CS1026B Assignment 1: Calculators

Due: **Tuesday, January 20th, 2026 @ 11:59 PM ET**

Change Log

- 2026-01-05 Initial post, no changes yet.

Table of Contents

Late Penalty

> ❗ Late submissions will be given a **zero grade** if you have insufficient late coupons remaining.

> ⚠ Any resubmissions past the due date will be considered late and will require late coupons.

Assignments will be accepted up to 3 days late, but **only if you have at least one late coupon remaining for each day late** (see the course syllabus for full details on how late coupons work).

If you submit late and do not have enough late coupons remaining, a **zero grade** will be given for the assignment (there is no partial late mark).

No action is required to submit late. Simply submit late and your late coupons will be applied by the TA during marking.

▲ Back to Top

⚖ Academic Integrity

By submitting this assignment, you agree that you have read and understood the academic dishonesty policy for this course covered in both the course syllabus and the academic dishonesty document. This includes:

- Assignments must be completed individually, without any help from others.
- No copying solutions, formulas, files, or written answers from other students.
- No sharing of any part of your work with other students.
- No unauthorized use of generative AI tools (e.g. ChatGPT, Copilot, etc.). You can not use generative AI for this assignment.

> ℹ Suspected violations will be reported to the integrity committee and receive a zero grade.

▲ Back to Top

Learning Objectives

1. Use variables to store and update values (`ints`, `floats`, and `strings`).
2. Read user input with `input()` and convert types when needed (e.g. `int()`, `float()`).
3. Produce correct output with `print()` that matches a required format exactly (including spaces, capitalization, and line breaks).
4. Perform basic arithmetic correctly using `+`, `-`, `*`, `/`, `//`, `%`, and `**`, and choose appropriate numeric types (`int` vs `float`) for the task.
5. Build simple string outputs using concatenation.
6. Submit to Gradescope successfully: upload the correct `.py` file, interpret autograder feedback, and resubmit after fixing issues.
7. Demonstrate basic code quality: meaningful variable names, basic whitespace/indentation, and comments.

▲ Back to Top

Assignment Instructions

> Make sure that you read over the Non-Functional Specification before you start coding. It contains requirements for comments, variable names, and style you need to follow.

For this assignment you will be creating ***three*** separate Python programs named:

- `weather.py` (based on Part 1)
- `download.py` (based on Part 2)
- `sms.py` (based on Part 3)

Each part of this assignment will provide a specification for what these programs will do, the input they will take, and the output they will provide. To obtain full marks from the autograder, you are required to follow the specification and example input/output exactly.

Keep in mind that **you can resubmit your code to the Gradescope autograder an unlimited number of times** leading up to the due date. Don't be afraid to resubmit your code often to test it, but we also highly recommend testing your code locally. Only the most recent resubmission

will count for marks, **you must ensure you submit all three programs in your final submission**, any missing programs will receive a zero grade.

Introduction

In this assignment you'll build three small "calculator" programs. They're simple on purpose. The point is not to do advanced math, it's to practice the core skills that every useful program needs: reading input, storing values in variables, doing basic arithmetic, and printing results clearly.

Each calculator takes information from the user, performs a few computations, and prints a result that someone could actually use. Think of these as tiny tools: short, focused, and reliable. Along the way you'll also learn the practical workflow of writing a Python program, running it to test your logic, and submitting it to the Gradescope autograder. Learning this submission process now is important as the same process will be used for future assignments.

By the end, you'll have written three simple Python programs from scratch and you'll have a solid foundation for everything we will build next.

▲ Back to Top

Part 1: Weather Index Calculator

For Part 1 you will build a small **Weather Index Calculator**, in a file named **`weather.py`**, that turns raw weather measurements into numbers people actually use to make decisions. Weather apps rarely show just the air temperature because temperature alone can be misleading. Wind and humidity change how your body gains or loses heat, which changes how conditions feel.

You will compute two common Canadian weather indices:

- **Wind Chill Index:** an estimate of how cold it feels when wind increases heat loss from exposed skin. On a windy day, the "feels like" temperature can be much lower than the air temperature, which matters for comfort and for cold-exposure risk.
- **Humidex:** an index that combines air temperature and humidity to describe how hot it feels when moist air makes it harder for sweat to evaporate. High humidex values can help explain why a day feels sweltering even if the temperature does not seem extreme.

Your program will ask the user for **temperature**, **humidity**, and **wind speed**, then calculate and display wind chill and humidex. The goal is to practice clean input, variables, arithmetic, and formatted output while working with formulas that are used in real forecasts.

1.1 User Input

Your program will collect three pieces of weather data from the user, in this exact order, using the `input()` function:

1. **Temperature (C)**:
    - Prompt the user for the air temperature in degrees Celsius.
    - The exact text of the prompt should be `"Enter temperature (C): "`.
    - Store it as an integer (for example: -15, 0, 22, 30).
    - You can assume the user inputs a valid whole number, no error checking is needed.
2. **Humidity (0 to 1)**:
    - Prompt the user for relative humidity as a decimal between 0 and 1 (inclusive).
    - The exact text of the prompt should be `"Enter humidity (0 to 1): "`.
    - Store it as a float (for example: 0.45 means 45% humidity, 0.70 means 70% humidity).
    - You can assume the user inputs a valid decimal number between 0 and 1 (inclusive), no error checking is needed.
3. **Wind speed (km/h)**:
    - Prompt the user for wind speed in kilometres per hour.
    - The exact text of the prompt should be `"Enter wind speed (km/h): "`
    - Store it as an integer (for example: 0, 10, 35).
    - You can assume the user inputs a valid non-negative whole number, no error checking is needed.

1.2 Wind Chill Calculation

To calculate the **Wind Chill Index (W)** in this assignment, you will use the standard Canadian wind chill equation. Wind chill is an index that estimates how cold conditions feel on exposed skin when wind increases heat loss.

For this calculation, assume the user will enter reasonable weather values. You do not need to validate the user's inputs.

Variables and Units:

- `T` = air temperature in °C *(input from user)*
- `V` = wind speed in km/h *(input from user)*
- `W` = wind chill index *(what you are calculating)*

> In your code make sure to use meaningful variable names using snake_case.
> For example, rather than "W", in your code use "wind_chill" or "wind_chill_index".

Compute wind chill (W) using:

$$W = 13.12 + 0.6215T - 11.37V^{0.16} + 0.3965TV^{0.16}$$

Explanation:
In the wind chill equation, T is the air temperature in °C and V is the wind speed in km/h. The fixed numbers (like 13.12, 0.6215, 11.37, 0.3965) are constants chosen so the index matches typical human heat loss in wind. The V^0.16 part represents how wind increases heat loss: as wind speed increases, this value increases, which changes the final "feels like" result.

1.3 Humidex Calculation

To calculate the **Humidex (H)** in this assignment, you will use the standard Canadian humidex equation. Humidex is an index that estimates how hot conditions feel when humidity makes it harder for sweat to evaporate and cool your body.

For this calculation, assume the user will enter reasonable weather values and a **humidity value between 0 and 1**. You do not need to validate the user's input.

Variables and Units:

- T = air temperature in °C *(input from user)*
- RH = relative humidity as a decimal from 0 to 1 *(input from user)*
- es = saturation vapour pressure in hPa *(calculated)*
- e = actual vapour pressure in hPa *(calculated)*
- H = humidex *(what you are calculating)*

> In your code make sure to use meaningful variable names using snake_case.
> For example, rather than "RH", in your code use "humidity" or "relative_humidity".

Compute humidex (H) using:

1. Calculate saturation vapour pressure (es):
$$es = 6.11 \times 10^{(7.5T)/(237.3 + T)}$$
2. Calculate actual vapour pressure (e):
$$e = RH \times es$$
3. Calculate humidex (H):
$$H = T + 0.5555(e - 10)$$

Explanation:
In these equations, T is the air temperature and RH tells you how much moisture is in the air compared to the maximum it could hold at that temperature. The value es is the maximum

(saturated) vapour pressure at temperature T, and e scales that down based on RH. The final humidex formula adds an adjustment based on e, so higher humidity (larger e) increases the humidex and makes the conditions feel hotter than the air temperature alone.

1.4 Output

Your program should output the **Wind Chill (W)** and the **Humidex (H)** for the input values even if they are outside of the assumed ranges (no error checking needed). Both values should be rounded to one decimal point using the `round()` function before being output. The Wind Chill (W) should have a `"C"` output after the value to show that it is in Celsius.

Make sure you match the formatting and spelling of the output shown in the following examples. All spacing, spellings, line breaks, capitalization, and punctuation should be the same.

Example 1.1

```
Enter temperature (C): 0
Enter humidity (0 to 1): 0.5
Enter wind speed (km/h): 10

Wind Chill Index: -3.3 C
Humidex: -3.9
```

Green text *is user input,* ***white text*** *is program output.*

Example 1.2

```
Enter temperature (C): 25
Enter humidity (0 to 1): 0.75
Enter wind speed (km/h): 35

Wind Chill Index: 26.1 C
Humidex: 32.6
```

Green text *is user input,* ***white text*** *is program output.*

Example 1.3

```
Enter temperature (C): -50
Enter humidity (0 to 1): 0
Enter wind speed (km/h): 100
```

```
Wind Chill Index: -83.1 C
Humidex: -55.6
```

Green text is user input, white text is program output.

Example 1.4

```
Enter temperature (C): 35
Enter humidity (0 to 1): 1
Enter wind speed (km/h): 0

Wind Chill Index: 34.9 C
Humidex: 60.7
```

Green text is user input, white text is program output.

▲ Back to Top

Part 2: Download Time Estimator

When you download a big file, the progress bar usually shows an estimate like *"2 minutes 30 seconds remaining"*. That number is not magic, it is just a quick calculation based on how much data is left and how fast your connection can move that data. In this part, you will build a **Download Time Estimator**, in a file named **`download.py`**, that asks the user for a **file size** and a **download speed**, then reports an estimated time in a clean `hours : minutes : seconds` format. This is the same kind of calculation used by download managers and streaming apps to give you a "time remaining" estimate.

Be careful with units. The file size will be input in **megabytes (MB)**, while internet speed will be input in **megabits per second (Mbps)**. A byte is 8 bits, so 1 MB = 8 megabits (Mb). That means you must convert the file size from megabytes to megabits before performing your calculations.

2.1 User Input

Your program must ask the user for two values in this exact order, using the `input()` function:

1. **Total file size (MB):**
   - Prompt the user for the file size in **megabytes**.
   - The exact text of the prompt should be `"Enter the file size (in MB): "`.
   - Store it as a real number (a float), for example 52.5, 0.45, etc.

- You can assume the user inputs a valid positive decimal number, no error checking is needed.

2. **Download speed (Mbps):**
   - Prompt the user for the download speed in **megabits per second**.
   - The exact text of the prompt should be `"Enter the download speed (in Mbps): "`.
   - Store it as a real number (a float), for example 75.5, 0.5, etc.
   - You can assume the user inputs a valid positive decimal number, no error checking is needed.

2.2 Download Time Calculations

After taking input from the user, your program should estimate how long the download will take by working through the calculation in a few clear steps. The goal is to compute a total number of seconds, then convert that into hours, minutes, and seconds. Have your program perform the following steps to calculate the download time:

1. **Convert the file size into the right units (megabits).** Don't round this result, keep it as a real number.
2. **Estimate the total download time in seconds.** Round this into a whole number of seconds using the `round()` function.
3. **Break total seconds into hours, minutes, and seconds.** Each should be whole numbers (integers), if the total download time is less than one second (after rounding) they would all be zero.

Calculation Hints

1. One megabyte (MB) is equal to eight (8) megabits (Mb).
2. It is recommended to store the result of each conversion in its own variable, for example you could store the number of seconds in a variable called `total_seconds`.
3. To break the total seconds into hours, minutes, and seconds remaining you will need to use integer division (`//`) and the modulo (`%`) operators. Use a combination of `//` and `%` to peel off each time unit cleanly.

2.3 Output

After performing the calculations, your program must display the estimated download time in the **`H:MM:SS`** format. Your output should include:

- **H** is the number of hours (whole number, can be 0 or more, no leading zero required, can have one or more digits).
- **MM** is the number of minutes (whole number, can be 0 or more, no leading zero required, can have one or two digits).
- **SS** is the number of seconds (whole number, can be 0 or more, no leading zero required, can have one or two digits).

Make sure you match the formatting and spelling of the output shown in the following examples. All spacing, spellings, line breaks, capitalization, and punctuation should be the same.

Example 2.1

```
Enter the file size (in MB): 100
Enter the download speed (in Mbps): 5

Estimated download time: 0:2:40
```

Green text *is user input,* ***white text*** *is program output.*

Example 2.2

```
Enter the file size (in MB): 123456789.42
Enter the download speed (in Mbps): 10.37

Estimated download time: 26455:58:16
```

Green text *is user input,* ***white text*** *is program output.*

Example 2.3

```
Enter the file size (in MB): 1000
Enter the download speed (in Mbps): 8000

Estimated download time: 0:0:1
```

Green text *is user input,* ***white text*** *is program output.*

Example 2.4

```
Enter the file size (in MB): 10
Enter the download speed (in Mbps): 5000
```

```
Estimated download time: 0:0:0
```

Green text is user input, white text is program output.

Hint: Decimals in Output

If you are getting decimals in your output such as `26455.0:58.0:16.0` rather than `26455:58:16` you either need to convert the value to an integer before outputting or use an f-string to format the output with no decimal places (either should work).

▲ Back to Top

Part 3: SMS Moderator Billing Calculator

For this last part of the assignment, you will write a small tool, in a file named **`sms.py`**, that estimates the "cost" of sending a text message. Real messaging apps still have limits under the hood (like message length, segmenting long messages, and systems that scan or filter content), so your program will take a user's message, clean it up, and calculate how many 160-character SMS segments it would require. It will then compute a base cost using a per-segment rate and print a clear summary for the user.

3.1 User Input

Your program must ask the user for three values in this exact order, using the `input()` function:

1. **Message to send (text/string):**
   - Prompt the user to enter the text message. This should be read as a string and may include spaces and punctuation.
   - The exact text of the prompt should be `"Enter the SMS message text: "`.
   - You can assume the user inputs at least one character, no error checking is needed.
2. **Cost per SMS segment (dollars):**
   - Prompt the user to enter the cost (in dollars) to send one SMS segment. This value will be a **positive decimal number** (real number).
   - The exact text of the prompt should be `"Enter the SMS cost per segment: $"`.
   - You can assume the user inputs a valid positive number, no error checking is needed.
3. **Banned word to censor (text/string):**

- Prompt the user to enter a single word that should be censored in the preview. This should be read as a string that contains a single word.
- The exact text of the prompt should be `"Enter the word to censor: "`.
- You can assume the user inputs one word with no spaces or punctuation, no error checking is needed.

3.2 Clean and Censor the Message

Before you calculate the cost to send the message, you must clean it and censor the banned word. Complete the following cleaning steps before calculating the cost:

1. **Remove any leading and trailing spaces in message.** For example if the user input `" Hello World "` with extra spaces, only the spaces at the start and end of the text should be removed and the result should be `"Hello World"`.
2. **Convert the text to all lowercase.**
3. **Replace any occurrence of the banned word** in the message text with three asterisks (*). Any occurrences at all should be replaced, even if they are part of another word. This replacement should be case insensitive (capitalization should be ignored). For example if the banned word was `"dUcK"` and the message was `"One duck, two DUCK, three DuCk!"` the result would be `"One ***, two ***, three ***!"`.

Hint: String Methods

These steps may sound complex, but there is no need to use if statements or loops here. All of these steps can be accomplished with the built in string functions and methods.

3.3 Calculate the Cost

Your program will compute the total cost based on the cleaned and censored version of the message, not the original message the user typed. The cost is based on the number of segments the SMS message has, where each segment can contain up to 160 characters. You will need to find the smallest number of 160-character segments that can hold the entire censored message (in other words, round up to the next whole segment when needed). For example, if a message contained 170 characters, this would be two segments; one with 160 characters and one with 10 characters.

Multiply the number of segments in the message by the **cost per SMS segment** you input from the user to find the total cost to send the message. The resulting cost should be a real number rounded to two decimal places using the `round()` function.

3.4 Output

Print a short report in the exact order shown in the examples below using the same spelling, formatting, and spacing. Use a divider line made with string repetition (for example, `"=" * 30`) to draw a line of 30 `"="` characters as shown in the examples.

The report should include:

1. Divider line of 30 `"="` characters.
2. Cleaned/Censored message preview
3. Divider line of 30 `"="` characters.
4. Number of characters in the censored message.
5. Number of SMS segments.
6. Total cost to send the SMS.

Note that the while the total cost should be rounded to two decimal places, Python may print it with a single decimal place if there are no decimals to display (for example, a cost of exactly $10.5 would be printed as $10.5 and not $10.50). For this assignment this is fine, and you are not required to pad the number with any trailing zeros.

Example 3.1

```
Enter the SMS message text: I love spoilers. No spoilers please.
Enter the SMS cost per segment: $0.05
Enter the word to censor: spoilers
==============================
i love ***. no *** please.
==============================
Characters: 26
Segments: 1
Cost: $0.05
```

Green text *is user input,* ***white text*** *is program output.*

Example 3.2

```
Enter the SMS message text: Running 10 min late to CS1026 lab. Can y
Enter the SMS cost per segment: $10
Enter the word to censor: Duck
==============================
running 10 min late to cs1026 lab. can you start without me? if the
==============================
```

```
Characters: 160
Segments: 1
Cost: $10.0
```

Green text *is user input,* ***white text*** *is program output.*

Example 3.3

```
Enter the SMS message text: When a program acts weird, add print sta
Enter the SMS cost per segment: $0.16
Enter the word to censor: int
==============================
when a program acts weird, add pr*** statements to show values. conv
==============================
Characters: 161
Segments: 2
Cost: $0.32
```

Green text *is user input,* ***white text*** *is program output.*

Example 3.4

```
Enter the SMS message text:                Any spaces at the start
Enter the SMS cost per segment: $100
Enter the word to censor: MESSAGE
==============================
any spaces at the start or end of the *** should be removed but not
==============================
Characters: 147
Segments: 1
Cost: $100.0
```

Green text *is user input,* ***white text*** *is program output.*

Example 3.5

```
Enter the SMS message text: Hey! Quick update on tonight: the party
Enter the SMS cost per segment: $0.01
Enter the word to censor: art
==============================
```

```
hey! quick update on tonight: the p***y st***s at 7:30 at mia's ap**
==============================
Characters: 363
Segments: 3
Cost: $0.03
```

Green text *is user input,* ***white text*** *is program output.*

▲ Back to Top

Non-Functional Specification and Other Details

In addition to the other requirements given in the previous sections, your program must also fulfill the following requirements:

- **Document each file:** You MUST provide a comment at the top of **all** Python files (files ending in .py) that contains 1) your full name, 2) your student number, 3) your uwo username (the same one you use to log in to OWL), 4) the date the file was created, and 5) a description of what this specific file does. This must be provided at the top of all files, and the description should be updated to describe the specific file it is in (it can not be the same in all files).
- **Other comments throughout your code:** You must also include comments in your code for any lines that are not easily understood. You just must use good judgment in what lines require comments, the marker may remove marks for over or under commenting.
- **Do not import any modules (other than math):** for this assignment there should be no need to import any modules. You can import the math module if you wish, but it is not required.
- **Match the example text:** The text contained in your messages and prompts to the user should match the examples given and all input must be taken in the same order.
- **Use meaningful variable names:** Your variable names should be meaningful, clearly state the values they store, and use the `snake_case` name format. Don't use variable names that are simply letters such as "t" or "W".
- **Follow good programming practices:** All included code must have a clear purpose and function. Including code that has no purposes, such as variables that are never used, may lead to marks being removed. You should understand each line of your program and each line should have a purpose. Marks may also be removed for other poor programming practices such as inappropriate recursion, redundant code, not making use

of the specified functions when appropriate, or other code quality issues that demonstrate a lack of understanding.

- **Follow good style:** As much as possible, you should follow the PEP8 style guide. Both the autograder and your IDE should provide hints about style violations.
- **Python 3.12:** You code should work correctly in Python 3.12 and newer. These are the versions of Python the auotgrader will be compatible with.
- **No hardcoding:** You may not hardcode your returns, outputs, or writing to files. Your code should work correctly for any valid input. Attempts to fool or trick the auotgrader will result in the marker manually assigning a zero grade for that test.

▲ Back to Top

Assessment

The assignment will be marked as a combination of your auto-graded tests and manual grading of your code logic, comments, formatting, style, etc. Marks will be deducted for failing to follow any of the specifications in this document, not documenting your code with comments, using poor formatting or style, hardcoding, or naming your files incorrectly.

Marking Scheme:

- **Autograded Tests:** 80 points
- **Header comment including your name, student ID, username, creation date, and description of file:** 8 points
- **Descriptive in-line comments throughout code:** 5 points
- **Meaningful variable names:** 2 point
- **Good programming practices *(see Non-Functional Specification)*:** 5 points

Total: 100 points

Note: *Marks from the autogradeed tests may also be manually removed by the marker for violating any of the requirements given in the Non-Functional Specification, hard coding solutions, or failing to implement any features as specified.*

▲ Back to Top

Submission Instructions

You must upload **all files for all parts** (`weather.py`, `download.py`, and `sms.py`) **to the Gradescope autograder**. For a tutorial on how to use Gradescope to submit a code assignment, please review this link.

You may resubmit your assignment to Gradescope an unlimited number of times before the submission deadline. Gradescope will give you some feedback from the autograder each time you submit.

Any submissions after the due date will be considered late and subject to the late policy. Do not resubmit after the due date if you do not want to use late coupons.

Your final submission **must** include the files for all three parts of this assignment. Any missing files will result on a zero grade for that part of the assignment. Only your most recent submission will count for your assignment grade.

Submissions will not be accepted via e-mail or through any other means. The autograder will only accept files with the correct file extensions. Make sure all of your files have one file extension and that they are named correctly. Any incorrectly named files will result in a zero grade for that part.

Remember, **you are required to check that the files you submit are correct and all autograder tests are passing**. Teaching assistants will only review your submission for items relating to the Non-Functional Specification when marking, they will not give marks back for failed autograder tests.

▲ Back to Top