15-110: Principles of Computing, Spring 2018

Programming Assignment 3

Due: Tuesday, February 6 by 9PM

Note: You are **responsible for protecting your solutions** to the following problems from being seen by other students both physically (e.g., by looking over your shoulder or verbal discussion) and electronically. In particular, since the lab machines use the Andrew File System (AFS) to share files worldwide, you need to be careful that you do not put files in a place that is publicly accessible.

If you are doing the assignment on the Gates-Hillman Cluster machines we use in the lab or on `unix.andrew.cmu.edu`, please remember to have your solutions inside a `private` folder (which is under your home directory). Our recommendation is that you create a `pa3` folder under `~/private/15110` for this assignment. That is, the new directory `pa3` is inside the directory named `15110`, which is inside the `private` directory.

Overview

For this assignment, you will create a Python file for each of the problems below. You should save all of these files in a folder named `pa3`. Once you have every file, you should zip up the `pa3` folder and submit the zipped file on Autolab.

This assignment will make you practice the control structures that we have learned so far (e.g., conditionals, for loops, and while loops) and make you learn how to write the corresponding Python code to a given algorithm. As you will discover this semester, computer programs are rarely correct whey they are first written. They often have bugs. In addition to writing the code, you should test your code on multiple inputs, for which you independently know the correct output (e.g., by plugging the inputs into a calculator).

Problems

1. [1 point] Four numbers a, b, c, and d are called a Pythagorean quadruple if $a^2 + b^2 + c^2 = d^2$. For example, the quadruple (4, 13, 16, and 21 is a Pythagorean quadruple since $4^2 + 13^2 + 16^2 = 21^2$. In a file named `quadruple.py`, write a Python function `quadruple(a, b, c, d)` that takes 4 integers as parameters. (Note that the four parameters represent the values a, b, c and d in that specific order.) Your function must return `True` if the arguments form a Pythagorean quadruple and `False` otherwise.
   For more Pythagorean quadruples you can use for testing, see here:
   https://en.wikipedia.org/wiki/Pythagorean_quadruple

   Example Usage:

```
-bash-4.2$ python3 –i quadruple.py
>>> quadruple(4, 13, 16, 21)
True
>>> quadruple(1, 2, 3, 4)
False
>>> quadruple(8, 4, 8, 12)
True
>>> quadruple(4, 8, 12, 8)
False
```

2. [1 point] In a file named `smallest.py`, define a Python function `smallest(x, y, z)` that requires 3 integer parameters and returns the smallest one among them. You may assume that the function will always be called with three integers that are not equal to one another. You may **NOT** use the built-in function `min`.

   Here is an algorithm to follow:

   1 . If x is smaller than y, then y can't be the smallest, so do the following:
      a. if x is smaller than z, then x is the smallest so set result equal to x.
      b. otherwise, z is the smallest so set result equal to z.
   2. Otherwise, x can't be the smallest, so do the following:
      a. if y is smaller than z, then y is the smallest so set result equal to y.
      b. otherwise, z is the smallest so set result equal to z.
   3. Return result as the final answer.

   Example Usage:

```
-bash-4.2$ python3 –i smallest.py
>>> smallest(300, 200, 500)
200
>>> smallest(2357, 1589, 1156)
1156
```

3. [3 points] The Fibonacci number sequence starts with 0 and 1 as its first two numbers. All subsequent numbers in the sequence are the sum of the previous two numbers. Here are the first 15 numbers in the Fibonacci number sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

(a) In the file `fibonacci.py`, write a Python function `nth_fib(n)` that takes one positive integer parameter `n` and returns the n^{th} Fibonacci number in the sequence using this algorithm:

1. If n is 1, then just return 0.
2. If n is 2, then just return 1.

If you get this far, then n is not 1 or 2, so it must be 3 or higher.
So follow this algorithm to complete your function:

3. Set y equal to 0.
4. Set z equal to 1.
5. Repeat the following n-2 times:
    a. Set x equal to y.
    b. Set y equal to z.
    c. Set z equal to the sum of x and y.
6. Return z.

Example Usage:

```
-bash-4.2$ python3 -i fibonacci.py
>>> nth_fib(1)
0
>>> nth_fib(2)
1
>>> nth_fib(7)
8
>>> nth_fib(15)
377
```

(b) In the same file `fibonacci.py`, write a function `fib_finder(value)` that has one positive integer parameter `value` and returns the first Fibonacci number that is greater than `value`. You will use the algorithm above starting at step 3, but you will need to use a `while` loop for this problem since you won't know how many times to repeat the loop above. Think about what condition you need for the `while` loop to stop at the right time.

```
-bash-4.2$ python3 -i fibonacci.py
>>> fib_finder(377)
610
>>> fib_finder(10000)
10946
>>> fib_finder(1000000)
1346269
```

4. [2 points] By printing out different characters at different locations, it is possible to create images. This is sometimes called as ASCII art, and it works best in Terminal that uses a fixed-width font. Regular shapes, such as the 6 X 6 "square" shown below, are easy to create algorithmically.

```
XXXXXX
X    X
X    X
X    X
X    X
XXXXXX
```

Recall that you can print a single X without moving to the next line this way:
`print("X", end="")`
You can just move to next line without printing anything this way:
`print()`

An ASCII square can be created using the following algorithm, which requires the square's `size` (that is, the number of X's on each side). You may assume that `size` is an integer that is greater than or equal to 2:

1. Print `size` X's (i.e. `size` number of X's) using a loop, then move to next line.
2. Repeat the following `size-2` times:
    a. Print one X and stay on the same line.
    b. Print `size-2` spaces on the same line using a loop.
    c. Print one X and move to next line.
3. Print `size` X's (i.e. `size` number of X's) using a loop, then move to next line.

Note that when `side` = 2, a special case of a square would arise where step 2 would be skipped. If you code this correctly, you don't need additional code to deal with this. It should just work correctly.

Another note: If you look carefully at the algorithm, step 2b is a loop, which will be inside of a loop. This is called a *nested loop*. Loops can be inside of loops! When you use a loop inside of another loop, make sure to use a different loop variable for each loop (e.g., if the outer loop uses `i`, the inner loop can use `j`).

In a file named `make_square.py`, implement the algorithm above as a Python function `make_square(size)`. Your function must return `None` when it is done.

Example Usage:

```
-bash-4.2$ python3 –i make_square.py
>>> make_square(4)
XXXX
X  X
X  X
XXXX
```

5. [3 points] Hourly workers typically earn overtime when they work more than 40 hours per week. Suppose that overtime is twice the regular pay rate for the additional hours.

   In a file names `wages.py`, define a Python function `paytable(rate)` that requires `rate` as a parameter. The rate is the hourly pay rate of the employee as an integer (for example, 12 means $12/hour). Your function should print a list of the total pay if the employee works 20, 25, 30, 35, 40, 45, 50, 55 and 60 hours as shown below. You may assume that the pay rate is between 5 and 124 inclusive. Again, your function must return `None` when it is done.

   You will get up to 2 points if your table prints the correct values (hours and corresponding pay). You will get the additional point if your output looks exactly as shown below including spacing. (HINT: you will need an `if` statement to determine if you need the extra space.) As you work on this problem, get the output numbers correct first. Then try to get the spacing if you can.

   Example Usage:

```
-bash-4.2$ python3 –i wages.py
>>> paytable(10)
20 hours: $ 200
25 hours: $ 250
30 hours: $ 300
35 hours: $ 350
40 hours: $ 400
45 hours: $ 500
50 hours: $ 600
55 hours: $ 700
60 hours: $ 800
>>> paytable(24)
20 hours: $ 480
25 hours: $ 600
30 hours: $ 720
35 hours: $ 840
40 hours: $ 960
45 hours: $1200
50 hours: $1440
55 hours: $1680
60 hours: $1920
```

Submission

You should now have the `pa3` folder that contains the following five Python files:

```
quadruple.py        smallest.py
fibonacci.py        make_square.py
wages.py
```

Zip up the folder and submit the zipped file named as `pa3.zip` on Autolab.