CSC108H Assignment 1

Due Feb. 11, 2011 by 10 pm

Introduction

In this assignment you will build a module called `charlie` and you will write several functions for it. Like legendary comedian Charlie Chaplin, our module will be multitalented: it will include functions from Boolean logic, geometry and digit reasoning.

Download the seed file `charlie.py` from Blackboard. You will be filling it in as you go along.

Submission

You are allowed and encouraged to work in pairs on this assignment. You will be submitting it to MarkUs as outlined in class. You should **only** submit the file `charlie.py`. It should **not** execute any code when it's being imported. You are welcome to use the `__main__` block for testing purposes but make sure that when you submit, the only statements it contains are those required in Part 6. You are not allowed to import/use any other modules or to write additional functions when writing `charlie`.

Part 1: Boolean Logic

Boolean logic is a system of principles that governs operations done with Boolean values. So far, we are familiar with the logical operators NOT, AND and OR. There are three other operators that Boolean logic often uses in applications such as binary circuits:

NAND (Negated AND): This operation yields True iff at least one of the operands is False.

XOR (Exclusive OR): This operation yields True iff exactly one of the operands is True.

NOR (Negated OR): This operation yields True iff both operands are False.

These operations can be derived from NOT, AND and OR. Fill in the bodies of `nand()`, `xor()` and `nor()`: the corresponding function for each of these operations. Each function should take two Boolean values as arguments and return the appropriate result of the operation by using a combination of `and`, `or` and `not`.

Part 2: Universal Logical Operators

Some operators in logic have a special property: you can generate the result of any other logical operator just by combining operands using these special operators. These are known as **universal operators**. NAND is one such operator. It's possible to create the equivalent of any logical operation we've seen above, as well as NOT, AND and OR by using a combination of NAND operations. (By the way, two logical operations are equivalent if they have the same truth table, in other words, if they yield the same results for every combination of input values.)

For example, we can create an equivalent to NOT by computing the NAND of the operand with itself:

x	**NOT x**	**X NAND x**
True	False	False
False	True	True

If x is True, then True NAND True is False, which is the same result as NOT True.
If x is False, then False NAND False is True, which is the same as NOT False.

You will notice the `nand_not(a)` function is already implemented in the seed file.

Fill in the bodies of the following functions:

- `nand_or(a,b)` which takes two parameters and returns True iff at least one of them is True.
- `nand_and(a,b)` which takes two parameters and returns True iff both of them are True.

Inside the function you are **not allowed** to use comparison operators, conditionals (`if`s) or the keywords `not`, `and`, or `or`. You must only use calls to the `nand` function you wrote above (and, if you absolutely have to, assignment statements).

Part 3: Factors, Areas and Volumes

Fill in the bodies of the following functions and write docstrings for them. Unless otherwise specified, you should assume that the parameters can be integers or floats. Your functions should not lose precision and, unless otherwise specified, they should return **floats**:

`factor(a,b)`: takes as parameters two integers and returns True if a is divisible by b. 1 is a factor.

`area_triangle(b,h)`: takes as parameters the base and height of a triangle and returns its area.

`area_prism(h,w,d)`: takes the height, width and depth of a rectangular prism and returns its surface area.

`volume_prism(h,w,d)`: takes the height, width and depth of a rectangular prism and returns its volume.

The mathematical formulae for areas and volumes can be found online.

Part 4: Digit by Digit

In this part you will learn how to isolate digits in a four-digit number. You will need to complete the body of the function `digit_info(num)` that takes a 4-digit number, stores each of its digits in a variable and prints, on consecutive lines, the sum of the digits of the number, a breakdown of the number by digits, and the reverse of the number.

Example: To find the tens digit of 436:

1) Eliminate everything before the tens digit by computing the modulo of the 436 with 100:

```
436 % 100 = 36
```

2) Compute the integer division of the result with 10 to get the number of tens:

```
36 / 10 = 3
```

Fill in the bodies of the following functions:

`thousands(num)`: returns the thousands digit of `num`

`hundreds(num)`: returns the hundreds digit of `num`

`tens(num)`: returns the tens digit of `num`

`units(num)`: returns the units digit of `num`

*Note: Each function takes the four-digit number as an argument, and that's the only parameter it's allowed.

Part 5: Digit by digit (better)

You may have noticed that some of the four helper functions you wrote above contain code which looks quite similar. There is a general algorithm to obtain any digit of a number, and you have been using special cases of it.

You should complete the function called `digit(num, pos)` that takes two parameters: the 4-digit number and the position of the digit you'd like (0 for the units, 1 for the tens, 2 for the hundreds and 3 for the thousands). The function should return the selected digit as an `int`. It should **not** call any functions from Part 4 or use if statements. It should also work with a number of any length.

Finally, write a new function called `digit_info2` that does the same thing as `digit_info`, except it only calls the new function `digit`.

Part 6: Standalone `charlie`

While `charlie` contains useful functions that do various things, it doesn't do anything as a standalone program.

In the `__main__` block of the module, write code that asks for a number from the user. Your program should use `raw_input()` and work with any integer string entered by the user. You are not responsible for dealing with arbitrary strings that can't be converted to integers. If the number entered is not a positive integer **less than 100**, the program should print a helpful message and ask again until the user enters an appropriate integer. Once the program has a user-entered number in the desired range, it should store it in a variable called `user_num`.

Then, it should print the factors of the number on separate lines. A is a factor of B if B is evenly divisible by A, that is, if B `/` A has no remainder. The smallest possible factor a positive integer can have is 1. The largest factor of any positive integer is the integer itself.

Your program should check every number between 1 `and user_num`, in ascending order. It should print the number on its own line if it's a factor of `user_num`, and it shouldn't print anything if it's not. Your code should call the function `factor(a,b)` you wrote in Part 3.

Sample output:

```
Please enter a number: 4
1
2
4
```

Marking, Style and Specifications

These are the aspects of your work that we will focus on in the marking:

- **Correctness:** Your functions should perform exactly as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks.
- **Formatting style:** Make sure that you follow the formatting rules in the Assignment Style Guide.
- **Programming style:** Your variables' names should be meaningful, and your code as simple and clear as possible.
- **Docstrings**: Each function should have a docstring that describes its parameters, what the function does, and what is returned by the function, if anything.

There should be no console output from your program other than what is explicitly specified above, and there should be no calls to `raw_output()` anywhere inside a function definition. If a function unexpectedly outputs to the console, or asks for user input, you will receive 0 for correctness on that function. You should also not change the headers of functions definitions – they should only take the parameters specified.