ESC 180 H1F Project # 1 — General Instructions

Due: Oct. 12 by 10:59pm **Worth: 6%**

Late submission accepted with no penalty until end of Oct. 16. 5% per day penalty after (rounded up)

Submitting your project

You must hand in your work electronically, using Gradescope. Log in to

`https://www.gradescope.com/courses/432785`

Login information will be sent to your utoronto.ca email this week.

You can submit individually or as a team of two. To submit as a team of two, have one team member submit first, and then add the other person to their team.

For this project, you must hand in just one file:

- `credit.py`

You can submit a new version of the file at any time (though the lateness penalty applies if you submit after the deadline)—look in the "Replace" column. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission.

If your file is not named exactly as specified, your code will receive zero for correctness.

The `if __name __== __"main"__` block

All your testing code should be inside the `if __name __== __"main"__` block. All you global variables must be initialized outside of the `if __name __== __"main"__` block.

Clarifications and discussion board

Important clarifications and/or corrections to the project, should there be any, will be posted on the ESC180 Piazza. responsible for monitoring the CSC180H1F Piazza discussion board.

Hints & tips

- Start early. Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. You can overwrite previous submissions with more recent ones, so submit early and often—a good rule of thumb is to submit every time you get one more feature implemented and tested.
- Write your code incrementally. Don't try to write everything at once, and then compile it. That strategy never works. Start off with something small that compiles, and then add functions to it gradually, making sure that it compiles every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start—ask questions if anything is unclear!
- Inspect your code before submitting it. Also, make sure that you submit the correct file.
- Seek help when you get stuck! Check the discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during the lab if you are having difficulties with programming. Go to the instructors' office hours if you need extra help with understanding the course content.

At the same time, beware not to post anything that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved! If you cannot think of a way to ask your question without giving away part of your solution, then please drop by office hours or ask by email instead.

- If your email to the TA or the instructor is "Here is my program. What's wrong with it?", don't expect an answer! We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

How you will be marked

We will mark your project for correctness. We will run a large number of test cases on your code, and base your mark on the number of test cases that your code passes.

You are encouraged to comment your code and to use good programming style, but those aspects are not marked explicitly.

You are strongly encouraged to thoroughly test your code, using both typical cases and "edge" cases. We are not directly marking your testing strategy, but usually, people who test their code pass more of our marking test cases; so you are *indirectly* marked on your testing strategy.

Correctness

We will run your functions using a Python 3 interpreter. Please ensure that you are running Python 3 as well. To check what version of Python you are running, you can run the following in your Python shell:

```
import sys
sys.version
```

Syntax errors in your code will cause you to lose most of the marks for this project.

Note that **your functions must be implemented precisely according to the project specifications.** Their signatures should be exactly as in the project handout, and their behaviour should be exactly as specified. In particular, make sure that functions do not print anything unless the project specifications specifically demand that, and that the functions return exactly what the project handout is asking for.

Testing

Testing is not graded explicitly. Below is a guide to how to write your testing code.

You should include code that tests the functions that you have written to make sure that they match the project specifications. Make sure that you test your functions thoroughly. That means that you should make sure that your functions work for all the different possible scenarios. Mindlessly plugging in various parameter values is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

The documentation of the testing strategy should include, for each function you test and for each test case, a description of what output your function should produce, and a brief explanation of why the test case and output are significant in verifying the correctness of the program.

Documentation

Documentation is not graded explicitly. Below is a guide to how to document your code.

When writing code, you must write documentation to describe what your code is doing. Documentation helps others and yourself understand what your code is meant to do. The general rule of thumb for documentation states that you should add comments to your code in the following situations:

- For every function, as a docstring, to describe the parameters of the function and what the function does. See below for more details on docstrings.
- Before every global variable declaration, to describe what kind of information the variable stores and what properties (if any) that information is supposed to have throughout the execution of the code.
- Before all complicated sections of code, to help the reader understand what that code section is trying to do.
- In general, comments should *not* simply restate what the code does (this does not add any useful information to the code). Comments should *add* information that is implicit in the code, *e.g.*, about what purpose a computation serves, or why a certain section of code is written the way it is.

Style

Good style practices should be adhered to when writing your code. This includes the following:

- Use Python style conventions for your function and variable names. In particular, please use "pothole case": lowercase letters with words separated by underscores (`_`), to improve readability.
- Choose good names for your functions and variables. For example, `num_coffee_cups` is more helpful and readable than `ncc`.
- To make sure your program will be formatted correctly is never to mix spaces and tabs when indenting —use only tabs, or only spaces.
- Put a blank space before and after every operator. For example:

```
b = 3 > x and 4 - 5 < 32  # good style: easy to read

b= 3>x and 4-5<32         # bad style: hard to read
```

- Write a docstring comment for each function. (See below for guidelines on the content of your docstrings.) Put a blank line after every docstring comment.
- Each line must be less than 80 characters long, including tabs and spaces. You should break up long lines using `\`.
- Your code should be readable and readily understandable.

Guidelines for writing docstrings

- Describe precisely *what* the function does.
- Do not reveal *how* the function does it.
- Make the purpose of every parameter clear.
- Refer to every parameter by name.
- Be clear about whether the function returns a value, and if so, what.

- Explain any conditions that the function assumes are true. Examples: "`n is an int`", "`n != 0`", "`the height and width of p are both even`".
- Be concise.
- Ensure that the text you write is grammatically correct.
- Write the docstring as a command (*e.g.*, "Return the first ...") rather than a statement (*e.g.*, "Returns the first ...").

For this project, you will implement a simulator for credit card transactions. You will maintain the balance owed on the credit card by keeping track of new purchases, the interest accrued, and bill payments that can sometimes be partial. In addition, you will implement a simple flagging algorithm, and make sure that the card is deactivated (i.e., no further purchases can be made) if fraud is suspected.

The simulation proceeds as a series of operations, which are simulated using calls to the functions that you will define. For example, a simulation might proceed as follows:

```
if __name__ == '__main__':
    initialize()
    purchase(80, 8, 1, "Canada")
    print("Now owing:", amount_owed(8, 1))
    pay_bill(50, 2, 2)
    print("Now owing:", amount_owed(2, 2))
    print("Now owing:", amount_owed(6, 3))
    purchase(40, 6, 3, "Canada")
    print("Now owing:", amount_owed(6, 3))
    pay_bill(30, 7, 3)
    print("Now owing:", amount_owed(7, 3))
    print("Now owing:", amount_owed(1, 5))
```

The lines in the code above correspond to purchases, bill payments, and inquiries about amounts owed. The passage of time is simulated by having dates in the function calls.

The credit card account operates according to the following rules.

- Initially, the amount owed is 0.
- The amount owed is divided into two parts: the amount that is accruing interest, and the amount that is not accruing interest. The only money that is not accruing interest during the month is the money spent on purchases during that same month. Any other money owed is accruing interest.
- An interest of 5% is added to the amount owed in the last second of each month. Assume that no purchases are made between the time the interest is added to the amount owed and the time that the month changes.
- When the credit card bill is paid, and the amount owed is not paid in full, the payment first goes to pay the amount that is accruing interest, and only then to pay the amount that is not accruing interest.
- If the card is used for purchases in three different countries in a row, the card is deactivated. The third purchase does not work, and no further purchases can be made.

  For example, if two purchases are made in Canada and the United States, and the next attempted purchase is in France, the third purchase does not go through, and no money is added to the amount owed. If another attempt to make a purchase is then made in Canada, again, the purchase attempt fails and no money is added to the amount owed.

  On the other hand, if purchases are made in Canada, then France, then Canada, then Canada, then the United States, and then the United States, all the purchases go through, since at no point were purchases made in three different countries in a row.

Here is a sample output of a simulation, along with comments explaining the output.

```
if __name__ == '__main__':
    initialize()
    purchase(80, 8, 1, "Canada")
    print("Now owing:", amount_owed(8, 1))      #80.0
    pay_bill(50, 2, 2)
    print("Now owing:", amount_owed(2, 2))      #30.0     (=80-50)
    print("Now owing:", amount_owed(6, 3))      #31.5     (=30*1.05)
    purchase(40, 6, 3, "Canada")
    print("Now owing:", amount_owed(6, 3))      #71.5     (=31.5+40)
    pay_bill(30, 7, 3)
    print("Now owing:", amount_owed(7, 3))      #41.5     (=71.5-30)
    print("Now owing:", amount_owed(1, 5))      #43.65375 (=1.5*1.05*1.05+40*1.05)
    purchase(40, 2, 5, "France")
    print("Now owing:", amount_owed(2, 5))      #83.65375
    print(purchase(50, 3, 5, "United States"))  #error    (3 diff. countries in
                                                #          a row)

    print("Now owing:", amount_owed(3, 5))      #83.65375 (no change, purchase
                                                #          declined)
    print(purchase(150, 3, 5, "Canada"))        #error    (card disabled)
    print("Now owing:", amount_owed(1, 6))      #85.8364375
                                                #(43.65375*1.05+40)
```

We provide you with a "starter" version of `credit.py`—a skeleton of the code you will have to write, with some parts already filled in. Please read it carefully and make sure you understand everything in the starter code *before* you start making changes! You must not change the function signatures.

When designing and testing your functions, you may assume that every date given is a valid date in 2020 (there is no need to check that). However, you may not assume that dates used later in the simulation always occur chronologically after dates that were used earlier.

Part 1.

Implement the following functions in `credit.py`. Note that the names of the functions are case-sensitive and must not be changed. You are not allowed to change the number of input parameters. Doing so will cause your code to fail when run with our testing programs, so that you will not get any marks for functionality.

Note that "iff" means "if and only if." When we say that a function `f` returns True iff condition `c` holds, we mean that `f` returns `True` if `c` holds, and `False` if it doesn't hold.

Subpart (a) `date_same_or_later(day1, month1, day2, month2)`

This function returns `True` iff the date `(day1, month1)` is the same as the date `(day2, month2)`, or occurs later than `(day2, month2)`. Assume the dates given are valid dates in the year 2020.

Subpart (b) `all_three_different(c1, c2, c3)`

This function returns `True` iff the values of the three strings `c1`, `c2`, and `c3` are all different from each other.

Subpart (c) `purchase(amount, day, month, country)`

This function simulates a purchase of amount `amount`, on the date `(day, month)`, in the country `country` (given as a capitalized string). The function should return the string `"error"` and not have any other effect (except for possibly disabling the card) if any of the following conditions obtain:

- There already was a simulation operation on a date later than `(day, month)` (e.g., a purchase or a check for the amount owed).
- The card is becoming disabled due to the current attempted purchase, or is already disabled.

You may assume that `amount` is greater than 0 and that `country` is a valid country name.

Subpart (d) `amount_owed(day, month)`

This function returns the amount owed as of the date `(day, month)`

This function returns the string `"error"` if there already was a simulation operation on a date later than `(day, month)` (e.g., a purchase or a check for the amount owed).

Subpart (e) `pay_bill(amount, day, month)`

This function simulates the payment of the amount owed on the credit card. When the credit card bill is paid, and the amount owed is not paid in full, the payment first goes to pay the amount that is accruing interest, and only then to pay the amount that is not accruing interest.

You may assume that `amount` is greater than 0.

This function returns the string `"error"` if there already was a simulation operation on a date later than `(day, month)` (e.g., a purchase or a check for the amount owed).

Part 2.

In the `if __name__ == "__main__"` block, add code that tests your functions. While you may run large simulations as well, your job is to come up with a testing strategy that ensures that your code works according to the project specifications. This is best done by testing every individual aspect of the behaviour of the code. Add comments to clarify the testing strategy: the goal is to make sure that it is possible to look at the testing code that you wrote and the comments that you have added, and be convinced that you have tested for all the categories of the typical cases, and all the categories of the boundary/edge cases.

Note that your testing strategy is not explicitly graded.