input. Then it prints “Error”. The expected output is “Error”.
* main () function calls upc\_test\_suite () which tests each of the supportive function.  
  For the  program, it asks the user to input a filename. Then it checks if the filename is valid. If it is valid, it continues with the program. If it is not valid, it displays error and asks the user to input the correct file name. Then it continues with the program by calling the following functions:

is\_valid(number)  
 convert\_to\_binary(number)

If it is a valid number, then draw\_code(t, sz) function is called. If it is not a valid function, write\_error\_message (t, sz) function is called.

def main():

main()

# **SUMMARY**

Initially our design was made to take an input from the user. However, in our final algorithm, we made a function that will read the file that contains the 12 upc-code digits.

We created a function that takes an inputed file name as a parameter and then reads the file content of the file. It then returns the content as a tuple. The function only takes input of a file that exist in the file folder. It does not take in any file name that does not exist in the file folder.

For the convert to binary, we used 2 dictionaries; one for the left half of the upc-code and the other for the right half of the upc-code.

In the initial design, the function was supposed to calculate the total of the even numbered indexes and the odd numbered indexes and then returns the total value. Then after that we designed another function that will used that total value to check if it gives a remainder of 0 when divided by 10. If the remainder is not 0 it subtracts the remainder from 10

In the implementation however, we used one function that first calculates the total for even and odd numbered indexes and then uses that total to check for the modulo.

Overall we spent like 10 hours (2 hours a day) from the design to the implementation of the program and making it to be a workable program.

# **IMPLEMENTATIONS**

We were able to successfully achieved our initial design which is written below

We were able to:

1. Check if the length of the string is 12
2. If the length of the string is 12 continue with the program
3. If the length of the string is not 12 display error
4. Identify the numbers in the odd position
5. Add the numbers in the odd position
6. Multiply the result by 3
7. Store the result in a variable R1 or any other variable
8. Identify the numbers in the even position
9. Add the numbers in the even position except the 12th number
10. Store the result in variable R2 or any other variable
11. Sum R1 and R2
12. Store the value of R1 and R2 in T1
13. Divide T1 by 10 and find the remainder
14. If the remainder is 0, return that value
15. If the remainder is not 0,

# **TESTING:**

* We will write a unit test for all the above so that we can use test driven design as we build out. For this we will need two functions: testit(did\_pass) to run each unit test and upc\_test\_suite () to contain the test themselves.  Hence upc\_test\_suite () will be designed to test the following:  
  (R1=result1, R2=result2, T1=Total1)
  + is\_twelve\_numbers(digit)
  + separate\_num(digit)
  + sum\_odd\_num(digits)
  + sum\_even\_num(digits)
  + sum\_total(R1, R2)
  + check\_remainder(T1)
  + check\_number(T1)
  + is\_valid(number)

The inputs will vary. For is\_twelve\_numbers(digit) function, the input will be different 12-digit numbers and the answer would be True. That is we are checking if True is returned for each case. This is the expected case. For the unexpected case, we will have inputs of 5-digit numbers, 3-digit numbers, 9-digit numbers, 15-digit numbers, and 13-digit numbers. The answer would be False. That is we would be checking if False is returned for each of these. For separate\_num(digit) the input we will use different 12-digit numbers to check if the function correctly splits it into a list with the same numbers. For the unexpected cases, we will test other number lengths, like 2 digits, 10 digits, 5 digits, and 20 digits,  because the function will be designed to separate or split numbers of any length and put them into list. When testing the sum\_odd\_num(digits), for the expected cases, we will test different 12-digit numbers to see if it correctly returns the sum of the numbers in the odd position. For the unexpected cases, we will test 3-digit numbers, 9-digit numbers, 13-digit numbers, and 14-digit numbers, because the program will be designed to sum up all the numbers in the odd-position no matter the length of the numbers. When testing sum\_even\_num(digits) function, for the expected cases, we will test to see if the program correctly adds the numbers in the even position of different 12-digit numbers. For the unexpected case, we will will test 14-digit and  16-digit numbers to make sure the program correctly skips the 12th position.  While checking sum\_total(R1, R2) function, for the expected case we will test for the sum of different single-digit numbers. For the unexpected case, we will test for the sum of different  2-digit and 3-digit numbers. For check\_remainder function(T1) function, for the expected cases we will test different 2-digit numbers and for the unexpected case we will check different one-digit and three-digit numbers to see if it returns True if the remainder is 0 and False otherwise. When testing check\_number(T1) function, we will use different 2-digit numbers for the expected case, and different single-digit and three-digit numbers for the unexpected cases. If the function works well the output should either be 0 if T1 is divisible by 10 or 10-(T1/10) if T1 is not divisible by 10. When testing  is\_valid(number), for the expected case we will test different 12-digit numbers.Some of the 12-digit numbers we will test will be valid UPC codes which should return True and others will be invalid UPC codes which will return False. For the unexpected cases, we will try different numbers that are not 12-digit numbers. In this case there should be an Error.

Everything described above is the test suite that is contained in the upc\_test\_suite () function. This test each of the supportive functions. This does not have its own output. The results of the test are displayed by testit(did\_pass) function. There is no way of testing draw\_code(t, sz) and write\_error\_message (t, sz) functions. If convert\_to\_binary(number) works then it should work.

* testit(did\_pass) function takes the different functions, the inputs of those functions and the expected result as input. This function prints the result of the unit test as output.

# **FILES:**

The following were the files used:

* upc-input1.txt
* upc-input2.txt

# **ERRORS:**

The following were the errors we encountered in the lab:

* Attribute error: some of our modules attribute were not recognized.
* Global variable/local variable not defined

There were other list of errors that we experienced which includes syntax error and run time errors.

# **COMMENTS:**

The tricky part was getting the turtles to draw the barcode. We also had difficulties in writing the UPC code on the screen with the appropriate spacing that is required.

Overall, the lab was challenging but it improved our knowledge. We were also able to practice using turtles. Finally, the we felt joy and happiness when we were able to draw the barcode.