# CSE 101: Computer Science Principles (Fall 2019)

## Lab #4

**Assignment Due: Saturday, October 12, 2019, by 11:59 PM**

## Assignment Objectives

By the end of this assignment you should be able to develop original Python functions to solve simple programming problems involving loops and strings.

## Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct (**note that the test cases we give you in the starter files are just examples; we will use different test inputs to grade your work!**). You need to complete (fill in the bodies of) these functions for the assignments. **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change any function headers, the grading program will reject your solution and mark it as incorrect.

## Directions

Solve each of the following problems to the best of your ability. We have provided a code skeleton for each of the programming problems. The automated grading system will execute your solution to each programming problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 3 problems, and is worth a total of 30 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your work in a single file. If you do so, we may not be able to grade your work and you will receive a failing grade for this assignment!
- ⚠ Every function **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Every function must explicitly ***return*** its final answer; the grading program will ignore anything that your code prints out. Along those lines, do **NOT** use `input()` anywhere within your functions (or anywhere before the `if __name__ == "__main__":` statement); your functions should get all of their input from their parameters. Programs that crash will likely receive a failing grade, so test your code thoroughly **with Python 3.7.4 or later** before submitting it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your completed work as directed by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time, or that is submitted before you have signed the course's Academic Honesty agreement.
- ⚠ **ALL** of the material that you submit (for each problem) **MUST** be your own work! You may not receive assistance from or share any materials with anyone else, except for the instructor and the (current) course TAs.

## Part I: Martian Math (10 points)

(Place your answer to this problem in the "martian.py" file)

Martian integer multiplication works as follows (on paper):

1. Write the two integers to be multiplied side by side, at the top of two columns.
2. Divide the number on the left by 2, ignoring any remainder, and double the number on the right.
3. Repeat step 2 for the newly-created numbers, until the number on the left reaches 1.
4. Each time you divide, if the number on the left of that row is odd, add the number on the right to your total.
5. When you are done, the total will be equal to the product of the original two numbers.

For example, suppose that we want to multiply 22 by 4:

```
22  4
11  8
5   16
2   32
1   64
```

The second, third, and fifth rows all have an odd number on the left. If we add up the values on the right of those rows, we get 8 + 16 + 64, which equals 88 (the product of 22 and 4).

Complete the `martian()` function, which takes two (positive, non-zero) integer values to multiply. This function counts and returns the number of times that the left column value is odd, **NOT** the final product. For example, using the example above, `martian(22, 4)` would return 3, not 88 (since there are three rows with odd values on the left).

**Examples:**

| Function Call | Return Value |
|---|---|
| `martian(22, 4)` | 3 |
| `martian(15, 2)` | 4 |
| `martian(103, 5)` | 5 |
| `martian(768, 21)` | 2 |

## Part II: Friendly Numbers (10 points)

(Place your answer to this problem in the "friendly.py" file)

An integer is described as *friendly* if every sequence of *n* digits (counting from the left) is evenly divisible by *n*. In other words, the leftmost digit is evenly divisible by 1,[1] the leftmost (or first) two digits are evenly divisible by 2, and so on ("evenly divisible" means that the remainder is 0).

For example, 42325 is a friendly number:

- 42325 is evenly divisible by 5
- 4232 is evenly divisible by 4
- 423 is evenly divisible by 3
- 42 is evenly divisible by 2
- 4 is evenly divisible by 1

Complete the `friendly()` function, which takes two integer arguments, in the following order: a number to examine, and the total number of digits in that value (for example, if the first argument was a 9-digit number, the second argument would be 9). The function returns the Python value `True` (**NOT** the string "True") if the first argument is friendly, and `False` otherwise.

**HINT:** Use a loop to count down from the starting number of digits to 1. Use floor division and the modulo operator to remove one digit at a time from the (right) end of the number as you test different numbers of digits.

**Examples:**

| **Function Call** | **Return Value (NOT a string!)** |
|---|---|
| `friendly(42325,5)` | `True` |
| `friendly(82736451,8)` | `False` |
| `friendly(18,2)` | `True` |
| `friendly(497,3)` | `False` |

[1] This is trivially true for every integer.

## Part III: Binary Addition (10 points)

(Place your answer to this problem in the “add_binary.py” file)

**Sorry! The directions below are LONG, but that’s because they are meant to be VERY thorough.**

In lecture, we discussed binary (base 2) representation for numbers. We can perform arithmetic (like addition) on numbers in any base. In fact, binary addition is simpler than base 10 addition because there are fewer rules. Like base 10 addition, binary addition works from right-to-left, adding one pair or column of digits at a time, according to the following rules:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0, plus a carry bit of 1 (for the next column) — recall that 1 + 1 = 2, and 2 is represented as 10 in binary

If you have three 1s to add (1 + 1, plus a carry bit of 1 from the previous column), the sum is 1, with a carry bit of 1 (1 + 1 + 1 = 3, and 3 is represented as 11 in binary).

For example, consider the following addition problem (11 + 11, in base 10):

```
 1011
+1011
-----
```

We start by adding the rightmost column. 1 plus 1 gives us 0, plus a carry bit of 1:

```
   1
 1011
+1011
-----
    0
```

The second column has two 1s, plus a carry bit of 1. 1 + 1 + 1 equals 1, plus a new carry bit of 1:

```
  1
 1011
+1011
-----
   10
```

The third column (counting from the right) has a 1 (the carry bit) plus two 0s. This adds up to 1 (with nothing to carry to the next column, which is effectively a carry bit of 0):

```
 1011
+1011
-----
  110
```

Finally, the fourth (leftmost) column has two 1s (and an implied carry bit of 0). 1 plus 1 equals 0, with a carry bit of 1:

```
1
 1011
+1011
-----
 0110
```

There are no columns left to process, but we have a non-zero carry bit, which is placed on the left of the existing result, giving us a final answer of 10110 (which is 22 in base 10).

For this problem, we will add two binary numbers, but in reverse order. That is to say, the digits (bits) of each binary number will be represented in opposite order from normal. For example, the binary value 1011 (which is 11 in base 10) will be represented as 1101. This will make the addition process slightly easier for us to perform; when we're finished, we'll just reverse the order of the bits in the result to get the answer with the bits in the correct order.

**Note:** This problem uses strings, which we will discuss in class this week. All of the information you need to complete this problem is described below, but you can also refer to the lecture slides on strings (or wait for class on Monday) to make sure that you understand the string concepts that this problem requires.

Complete the `add_binary()` function, which takes two strings that represent binary numbers (with the bits already in reverse order); each string only contains the characters '0' and '1', and both strings are of equal length. The function should use the procedure above to create a new string that represents the sum of the two parameters. Finally, it should return that string with the characters in reverse order. **DO NOT convert the input to base 10, add them, then convert back to binary!**

The trick here is to use a loop that counts from 0 up to the length of one of the numbers (it doesn't matter which one; they're both the same length). Each round of the loop examines the bits at the same position (index) in both numbers (for example, one round of the loop might examine the third bit in both numbers; the next round would examine the fourth bit in both of the numbers, and so on). For each position, use a set of `if` statements (or something similar) to decide what you should add to the result, and what you should carry over to the next column (if necessary). At worst, this requires eight `if` statements (two for each of the rules above, depending on the value of the carry bit). For example:

```
if first[index] == "0" and second[index] == "0" and carry == "0":
    # add 0 to 0 (no carry bit)
    sum = sum + "0"                 # 0 + 0 (+ 0) is 0
    carry = "0"                     # don't carry anything to the next column
elif first[index] == "0" and second[index] == "0" and carry == "1":
    # add 0 + 0 (plus 1 for the carry bit)
    sum = sum + "1"                 # 0 + 0 + 1 is 1
    carry = "0"                     # reset carry bit to 0
...
```

1. Start by using the `len()` function to determine how many characters (bits) are in one of the parameters. For example:

   `number_of_bits = len(first)`

2. Create two string variables: one for the in-progress sum, and one for the current value of the carry bit. Initialize the sum variable to the empty string (just a set of empty quotation marks), and the carry bit variable to the string "0".

3. Use a loop that counts from 0 up to (but not including) the total number of bits (0 is the index of the first character in a string). You can use either a `while` loop or a `for` loop for this. Place your set of `if` statements (described above) inside this loop; each `if` statement should use the + operator to add a new digit (in quotes; it's a string) to the end of the sum variable and set the value of the carry bit variable appropriately. Use the loop variable to retrieve the character/bit at a given position in each parameter string by placing it in square brackets after the name of the string variable. For example, if `data` is a string, `data[p]` returns the character at index (position) `p`.
4. After your loop ends, if the carry bit is "1", add it to the end of your sum variable.
5. Finally, it's time to reverse the sum to put the bits back into the proper order. You can do this by using the *slicing*[2] operator `[::-1]`. For example, if `name` had the value `"lizard"`, `name[::-1]` would produce `"drazil"`. Don't forget to use **TWO** colon characters inside the brackets!

**Examples:**

| **Function Call** | **Return Value (a string)** | **Notes/Explanation** |
|---|---|---|
| `add_binary("011", "001")` | `1010` | 011 (reversed) is 6; 001 (reversed) is 4; 6 + 4 is 10 (1010 in binary) |
| `add_binary("1101", "0111")` | `11001` | 1101 (reversed) is 11; 0111 (reversed) is 14; 11 + 14 is 25 (11001 in binary) |
| `add_binary("100", "011")` | `111` | 100 (reversed) is 1; 011 (reversed) is 6; 1 + 6 is 7 (111 in binary) |
| `add_binary("1100111", "0111101")` | `11010001` | 1100111 (reversed) is 115 in binary; 0111101 (reversed) is 94 in binary; 115 + 94 is 209 (11010001 in binary) |

[2] We'll cover slicing in class on Monday, or Wednesday at the very latest.