**MSCF Python Programming Basics**

**Homework 1**

***Due At 11:59 pm US Eastern Daylight Time,***

***Tuesday, July 6, 2021***

**100 Points Total**

The program file **hw1.py** on Canvas contains test code for all of the functions you are expected to write for Homework 1. Add each function definition to the program file where indicated, then execute the corresponding test code.

1. A trivial example: **mean\_of\_3** function and testing code

Here, the instructions ask you do define a **mean\_of\_3** function, which computes and returns the mean of three argument values. In this case, we have defined the function for you.

Below the place where you are asked to define the function is testing code that calls **mean\_of\_3**. As we discussed, text from # to the end of the current line is a comment, and is ignored by the Python shell.

But what if you want to comment out a long block of lines of code? It is annoying, time-consuming, and error-prone to put a **#** in front of each line, and annoying, time-consuming, and error-prone to remove these **#** characters again later on (unless your Integrated Development Environment does this for you!).

So a common trick is to use triple-quoted strings as comments: place a triple-quote above the block of lines, and another triple-quote below the block of lines. This works *unless* the block of lines contains a triple-quoted string using the same style of quotes (single- or double-quotes). You can easily restore the block of code by commenting out the initial and final triple-quotes using the **#** character.

We have illustrated this idea around the testing code for **mean\_of\_3**. Please examine and execute this initial **hw1.py** program file.

1. Fibonacci numbers: **Fib** function

We defined a naïve **factorial** function and a more robust **factorial** function in the Week 1 lecture. Given some non-negative integer, **n**, as an argument, these functions return the value **n!**. Factorials occur often in probability and statistics.

Define a function **Fib** that takes a non-negative integer, **n**, as an argument, and that returns the **n**th Fibonacci number. For the two special cases (or initial conditions) **0** and **1**, **Fib(0)** should return **0** and **Fib(1)** should return **1**. Otherwise, the value of **Fib(n)** should be **Fib(n-2)** + **Fib(n-1)**.

Define **Fib** using a **while** loop, *not* recursion.

Restore the **Fib** testing code to the program by using **#** to comment out the triple-quotes at its beginning and end of the testing code. Save your program and execute. The output should be correct values, or meaningful error messages for invalid arguments.

1. The modulus (**%**), floor divide (**//**), and unary **+** and **–** operators

In the Week 1 Lecture Slides, we described five arithmetic operators: **\*\*** (exponentiation), **\*** (multiplication), **/** (division), **+** (addition), and **–** (subtraction). These are all binary operators: they each take two operands, one on the left and one on the right.

The **%** (modulus) and **//** (floor division) binary operators have the same precedence as the **\*** and **/** operators.

*m* **%** *n* yields the non-negative remainder after division of *m* by *n*; for example: **13 % 5 == 3**, **24 % 6 == 0**, **-3 % 4 == 1**.

Somewhat strangely, in Python this is also defined for **float** values as well as **int** values; for example: **5.1 % 2.5 == 0.09999999999999964** (notice the unfortunate roundoff problem).

*m* **//** *n* yields the integer part of the division of *m* by *n*; for example: **13 // 5 == 2**, **24 // 6 == 4**, **-3 // 4 == -1**. In Python, this is also defined for **float** values; for example, **5.1 // 2.5 == 2.0**. Unlike **/** (ordinary division), **//** yields an **int** value if both operands are **int**s.

For any numeric *m* and *n* (*n* != 0), it is true (within the precision of the data types!) that: **(***m* **//** *n***) \*** *n* **+** *m* **%** *n* **==** *m*

(Part 2, continued)

Python also provides *unary* **+** and *unary* **-**. **+(***expr***)** yields the value **(***expr***)**, and

-**(***expr***)** yields the negative of **(***expr***)**. In practice, unary **+** is almost never used; unary **-** is frequently used. **-5** is the unary **-** applied to **5**, and **-(x + 3)** is the negative of the quantity **(x + 3)**.

The arity, precedence, and associativity of these nine arithmetic operators are, from highest precedence to lowest:

\*\* binary right-to-left

+ - unary

\* / % // binary left-to-right

+ - binary left-to-right

*Quick check:* What is the value of: **-7 \*\* 2**

Define these functions:

**is\_even** that takes an **int** as argument and returns **True** if that **int** is even, otherwise **False**.

**is\_odd** that takes an **int** as argument and returns **True** if that **int** is odd, otherwise **False**.

**is\_div\_by\_n** that takes two **int** arguments (call them *m* and *n*) and returns **True** if *m* is evenly divisible by *n*, otherwise **False**.

**neg\_of** that takes a numeric argument (an **int** or **float**) and returns the negative of that argument.

Restore the Part 2 testing code to the program by using **#** to comment out the triple-quotes at its beginning and end of the testing code. Save your program and execute. The output should be correct values, or meaningful error messages for invalid arguments.

1. Sums of integers

Define these functions:

**sum\_of\_n** that takes a non-negative integer (say, *n*) as its argument, and that returns the value of the sum 0 + 1 + … + *n*.

**sum\_of\_n\_sqr** that takes a non-negative integer (say, n) as its argument, and that returns the value of the sum 0 + 1 + 4 + 9 + … + *n*2.

Restore the Part 3 testing code to the program by using **#** to comment out the triple-quotes at its beginning and end of the testing code. Save your program and execute. The output should be correct values, or meaningful error messages for invalid arguments.

1. Type conversion among low-level data types

You can convert among the low-level data types using conversion functions named the same as the types:

**int(***x***)** # convert *x* to **int** if possible

**float(***x***)** # convert *x* to **float** if possible

**str(***x***)** # convert *x* to **str** (almost always possible!)

**bool(***x***)** # convert *x* to **bool** (almost always possible!)

Converting a **float** to **int** truncates the fraction toward 0:

**int(3.3) == 3**

**int(-3.3) == -3**

This is in contrast to floor division which “truncates” toward negative infinity:

**3.3 // 1 == 3.0**

**-3.3 // 1 == -4.0**

A very large magnitude **float** will typically “convert” to an **int** containing mostly garbage digits. For example:

**10 / 3 \* 10 \*\* 100 == 3.3333333333333335e+100**

This has a roundoff error in the last digit, and conversion to **int**

**int(3.3333333333333335e+100) == 33333333333333335158542**

**29847581664884084396642883020906087565122725247674177**

**6210028647756756328382464**

compounds the silliness.

Converting an **int** to a **float** works fine *if* the **int** value is within the range that will fit in a 64-bit **float**. (You could equally well perform this conversion by adding 0.0, multiplying by 1.0, or dividing by 1.)

**float(123456789) == 123456789.0**

**float(1234 \*\* 1234)** is an OverflowError

(Part 4, continued)

Conversions from strings (**str**) to numeric values generally work “as expected:”

**int('1234') == 1234**

**float('12.34') == 12.34**

**int('12.34')** is a ValueError

**float('12.34 hello')** is a ValueError

Conversions *to* **bool** are fairly straightforward: **0**, **0.0**, **''** (the empty string), and **None** convert to **False**, anything else converts to **True**.

Conversions *from* **bool** to **int**/**float** are likewise straightforward: **True** converts to **1**/**1.0** and **False** converts to **0**/**0.0**. In fact bool values can be used in arithmetic expressions directly:

**True + True \* False / True == 1.0**  # division yields **float**

Restore the Part 4 testing code to the program by using **#** to comment out the triple-quotes at its beginning and end of the testing code. Predict what output should be produced by each print function call. Save your program and execute, to see whether you were correct. If necessary, comment out any print function calls that produce errors that halt the program. (You don’t need to enter your predictions into your code file for this part.)

1. Experiment with Spyder and PyCharm

Try using Spyder to edit and execute your code.

Install and try using PyCharm to edit and execute your code.

Do you prefer IDLE, Spyder, or PyCharm? Add your opinions to the code file that your team turns in.

***And Finally***

Submit your homework electronically to Canvas. Your team should submit a single **.zip** file, containing just your **hw1.py** file.