CMSC 201 Fall 2017
Lab 03 – Simple Decisions

Assignment: Lab 03 – Simple Decisions
Due Date: **During discussion**, September 18th through September 21st
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

In Lab 2, you did some basic programming and learned how to find and fix errors in Python code. This week's lab will put into practice some of the material learned in class, including expressions, user input, Python's operators, and simple decision structures.
(Having concepts explained in a new and different way can often lead to a better understanding, so make sure to pay attention.)

Part 1A: Review – Using Variables and Expressions

Using variables in Python is easy! There are just two important rules we have to remember:

1. Use meaningful variable names! For example, **`numberOfBooks`** is a much better variable name than **`NOB`** or **`numb`** or **`x`**. Something like **`numBooks`** would also work, if you want to keep it a bit shorter.
2. Before we can use a variable, it must be ***initialized***. In other words, we have to put a value into the "box" before we can start using the variable. We do this using the ***assignment operator***, or equals sign (**=**).

An ***expression*** is code that calculates or produces new data and data values. Expressions are what allow us to create interesting Python programs. The word "expression" is really just a fancy name for something that can be evaluated to a single value.

One important thing to remember is that expressions must always be on the right hand side of the assignment operator!

Part 1B: Review – User Input and Casting

User input is a way to get information from the user after you've finished writing your program. Much like expressions, user input is an important step in creating Python programs that do interesting things.

The Python code to get input from the user will look something like this:

```
    userName = input("Enter your name please: ")
```

When your program is run, this code will print out the message **`"Enter your name please: "`** to the screen. After the user puts in their answer and hits enter, the text they entered will be stored as the value of **`userName`**.

If the user enters an integer, the value will be automatically stored as a string. However, we can’t do addition or multiplication with a string. (Python treats integers and strings very differently!)

We can fix this by telling the program that the input should be stored as an integer. Doing this is called ***casting***, a process in which Python changes a variable from one type to another. For example, if we want to convert the user’s age to an integer, we could write something like this:

```
    userAge = int(input("Enter your age please: "))
```

If we wanted their GPA (which would be a decimal number, which Python calls a float) we could write something like this:

```
    userGPA = float(input("Enter your GPA please: "))
```

Part 1C: Review – Comparison Operators

Mastery of logic is essential to understanding ***conditional statements***. It is used in pretty much any program that you will ever write. ***Comparisons*** are the heart of logical statements. When we write programs, we often want to compare two pieces of information, testing to see if that comparison evaluates to **`True`** or **`False`**.

We can make those comparisons using any of the following ***comparison operators***, which compare two pieces of information:

- **`<`** (less than)
- **`>`** (greater than)
- **`<=`** (less than or equal to)
- **`>=`** (greater than or equal to)
- **`==`** (equivalent to)
- **`!=`** (not equivalent to) also known as "bang" equals

For example:

```
num = 500                 # Set the value of num
num < 1000                # This evaluates to True
1456 >= num               # This evaluates to True
300 != 300                # This evaluates to False
"hello" == "goodbye"      # This evaluates to False
```

Notice how you can mix variables and "raw" data and still make valid comparisons. Unlike the assignment operator (**=**), it doesn't matter what goes on the left hand or right hand side of a comparison operator.

Part 1D: Review – Logical Operators

You can also combine two or more comparison statements by using:

- **and**
    - Both comparisons must be **True** for this to evaluate to **True**
- **or**
    - At least one comparison must be **True** for this to evaluate to **True**

For example:

```
num = 500
(500 <= num) and (num <= 1000)                # True
num > 487 or  num <= 342                      # True
num > 487 and num <= 342                      # False
("hello" == "hello") and ("dog" == "cat")    # False
"hello" == "hello" or "dog" == "cat"         # True
```

You do not have to use parentheses around a comparison statement, but it does have the benefit of making your code clearer and easier to read.

A third logical operator available to you is called **not.** This operates on one logical statement, "flipping" the truth value of that statement. So, a logical statement that is **True** will be flipped to **False**, and a logical statement that is **False** will be flipped to **True**.

For example:

```
isDog = True
not isDog                # False
"dog" == "cat"           # False
not ("dog" == "cat")     # True
(4 > 5)                  # False
not (4 > 5)              # True
5 > 4                    # True
not (5 > 4)              # False
```

Part 1E: Review – Decision Structures

Being able to make comparisons is only the first step. We also need a structure to execute different code based on the value of a comparison. There are three such structures available: "**if**", "**if-else**", and "**if-elif-else**". These are called ***decision structures***.

A basic "if" statement looks like this:

```
if age >= 65:
    print("If you are", age, "you are old.")
```

The **print()** statement is only executed if the value of the variable **age** is larger than or equal to 65. Whatever is "inside" the **if** statement (meaning one indentation level in) will be executed if the statement evaluates to **True**.

What if you want something different to happen if the logical statement is not **True**? To do this, just use an "**else**" statement right after an "**if**" like so:

```
if age >= 65:
    print("If you are", age, "you are old.")
else:
    print("If you are", age, "you are young.")
```

What if there are several exclusive logical statements you need to test? Simply use an "**elif**" statement combined with an "**if**."

```
if age >= 65:
    print("If you are", age, "you are old.")
elif age >= 45:
    print("If you are", age, "you are middle aged.")
elif age >= 25:
    print("If you are", age, "you're a young adult.")
else:
    print("If you are", age, "you are young.")
```

Important: The very first logical statement that evaluates to **True** will have its associated code executed, and *everything else will be skipped over*.
Also, you must have an "**if**" statement before you use any "**elif**" statements or an "**else**" statement.

Part 2: Exercises

In class, we've discussed using sequential and decision structures to control the "flow" of your code. Decision structures like **`if`**, **`elif`**, and **`else`** allow a Python program to execute a set of statements only if certain conditions are True (or False).

In this lab, you'll be creating two files: **`food.py`** and **`class.py`**, both of which will make use of comparisons and decision structures. Both files will be counted as part of the grade for Lab 3.

Tasks

- ☐ Create a **`food.py`** file from scratch
- ☐ Run and test your **`food.py`** file
- ☐ Create an **`class.py`** file from scratch
- ☐ Run and test your **`class.py`** file
- ☐ Show your work to your TA

Part 3A: Creating Your Files

First, create the **`lab03`** folder using the **`mkdir`** command -- the folder needs to be inside your **`Labs`** folder as well. *(If you need a reminder of how to create and navigate folders, try asking a classmate next to you for help. If you're both stuck, ask the TA or refer to the instructions for Lab 1.)*

Next, create two Python files (**`food.py`** and **`class.py`**) using the "**`touch`**" command in GL.

The "`touch`" command creates a new blank file, but doesn't open it.

Once a file has been "touched", you can open and edit it using emacs.

```
touch food.py
touch class.py
emacs food.py
```

The first thing you should do with any new Python file is create and fill out the comment header block at the top of your file. Here is a template:

```
# File:          FILENAME.py
# Author:        YOUR NAME
# Date:          9/TODAY/2017
# Section:       YOUR SECTION NUMBER
# E-mail:        USERNAME@umbc.edu
# Description:   YOUR DESCRIPTION GOES HERE AND HERE
#                YOUR DESCRIPTION CONTINUED SOME MORE
```

Part 3B: Breakfast Food (`food.py`)

This is the first of two programs that must be written for this lab.
This first program uses a simple **`if-else`** block, and compares strings for equivalence. First, the program asks the user what they ate for breakfast. If the input is “green eggs” or “ham” exactly, it should tell the user that they made an excellent choice.

Using a single **`if-else`** statement, check if the input matches “green eggs” or “ham” (in lowercase).

- If the input is “green eggs” or “ham” print:
  - Excellent choice!
- Otherwise, print:
  - FOOD is a strange choice for breakfast.
  - (Where FOOD is the food the user input.)

(Python is case-sensitive, so "ham" is not the same as "Ham" or "HAM" when comparing strings.)
Hint: Don’t forget that the Boolean operators "**`and`**" and "**`or`**" exist!

Here is some sample output, with the user input in **blue**.
(Yours does not have to match this word for word, but it should be similar.)

```
bash-4.1$ python food.py
Please enter what you ate for breakfast: Oatmeal
Oatmeal is a strange choice for breakfast.

bash-4.1$ python food.py
Please enter what you ate for breakfast: green Eggs
green Eggs is a strange choice for breakfast.

bash-4.1$ python food.py
Please enter what you ate for breakfast: green eggs
Excellent choice!

bash-4.1$ python food.py
Please enter what you ate for breakfast: ham
Excellent choice!
```

Part 3C: Course Advice (`class.py`)

This is the second of two programs that must be written for this lab.
This second program requires the use of slightly more complex decision structures, and is used to offer advice about a user's class.

The program should first ask the user for their professor's name. If they enter "Gibson" or "Neary" the program should ask them for the number of students in their class.

Using decision structures, have your program execute certain print statements following these rules:

- If they enter a professor name other than "Gibson" or "Neary"
  - "NAME doesn't teach CMSC 201! You're in the wrong class!" where NAME is the professor name entered
  - (and do NOT ask them for the number of students)

- If there are 80 students or more, print:
  - Wow, so many options for study groups!
- If there are less than 80 students, but at least 30 students, print:
  - You have so many chances to make new friends!
- If there are less than 30 students, print:
  - You'll get to know all of your classmates really well! :)

(See the next page for sample output.)

Here is some sample output for **`class.py`**, with the user input in **blue**. (Yours does not have to match this word for word, but it should be similar.)

```
bash-4.1$ python class.py
Enter your professor's name: Joshi
Joshi doesn't teach CMSC 201!  You're in the wrong class!

bash-4.1$ python class.py
Enter your professor's name: Neary
How many students in your class? 100
Wow, so many options for study groups!

bash-4.1$ python class.py
Enter your professor's name: Gibson
How many students in your class? 58
You have so many chances to make new friends!

bash-4.1$ python class.py
Enter your professor's name: Gibson
How many students in your class? 15
You'll get to know all of your classmates really well! :)

bash-4.1$ python class.py
Enter your professor's name: Neary
How many students in your class? 80
Wow, so many options for study groups!

bash-4.1$ python class.py
Enter your professor's name: Hrabowski
Hrabowski doesn't teach CMSC 201!  You're in the wrong
class!
```

Part 4: Completing Your Lab

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

Tasks

As a reminder, here are the tasks again:

- ☐ Create a **`food.py`** file from scratch
- ☐ Run and test your **`food.py`** file
  - ☐ If the user enters "green eggs" or "ham", they made an excellent breakfast choice
- ☐ Create a **`class.py`** file from scratch
- ☐ Run and test your **`class.py`** file
  - ☐ Print out advice based on the professor's name and the number of students
- ☐ Show your work to your TA

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!