Introduction to Computer Science CS105

Lab 7: Imperative Programming and Algorithm Design

In this lab, you will work on two problems that can be solved most easily with the imperative approach. You are not required to use loops rather than recursion, but you may find your programs are more concise and easier to understand if you do. In both cases, we will suggest an approach in a very abstract form that explains how the algorithm should work (see below), and leave you to complete the design and write the Python program in two ways.

Pre-Lab Work:

What values or errors do the following Python expressions produce? Predict by drawing a "box and arrow" diagram, and then check your answers. Within each group, some of the results depend on prior steps, so remember to enter all the steps, in the order given, to check an answer.

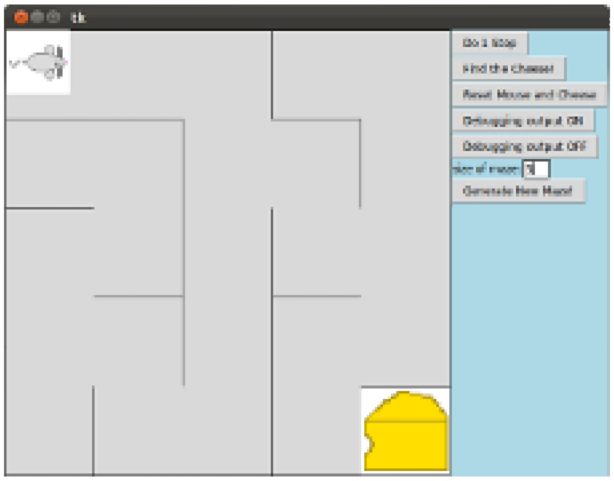
-
  ```
  list1 = [2, 3, 5, 7, 11]
  list1[2]
  [2, 3, 5, 7, 11][2]
  len(list1)
  list1+[13,17,19]
  len(list1)
  list1[6]
  (list1+[13,17,19])[6]
  list1+[13,17,19][6]
  ```

-
  ```
  list1 = [2, 3, 5, 7, 11]
  list1[2]
  list1 = list1+[13,17,19]
  len(list1)
  list1[6]
  ```

-
  ```
  list1 = [2, 3, 5, 7, 11]
  list1[2]
  list2 = list1+[13,17,19]
  len(list2)
  list2[6]
  list1[6]
  list1[3]
  list1[3] = 777
  list1[3]
  list2[3]
  ```

-
  ```
  list1 = [2, 3, 5, 7, 11,13,17,19]
  list2 = list1
  list2[6]
  list1[6]
  list1[3]
  list1[3] = 777
  list1[3]
  list2[3]
  ```

On the maze below, draw the path the mouse will take to reach the cheese if it explores the maze by moving forward while keeping its left paw on the wall until it is on the cheese; then draw a different path (in a different color, or in a different style, etc.) to show what happens if it keeps its *right* paw on the wall.



Lab Work:

1. **Reducing a fraction to "lowest terms"**

   Implement the algorithm described below for "reducing a fraction to lowest terms" (for example, for reducing $\frac{12}{18}$ to $\frac{2}{3}$). One way to reduce a fraction would be to search for numbers that evenly divide both the numerator and denominator (e.g., 2 divides both 12 and 18 evenly), and reduce both numerator and denominator as you find these "common divisors" (in our example with $\frac{12}{18}$, after finding the 2 we would be reduce to $\frac{6}{9}$ and search once again for common factors, finding 3 and reducing to $\frac{2}{3}$, at which point we would find there are no common factors other than 1, and stop). A much faster algorithm involves quickly finding the *greatest common divisor* ("GCD") of the numerator and denominator (e.g., 6 is the GCD of 12 and 18), and then dividing each by this number.

   The GCD-based algorithm is only fast if we can find the GCD quickly, but this can be done with one of the earliest known algorithms (described by the Greek mathematician Euclid in about 300 B.C.). One way to state this algorithm is:

   Given two positive integers, keep replacing whichever number is larger by the remainder of the larger divided by the smaller. When you get a remainder of zero, the other number is the GCD.

   Use this efficient GCD-based algorithm to build a function named `in_reduced_form` and a procedure named `reduce` in the file `A_fraction_reducer.py` in the `more_calculation` project. Both function and procedure will take one parameter: an array of two integers, for which the first represents the numerator and the second the denominator of a fraction. The `in_reduced_form` function should return the reduced fraction as a new array of two of integers without modifying its parameter object. Your `reduce` *procedure* should update the array with the reduced values, changing the object itself. Your function and procedure should be compatible with the existing `reduce_ui` procedure, and should work for any pair of integers that correspond to a valid rational number (they do not have to work for fractions with a denominator of 0, but convenintly `reduce_ui` will not try to reduce these).

   As usual, you should start by creating a test suite (for `reduced` and `reduce`), and then thinking about your algorithm before programming. In this lab, you may also want to think about how to organize the program into procedures and functions — do you want to have a separate `GCD` function? If so, you should combine your "top-down design" strategy with "bottom-up implementation and testing" — write tests for `GCD` function, then write and test the `GCD` function itself, and then turn your attention to programming your reducers.

2. **Mouse in a Maze**

   Any maze that is free of "cycles" (i.e., one without any way to keep walking and get back to the place you started without retracing your steps) can be solved (though sometimes inefficiently) via the following algorithm:

   Place your left (or right) hand on a wall, and keep walking with that hand against the wall until you reach your goal.

   In the file `find_the_cheese.py` of the `mouse_in_a_maze` project, write procedures to use this algorithm to help a mouse (which is trapped in a maze) to reach and eat the cheese. You may assume that the maze has no cycles and tha there will, in fact, be some cheese somewhere in the maze. You should have two procedures — the `one_step_to_cheese` procedure should take one step, and `move_to_cheese` should keep calling upon `one_step_to_cheese` until the mouse reaches the cheese, and then call upon `eat_cheese`. Your procedures should make use of the functions and procedures described in the `from graphics import...` line at the top of the file, summarized here for reference:

```
"""
The look_left(), look_right(), and look_ahead() functions report what
the mouse can see in these three directions.
Note that the mouse is very nearsighted and can see only to the
edge of the space the mouse is standing in, and can only see the
cheese when it is standing on/in it.

These functions return:
   "w"  if there is a wall at the edge of the space the mouse is in
   ""   if there is no wall there and the mouse can walk forward, or
   "c"  if the mouse is standing right in the cheese and can eat.

The turn_left() and turn_right() procedures rotate the mouse.

The move_forward() procedure moves forward.
Its precondition is look_ahead() == ""

The eat_cheese() procedure lets the mouse eat the cheese.
Its precondition is look_ahead() == "c"
"""
```

   You do not have to provide tests or Doctest comments for this question, but include any comments you need to make the expression of the algorithm clear.

Remember to commit your work when you are done (and earlier, if you like).