Due: Sunday 16 October by 10:59pm **Worth: 4%**

Submitting your project

You must hand in your work electronically, using the *MarkUs* system. Log in to

```
https://markus.cdf.toronto.edu/csc180-2016-09/en/main
```

using your UTORid.

To submit your work, again navigate to the Project 1 page, then click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". For this project, you must hand in just one file:

- `gamify.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.

Once you have submitted, click on the file's name to check that you have submitted the correct version. Remember that the names of the files you submit must be exactly as specified (and the case of the letters must be the same). 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 (for example, by calling `initialize()`.)

Clarifications and discussion board

Important clarifications and/or corrections to the project, should there be any, will be posted on the CSC180H1F Piazza. You are responsible for monitoring the announcements there. You are also generally 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, and the thoroughness and good documentation of your testing strategy. Below, we give some suggestions for imroving the readability of your code. You will not be marked on the documentation of your code, although we encourage you to document your code anyway.

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

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

When writing code, you should 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 ...").

Gamification is the integration of elements found in games – such as points and badges – into non-game activities. One example of gamification is students' being awarded badges and points for completing exercises in online courses. This is done in order to encourage students to complete the exercises. Another example is supermarkets awarding points to customers for purchasing various items. This is done, in part, to induce the customers to purchase more items in order to gain more points.

In this project, you will implement a simulator for an app that encourages the user to exercise more by awarding "stars" to the user for exercising. The simulator will model how the user behaves, and could be used to try out various strategies for awarding stars.

We imagine the user as accumulating "health points" and "fun points" (sometimes called hedons). Every activity is associated with gaining some number of health points and some number of hedons. Receiving a star increases the number of hedons that the user gains from performing the activity. Receiving too many stars too often makes the user lose interest in stars altogether.

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()
    perform_activity("running", 30)
    print(get_cur_hedons())            #-20 = 10 * 2 + 20 * (-2)
    print(get_cur_health())            #90 = 30 * 3
    print(most_fun_activity_minute())  #resting
    perform_activity("resting", 30)
    offer_star("running")
    print(most_fun_activity_minute())  #running
    perform_activity("textbooks", 30)
    print(get_cur_health())            #150 = 90 + 30*2
```

The lines in the code above correspond to performing various activities (running, resting, carrying textbooks), for various durations of time, as well as querying the system for the number of health points and hedons that the user accumulated and querying the system for the activity that would gain the user the most hedons if it were performed for one minute.

We assume that the use is always either running, carrying textbooks, or resting.

The user accumulates fun points and health points according to the following rules.

- The user starts out with 0 health points, and 0 hedons.
- The user is always either running, carrying textbooks, or resting.
- Running gives 3 health points per minutes for up to 180 minutes, and 1 health point per minute for every minute over 180 minutes that the user runs. (Note that if the user runs for 90 minutes, then rests for 10 minutes, then runs for 110 minutes, the user will get 600 health points, since they rested in between the times that they ran.)
- Carrying textbooks always gives 2 health points per minute.
- Resting gives 0 hedons per minute.
- Both running and carrying textbooks give -2 hedons per minute if the user is tired and isn't using a star (definition: the user is tired if they finished running or carrying textbooks less than 2 hours before the current activity started.) For example, for the purposes of this rule, the user will be tired if they run for 2 minutes, and then start running again straight away.

- If the user is not tired, running gives 2 hedons per minute for the first 10 minutes of running, and -2 hedons per minute for every minute after the first 10.
- If the user is not tired, carrying textbooks gives 1 hedon per minute for the first 20 minutes, and -1 hedon per minute for every minute after the first 20.
- If a star is offered for a particular activity and the user takes the star right away, the user gets an additional 3 hedons per minute for at most 10 minutes. (Note that the user only gets 3 hedons per minute for the first activity they undertake, and do not get the hedons due to the star if they decide to keep performing the activity:

```
offer_star("running")
perform_activity("running", 5)  #gets extra hedons
perform_activity("running", 2)  #no extra hedons
```

- If three stars are offered within the span of 2 hours, the user loses interest, and will not get additional hedons due to stars for the rest of the simulation.

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

```
if __name__ == '__main__':
    initialize()
    perform_activity("running", 30)
    print(get_cur_hedons())            #-20 = 10 * 2 + 20 * (-2)
    print(get_cur_health())            #90 = 30 * 3
    print(most_fun_activity_minute())  #resting
    perform_activity("resting", 30)
    offer_star("running")
    print(most_fun_activity_minute())  #running
    perform_activity("textbooks", 30)
    print(get_cur_health())            #150 = 90 + 30*2
    print(get_cur_hedons())            #-80 = -20 + 30 * (-2)
    offer_star("running")
    perform_activity("running", 20)
    print(get_cur_health())            #210 = 150 + 20 * 3
    print(get_cur_hedons())            #-90 = -80 + 10 * (3-2) + 10 * (-2)
    perform_activity("running", 170)
    print(get_cur_health())            #700 = 210 + 160 * 3 + 10 * 1
    print(get_cur_hedons())            #-430 = -90 + 170 * (-2)
```

We provide you with a "starter" version of `gamify.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.

Part 1.

Implement the following functions in `gamify.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) `get_cur_hedons()`

This function returns the number of hedons that the user has accumulated so far.

Subpart (b) `get_cur_health()`

This function returns the number of health points that the user has accumulated so far.

Subpart (c) `offer_star(activity)`

This function simulates a offering the user a star for engaging in the exercise `activity`. Assume `activity` is a string, one of `"running"`, `"textbooks"`, or `"resting"`.

Subpart (d) `perform_activity(activity, duration)`

The function simulates the user's performing activity `activity` for `duration` minutes. Assume `duration` is a positive `int`. If `activity` is not one of `"running"`, `"textbooks"`, or `"resting"`, running the function should have no effect.

Subpart (e) `star_can_be_taken(activity)`

The function returns `True` iff a star can be used to get more hedons for activity `activity`. A star can only be taken if no time passed between the star's being offered and the activity, *and* the user is not bored with stars, *and* the star was offered for activity `activity`.

Subpart (f) `most_fun_activity_minute()`

The function returns the activity (one of `"resting"`, `"running"`, or `"textbooks"`) which would give the most hedons if the person performed it for one minute at the current time.

Subpart (g) `initialize()`

The function initialize all the global variables in the program. The following code should run two independent simulations, with both `SIMULATION 1` and `SIMULATION 2` starting from the beginning.

```
initialize()
#SIMULATION 1 CODE
#....

intiailize()
#SIMLUATION 2 CODE
#...
```

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.