CSE 101: Introduction to Computational and Algorithmic Thinking

Lab #1

Fall 2018

Assignment Due: September 14, 2018, by 11:59 pm

Assignment Objectives

By the end of this assignment you should be able to design, code, run and test original Python functions that solve simple problems involving arithmetic and `if` statements.

Getting Started

The assignments in this course require you to write Python code to solve computational problems. To help you get started on each assignment, we will give you a "bare bones" file with a name like `lab1.py`. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You will fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `lab1.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

- Solve the following problems to the best of your ability. This assignment has a total of three problems, worth 12 points in all. The automated grading system will execute your solution to each problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 point. You must earn at least 8 points in order to pass (receive credit for) this lab.
- At the top of the `lab1.py` file, include the following information in comments, with each item on a separate line:
  - your first and last name as they appear in Blackboard
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Lab #1)

⚠ Your functions must be named as indicated below. Submissions that have the wrong function names can't be graded by our automated grading system.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Part I: Payroll Calculations (4 points)

Complete the `net_pay()` function, which takes four numerical arguments:

1. The total number of hours worked by a particular employee this week
2. The hourly pay rate for that employee (i.e., how many dollars per hour that employee earns)
3. The federal tax withholding rate (a value between 0.0 and 1.0). E.g., 0.20 represents a rate of 20%
4. The state tax withholding rate (a value between 0.0 and 1.0). E.g., 0.09 represents a rate of 9%

The `net_pay()` function calculates and returns the employee's net take-home pay for the week. This consists of the employee's base pay (the hours worked times the hourly pay rate), minus any federal withholding and state withholding. For example, suppose that an employee works 10 hours this week at an hourly pay rate of $9.75 per hour, with a federal withholding rate of 20% (0.2) and a state withholding rate of 9% (0.09). The employee's base pay is equal to 10 times 9.75, or $97.50. The federal withholding is 20% of that value, or $19.50. The state withholding is 9% of the base pay, or $8.775 (**DO NOT** worry about rounding the result). The employee's final pay is equal to $97.50 - $19.50 - $8.775, or $69.225 (again, don't apply any rounding for this problem).

Examples:

Function Call	Return Value
`net_pay(25, 10, .15, .10)`	`187.5`
`net_pay(40, 8, .25, .25)`	`160.0`
`net_pay(35, 20, .2, .09)`	`497.0`

Part II: Linear Motion (4 points)

If an object moves in a straight line at a given initial velocity v_i and constant acceleration a for t units of time, its final displacement *s* (the total distance moved) can be calculated using the formula

$$s = v_i(t_f - t_i) + \frac{1}{2}a(t_f - t_i)^2$$

where t_i and t_f represent the starting and ending times of the movement, respectively (i.e., if t_i was 3 and t_f was 7, then the object was in motion for a total of $7 - 3$ or 4 time units).

Complete the `displacement()` function, which takes four numerical arguments, in order:

1. the object's initial velocity v_i
2. the object's acceleration a, which can be either positive (acceleration) or negative (deceleration)
3. the time (t_i) at which the object began to move
4. the time (t_f) at which the object stopped moving

This function calculates and returns a numerical value corresponding to the object's total displacement s, using the formula given above. You may assume that t_f is always strictly greater than t_i.

Examples:

Function Call	Return Value
`displacement(5, 2, 1, 7)`	`66.0`
`displacement(3, 1, 4, 20)`	`176.0`
`displacement(18, -3, 2, 6)`	`48.0`

Part III: Calculating Stardates (4 points)

Star Trek represents time with the concept of a *stardate*, rather than our current month/day/year notation. Several people have attempted to develop algorithms for converting Gregorian dates into their corresponding stardates; we will use the formula from https://www.wikihow.com/Calculate-Stardates, simplified to ignore leap years. Note that this formula ***ONLY*** works correctly for years after 2323!

The formula to calculate a stardate is

$$(1000 * (y - b)) + (\frac{1000}{n} * (m + d - 1))$$

where:

- *y* represents the 4-digit Gregorian year to be converted to a stardate (e.g., 2374)
- *b* represents the *standard base year*, which is always 2323 for our purposes
- *n* represents the number of days in the year. For now, we are ignoring leap years, so *n* will always be 365.
- *d* represents the day of the month.
- *m* represents the *month number* (the number of days in the year prior to the start of that month). For example, January's month number is 0, since there are 0 days between the beginning of the year and January 1. For a standard 365-day year, use the following month number values:

Month	Month Number (m)		Month	Month Number (m)		Month	Month Number (m)
January (1)	0		May (5)	120		September (9)	243
February (2)	31		June (6)	151		October (10)	273
March (3)	59		July (7)	181		November (11)	304
April (4)	90		August (8)	212		December (12)	334

For example, suppose that we want to convert the Gregorian date July 12, 2467 into a stardate using the formula above. *y* is 2467, representing the year to convert. *b*, the base year, is 2323. *n*, the number of days in a year, is 365. July's month number (*m*) is 181 according to the table above. *d*, the day number, is 12. Plugging these values into the formula given above, we get $((1000 * (2467 - 2323)) + ((1000/365) * (181 + 12 - 1))$, which comes out to 144526.02739726 (stardates are normally rounded to two decimal places, but we will skip that step for now).

Complete the `stardate()` function, which takes three arguments in the following order:

1. an integer between 1 and 12 inclusive, representing the month (January is 1, December is 12).
2. an integer between 1 and 31 inclusive representing the day of the month. Assume that this value is always appropriate for the chosen month, e.g., if the month is 2 (February), this value will never exceed 28.
3. an integer representing a 4-digit year (2323 or greater).

Your function should use an `if-elif-else` chain to determine the proper value of the month number *m*. You may find it helpful to preset *b* to 2323 and *n* to 365, since those values will never change in our simplified formula.

The table below shows the expected result of calling this function with several sets of possible input values.

Examples:

Gregorian Date	Function Call	Return Value
February 21, 2365	`stardate(2,21,2365)`	`42139.72602739726`
November 29, 2401	`stardate(11,29,2401)`	`78909.5890410959`
May 4, 2390	`stardate(5,4,2390)`	`67336.98630136986`