CSC338 Assignment 2.

Due Date: Monday, May 25, by 11:59pm.

What to Hand In

Please hand in 3 files:

- Python File containing all your code, named `hw2.py`.
- PDF file named `hw2_written.pdf` containing your solutions to the written parts of the assignment. Your solution must be prepared in LaTeX. No handwritten solutions with be accepted.
- TeX file named `hw2_written.tex` containing latex source code. We will not accept for PDF if there is no latex source code submitted.

Your code will be auto-graded using Python 3.8, so please make sure that your code runs. **There will be a 20% penalty if you need a remark due to small issues that renders your code untestable.**

Submit the assignment on **MarkUs** by 11:59pm on the due date. See the syllabus for the course policy regarding late assignments. All assignments must be done individually.

```
import math
import numpy as np
```

Question 1

.

We'll be implementing a floating point system. For most of this question, assume that we are working with:

- base
  $\beta = 3$,
- precision $p = 5$, and
- exponent range $[L, U] = [-3, 3]$

Our floating point system will be normalized.

```
# These are the constants we'll use in our code
BASE = 3
PRECISION = 5
L = -3
U = +3
# We will represent a floating number as a tuple (mantissa, exponent, sign)
# With: mantissa -- a list of integers of length PRECISION
#       exponent -- an integer between L and U (inclusive)
#       sign -- either 1 or -1
example_float = ([1, 0, 0, 1, 0], 1, 1)
```

(a) – 1pt

How many normalized-point numbers are in our system? Use a formula in terms of `BASE, PRECISION, L and U` to compute the count, and save the result in the value `total_num_floats`.

```
total_num_floats = None
```

(b) – 3pt

Write a function `is_valid_float` that checks whether a tuple of the form `(mantissa, exponent, sign)` represents a valid, normalized float in our floating point system.

```
def is_valid_float(float_value):
    """Returns a boolean representing whether the float_value is a valid,
    normalized float in our floating point system.
    >>> is_valid_float(([1, 0, 0, 1, 0], 1, 1))
    True
    """
    (mantissa, exponent, sign) = float_value
    return None
```

(c) – 3pt

Construct a floating-point representation tuple of the form (mantissa, exponent, sign) of each of the following:

(a) The largest negative number representable in our floating point system. Store it in the variable `largest_negative`.

(b) The smallest positive number (greater than 0) in our floating point system. Store it in the variable `smallest_positive`.

(c) The value of 32 represented in our floating system. Assume chopping is used for rounding. Store it in the variable `float_32`.

```
largest_negative = ([], None, None)
smallest_positive = ([], None, None)
float_32 = ([], None, None)
```

(d) – 5pt

Write a function `to_num` that converts a tuple of the form `(mantissa, exponent, sign)` to a Python numerical value.

```
def to_num(float_value):
    """Return a Python floating point representation of ‘float_val‘
    These examples are for your understanding, and your actual output
    might vary slightly.
    >>> to_num(example_float)
    3.111111111111111
    """
(mantissa, exponent, sign) = float_value
return None
```

(e) – 5pt Write a function add that takes two tuples of the form `(mantissa, exponent, sign)`, and performs floating-point addition to obtain a third floating-point number in our representation `(mantissa, exponent, sign)`. You may assume that the signs of both addends are positive.

```
def add_float(float1, float2):
    """Return a valid floating-point representation of the form (mantissa, exponent, sign)
    that is the sum of ‘float1‘ and ‘float2‘. Raises a ValueError if the result of
    the addition is not a valid float.
```

```
        >>> add_float(example_float, example_float)
        ([2, 0, 0, 2, 0], 1, 1])
        """
        (mantissa1, exponent1, sign1) = float1
        (mantissa2, exponent2, sign2) = float2
        # You may assume that sign1 and sign2 are positive
        assert (sign1 == 1) and (sign2 == 1)
        # Add your code here
        return (None, None, None)
```

(f) – 2pt Show that `add_float(x,y)` is not associative. Include this part of your work in your PDF file.

Question 2.

– 4pt

We are going to show catastrophic cancellation by computing $\frac{1}{1-x}$ and e^x usting their Taylor series expansion.

$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n = 1 + x + x^2 + \ldots$ for $x \in (-1, 1)$ and
$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots$

The functions `h1` and `h2` returns a list of the first n elements of the Taylor series expansions of $\frac{1}{1-x}$ and e^x respectively.

```
def h1(x, n):
    """Returns a list of the first n terms of the Taylor Series expansion of 1/(1-x)."""
    return [pow(x,i) for i in range(n)]

def h2(x, n):
    """Returns a list of the first n terms of the Taylor Series expansion of e^x."""
    return [pow(x,i)/math.factorial(i) for i in range(n)]
```

Does computation of $\frac{1}{1-x}$ and e^x suffer from catastrophic cancellation? If so, chose appropriate values of x and show a sample of both lists. If no, briefly explain why not. Include your solution in the PDF writeup.

Question 3.

Consider the function `z(n)` below:

```
def z(n):
    a = pow(2.0, n) + 10.0
    b = (pow(2.0, n) + 5.0) + 5.0
    return a - b
```

(a) –1pt

Using Python, find all positive integers n for which the value of `z(n)` is nonzero. Save the result in a list called `nonzero_zn`.

```
nonzero_zn = []
```

(b) –4pt

Using what we learned about floating-point addition and the rounding rules, explain why the $z(n)$ takes on these particular non-zero values. Why is the expression zero for all other values of n? You may assume that Python uses IEEE Double Precision for floating-point arithmetic (i.e., round to even in base 2 with a mantissa of 53 bits). Hint: look at the rounding error produced by each of the three additions.