Due: **Worth:**

Submitting your project

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

- `workweek.py`

You can resubmit a new version of the file at any time before the deadline. Your last submission will be graded.

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()`.)

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 or your instructor.
- 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.

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 your mark to be 0. Make sure that you submit a file that does not contain syntax errors. If the file contains syntax errors, you will see that on Gradescope right away.

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.

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 …”).

For this assignment, you will implement an (incomplete, unfair, and unrepresentative) simulator of an engineering student's life during the work week from Monday at 12AM (i.e., midnight just after Sunday ends) through Friday at 5PM. You will keep track of the amount of knowledge (in units of knowledge called "knols") the student accumulates during the work week, of how many hours the student has spent sleeping during the week, and of whether the student is alert or not. During a given lecture, the student is alert if at least one of the following is true at the **start** of the lecture:

1. the student has spent **more than** (note: that's "strictly more than", not "more than or exactly") 30% of the work week so far sleeping, *or*
2. the student has had coffee **less than** one hour before commencing the activity

on the condition that they have not rendered themselves not alert for the rest of the week by drinking coffee twice in a period of less than three hours. If the student drinks a second cup of coffee in a period of less than three hours, they stop being alert and cannot become alert for the rest of the week. So for example, if the student has had coffee at 2PM and then at 5PM, they can still be alert, but if they have coffee at 2PM and then at 3PM (or any other time earlier than 5PM, including immediately after the first cup), they can't become alert again for the rest of the week.

If the student is alert at the start of the lecture, they obtain 4 knols per hour while attending lectures in the subject `"CSC"`, 2 knols per hour while attending lectures in the subjects `"MAT"`, `"PHY"`,`"ESC"`, and `"CIV"`, and 0 knols if the subject is none of the above (e.g., `"csc"`, `"cSC"`, `"aaaa"`, and `"CSC100"` are none of the above – valid course codes are written in all caps and with no digits). If the student is not alert at the start of the lecture, they obtain half the amount of knols they would obtain if they had been alert (i.e., 2 knol/hr for `"CSC"`, 1 knol/hr for the other listed courses, 0 for the rest). The alertness state is determined at the start of the lecture and does not change during the lecture – because all of the instructors are thoroughly captivating, of course!

The simulation might proceed as follows (the descriptions of the functions are given below) – keep in mind that this is just one possible example of using the simulator:

```
sleep(8)                    # sleep from 12AM to 8AM on Monday
attend_lecture("CSC", 2)    # attend the CSC lecture for 2 hours,
                            # gain 2*4 = 8 knols
attend_lecture("MAT", 30)   # attend the MAT lecture for 30 hours,
                            # gain 30*2 = 60 knols (note that since the student
                            # was alert at the start of the lecture, they gain
                            # two knols per hour for the entire 30 hours)
print(get_knol_amount())    # should print 68
print(get_hours_left())     # should print 73 (since 73 = 24 * 5 - 7 - 40)
```

In the simulation, the activities (sleeping, attending lecture, or drinking coffee) occur immediately one after the other in the order in which the functions corresponding to the events are called

We provide you with a "starter" version of `workweek.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!

Part 1.

Implement the following functions in `workweek.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.

Subpart (a) `knols_per_hour(subj, is_cur_alert)`

This function returns the number of knols per hour the student can obtain by studying subject `subj`. `is_cur_alert` is `True` iff the student is alert at the start of the lecture.

Subpart (b) `attend_lecture(subj, hrs)`

This function simulates attending a lecture in subject `subj` for `hrs` hours, if there are enough hours left in the work week (which ends on Friday at 5PM). You may assume that `hrs` is an integer. If there is not enough time left in the week to attend the lecture for `hrs` hours, `attend_lecture` has no effect. If `hrs` is negative, `attend_lecture` has no effect.

Subpart (c) `drink_coffee()`

This function simulates drinking coffee. Drinking coffee does not take up time (in other words, it takes 0 hours). If the student drinks two cups of coffee in a period of less than three hours, they stop being alert, and cannot become alert again during the week.

Subpart (d) is_alert()

This function returns `True` if the student is currently alert (`False` otherwise), according to the rules specified above.

Subpart (e) get_knol_amount()

This function returns the number of knols the student has accumulated so far.

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.