Submitting your assignment

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


using your ECF login and password. You may work alone or with a partner, though we recommend that you find a partner to work with.

To declare your partnership, one of you needs to invite the other to be a partner, and then they need to accept the invitation. To invite a partner, navigate to the Assignment 1 page, find “Group Information”, and click on “Invite”. You will be prompted for the other student’s ECF user name; enter it. To accept an invitation, find “Group Information” on the Assignment 1 page, find the invitation listed there, and click on “Join”. Note that, when working in a pair, only one person should submit the assignment.

To submit your work, again navigate to the Assignment 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 assignment, you must hand in just one file:

• workweek.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.

Clarifications and discussion board

Important clarifications and/or corrections to the assignment, should there be any, will be posted on the CSC180H1F home page on Portal. You are responsible for monitoring the announcements there. You are also responsible for monitoring the CSC180H1F 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 assignment for correctness, good coding style, good commenting style, and the thoroughness and good documentation of 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:

```python
import sys
sys.version
```

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

**Testing**

You should include code that tests the functions that you have written to make sure that they match the assignment 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 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.
• Use a tab width of 4 (Wingware’s default), if you use tabs at all. The best way to make sure your program will be formatted correctly is never to mix spaces and tabs—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 \. In Wing, you can find out what column the cursor is in by looking in the lower left-hand corner.
• 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., "PSY", "csc", "cSC", "aaaa", and "CSC180" 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 other EngSci 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:

```python
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!

When designing and testing your functions, you may assume that every argument that represents a number of hours is an integer. It is not necessary for your code to work with hour amounts given as floats, and the behaviour of the code is not specified in the case that hour amounts are given as floats.
Question 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.

**Part (a) knols_per_hour(subj, is_alert)**

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

**Part (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 `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.

**Part (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.

**Part (d) is_alert()**

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

**Part (e) get_knol_amount()**

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

Question 2.

Add code to test for a variety of scenarios, to make sure that your program meets the specifications above.