The purpose of this assignment is practice using basic expressions and statements, and simple user input/output.

Due: Sunday, January 31st, 11:59pm.

If you have questions, use the Piazza discussion forums (and professor/TA office hours) to obtain assistance.

Remember, do not publicly post code for assignments on the forum! Ask a general question in public, or ask a private question (addressed to "instructors") when you're asking about your particular code. Also please have a specific question; instead of "my code doesn't work, please help", we need to see something like "I'm having trouble when I add that particular line, what am I misunderstanding?". If you are unsure whether a question may be public or not, just mark it as private to be sure. We can change a post to public afterwards if it could have been public.

Background

Programs are more interesting when they involve the user – supplying values to use in calculations, asking for names, and so on. We can get a string from the user via the `input()` function, and then convert to other types as needed via other built-in functions of Python, such as `int()`, `float()`, and `bool()`.

Variables give us the chance to store values for later, recallable by name. We can even repeatedly update the value associated with that name, as a replacement: forgetting the old value forever, remembering the new value until further notice. As procedural programs are very much a ***sequence*** of executed instructions, the exact order of when we store (or replace) a value for a variable, and when we look up and use the current value, is an important part of understanding how to create the solution to a programming task.

This project will get us comfortable getting values of various types from the user and using variables while calculating different things. No control structure is needed for this project -- we'll explore control structures next, so be sure to master these more basic tasks now.

The current task is to perform some calculations on duration of a sequence of tasks, and interacting with the user with standard input (keyboard) and output (screen).

Requirements

You will turn in a Python file on BlackBoard to the appropriate assignment, using our naming convention of:

`userID_2XX_P1.py` **example:** ***`gmason76_225_P1.py`***

Of course, use your actual userID (like gmason76), section number (215 – 224) instead of 2XX, and then everyone's file name ends in **`_P1.py`**

Procedure

We are going to simulate a basic scheduler for Valentine's Day chocolaty orders. Suppose the table below lists possible items in the order and the corresponding time to prepare them. Also assume that all orders must be completed sequentially: items from an order have to be prepared one after another and never in parallel.

Item	Chocolate	Chocolate Cake	Chocolate Ice Cream
Time to Prepare	17 hours	1 hour 42 minutes	2 hours 7 minutes

Create your Python file. When this file is run, the following should happen (see sample run below):

Perform these actions (**in this order!**), storing their answers as needed in variables:

- print: `"Welcome to the Scheduler!"`
- ask: `"What is your name? "`
- ask: `"How many chocolates are there in the order? "`
- store their response. Note – assume this value is a non-negative whole number.
- ask: `"How many chocolate cakes are there in the order? "`
- store their response. Note – assume this value is a non-negative whole number.
- ask: `"How many chocolate ice creams are there in the order? "`
- store their response. Note – assume this value is a non-negative whole number.
- show them the total time (in minutes) to prepare the order, with formatting similar to this example. Note – it's all on one line.

  ```
  Total time: 1249 minutes
  ```

- ask: `"How many minutes do you have before the order is due? "`
- store their response. Assume this value is a whole number at least as large as their total time.
- show them how much extra time they have in minutes, as well as in the form of how many of each units from days to minutes. Note – we don't care whether there are a single or plural number of something; we just always write "days" and never try to write out "1 day".

  ```
  Your extra time for this order is 751 minutes:
  0 days
  12 hours
  31 minutes
  ```

- thank the user by name. It must match what they typed at the beginning of the interaction.

  ```
  Thank you, George!
  ```

Print out everything exactly as in the following example (except the OS-prompt lines, of course). Note that **blue text** is typed by the user running the code, and that the OS-prompt is red, like **demo$**

```
demo$ python3 gmason76_225_P1.py
Welcome to the Scheduler!
What is your name? George
How many chocolates are there in the order? 1
How many chocolate cakes are there in the order? 1
How many chocolate ice creams are there in the order? 1
Total time: 1249 minutes
How many minutes do you have before the order is due? 2000
Your extra time for this order is 751 minutes:
0 days
12 hours
31 minutes
Thank you, George!
demo$
```

Note: In this example, the user happened to type in George, 1, 1, 1, and 2000. Your code must work for any name, and any non-negative integers as specified. If you hardcode specific values for the measurements into your code and don't use user input, you aren't actually solving the program!

Notes

- Careful! Don't ask additional questions in your code, or ask them out of order; we need you to exactly match the printing order above, for whatever values the user types in.
- Precision is the name of the game – when programming, **you need to care about details.** Make sure you get the spacing exact; uppercase the correct letters; and so on.
- There's a bit of extra credit for exactly matching the spacing and formatting. Get these bonus points now; they are some of the easiest to get all semester! (And it makes for happy graders ☺)

Assumptions

As we are just starting to program, and don't even have if-else statements or exception handling at our disposal, we will assume some things that are not normally allowed for programs to assume. You must assume that:

1. The user will always enter an integer when we ask for an integer, a string when we ask for a string, etc.
2. All numbers the user types in are non-negative (zero or positive). They are large enough to ensure reasonable calculations (as noted above).

Testing your code

To check whether your code works as required, you should run your program, feed in inputs as the example above and compare the output with the provided one to see whether they match. You should also try many different inputs and check the outputs.

When you are ready for a batch of tests, we are providing two files to help you to see how it performs.

- **http://cs.gmu.edu/~yzhong/112/p1/tester_p1.py**
- **http://cs.gmu.edu/~yzhong/112/p1/tests.txt**

You don't need to look inside of those files, just put them in the same directory as your own code file and run this command:

```
demo$ python3 tester_p1.py gmason76_225_P1.py
..........
passed 10/10 tests.
```

If all goes well, you'll see that your code passed all tests (how ever many there are). Each dot represents one test case completing successfully. If a particular test fails, there will be quite a bit more printing, showing specific inputs and the expected outputs. You can scan through each of those examples, see what inputs were used, and test your code on it directly. You can also just scan the "expected" and "got" strings, and look for where they differ, and use that knowledge to go inspect your code and figure out what needs to be changed.

Normally, user input and printed output all get mixed on the screen together; our example had the red, black, and blue text all intermingled. But these test files have to list the user input separately in one place, and then the expected printed output in another. So while the content being compared ("expected" versus "got") might seem to be missing user input, that's just a side effect of the way testing is being performed.

It turns out that a much more proper way of testing is to test "units" of code, usually function definitions; but we don't even know what a function definition is yet! So we have this project's cobbled-together testing script to try to test entire program runs. It's a bit clunky, but we'll be testing the better way pretty soon.

Grading is partly based on passing the test cases. If you're getting the right numbers but failing test cases, check for spacing issues, capitalization, spelling errors, and so on. You get to see the exact tests we'll use when we test your code, so you might as well get them all correct before turning in your work! No surprises there.

Commenting

A comment in Python starts with # and continues to the end of the line. The computer entirely ignores comments, and they are only there for the human readers. We (graders) are humans, and we will be reading through your code! Explain to us what you are doing throughout the entire program's source code. **Even if it seems easy or obvious.**

Comment your code sufficiently so that someone reading the source (the .py file's contents) is guided with clarifying comments and descriptions of each line of code. You might have upwards of half of the entire file be comments; this is normal. Remember, comments should tell the whole story, and the actual code happens to implement that story. Please write comments as you go; it's much more useful to you than writing them all at the end. Also include the following comments at the **top** of the file as follows, personalizing all information.

```
#-------------------------------------------------------------------------------
# Name: YOUR NAME HERE.
# G#: Gxxxxxxx.
# Project X
# Lab Section XXX
#-------------------------------------------------------------------------------
# Honor Code Statement: I received no assistance on this assignment that
# violates the ethical guidelines set forth by professor and class syllabus.
#-------------------------------------------------------------------------------
# References: list any lecture slides, text book pages, any other (allowed)
# resources you've used.
#-------------------------------------------------------------------------------
# Comments and assumptions: A note to the grader as to any problems or
# uncompleted aspects of the assignment, as well as any assumptions about the
# meaning of the specification.
#-------------------------------------------------------------------------------
# NOTE: width of source code should be <= 80 characters to facilitate printing.
#23456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
#       10        20        30        40        50        60        70        80
#-------------------------------------------------------------------------------
```

Grading Rubric:

```
Submitted correctly:           5
Code is well commented:       20
User input obtained:          15
Converted to needed types:    10
Calculations correct:         50
-------------------------------
TOTAL:                       100   +5 bonus: exact outputs matched.
```

Reminders on Turning It In:

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available.

You can turn in your code as many times as you want; we will only grade the final version. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment, you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least for part of the program.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

→ Please do not ask for "free" extensions or the chance to turn in a different file without penalty at a later date. You can always use a late token if necessary, but it's best to just review your submissions the first time around.

→ Start with the first project, backing up your work somewhere safe. I'd suggest a cloud service like DropBox, GoogleDrive, and others. USB drives can be lost; we can accidentally delete the copy we're working on (but recover it with these services if we're careful!); so much can go wrong that you don't need to experience. Do yourself a favor and get in the habit now.