Simon Fraser University
School of Computing Science

Lab Week 7

(No submission required for this lab.)

Objective

In this week's lab, our objective is to understand the following concepts and practise creating/using them in Python programs:

1. Boolean expressions and compound conditions
2. Conditional statements
3. Iterative statements (loops)
4. Lists

Notes

1. For this lab, you are free to work on our own or with a partner.
2. This is a long lab. If you do not have enough time to finish these lab exercises within our lab session, no problem! You can finish the lab on our own outside the lab session either
   a. by coming back to CSIL and using a CSIL workstation in a lab room where there is no scheduled lab taking place, or
   b. by using our own computer at home or on campus or elsewhere.
3. Let's make sure we show our work to one of the TA's in the lab to ensure we understand the concepts (see Objectives above) exercised in this lab.

Lab Prep

Once we have logged into a CSIL Windows workstation, let's access our **U: drive/CMPT120** and create a directory called **Lab4**. Let's store any files we create as part of this lab into this directory.

Exercise 1 – Boolean Expressions and Compound Conditions

1. Let's create a Python program by copying and pasting each Python code fragment below. Then execute each program and observe what it prints on the computer monitor screen.

   Python code fragment 1:

```
# lexicographic order among characters and strings
# (according to the ASCII code, which is part of UNICODE!)

print("a" < "b")
print("a" < "A")
print("aa" < "a")
print("a" < "ab")

messageabc = "abc"
messagexyz = "xyz"
messageabcLong = "abc!!!"
print(messageabc < messagexyz)
print(messageabc < messageabcLong)
```

   What is *lexicographical order*? It is the order used to organize the words in a dictionary.

   When comparing two strings in a Python program (or on the Python Interpreter shell), here is what happens: the ASCII (UNICODE) value (which is a number) of each character of the string on the left hand side of the comparison (relational) operator is compared with the ASCII (UNICODE) value of the corresponding character of the string on the right hand side of the operator.

For example, if we type the following command on the Python Interpreter shell `"abc" < "xyz"`, the Python Interpreter first compares the ASCII (UNICODE) value of the letter `"a"` (97) with the ASCII (UNICODE) value of the letter `"x"` (120). Since `"a"`(97) `<` `"x"` (120), it can answer the question `"abc" < "xyz"` right away (without having to compare the other characters of the strings) and it does so by returning the value `True`.

However, if we type the following command on the Python Interpreter shell `"abc" < "abd"`, the Python Interpreter first compares `"a" < "a"` and since the answer is `False`, it moves on to the next pair of characters. In other words, it cannot yet answer the question `"abc" < "abd"`.

The Python Interpreter then compares `"b" < "b"` and since the answer is `False`, it moves on to the next pair of characters. In other words, it cannot yet answer the question `"abc" < "abd"`.

Finally, the Python Interpreter compares `"c" < "d"` and since the answer is `True`, then it returns it as the answer.

Here is an ASCII table.

What about the other comparison operators: **==, !=, >, <, >=, <=** ? **Play around with them and see how they behave**.

Python code fragment 2:

```
# Boolean operators: and, or, not
zero = 0
one = 1
eleven = 11
print(eleven == 0)
print(not(eleven == 0))
print(zero == 0 and eleven == 11)
print(zero == 0 and eleven != 11)
print(one == 0 or eleven == 11)
print(zero == 0 and zero < one)
print(zero == 0 and < one )      # Why is this causing a
syntax error?
```

Python code fragment 3:

```
# a variable may contain a Boolean value!
threeLess = 3 < 4
threeEqual = (3 == 4)
print(type(threeLess))
print(threeLess)
print(threeEqual)
print(threeLess and not threeEqual)
print(threeEqual or True)
```

2. Are the following conditions (**Condition 1** and **Condition 2**) below equivalent (i.e, do they produce the same Boolean result for any value assigned to `userGuess`)?

   Condition 1

   ```
   userGuess >= 1 and userGuess <= 10
   ```

   Condition 2

   ```
   not (userGuess < 1 or userGuess > 10 )
   ```

   To answer this question:

   1. Let's create some test cases (5 of them). For each test case:
      a. Test Data: Select a specific test data for the variable `userGuess` (for example, `5`). (Let's use the technique we illustrated in class to select each of our 5 test data.)
      b. Expected Result: To compute the expected results for our selected test data, assign this test data to the variable `userGuess` and hand trace (execute/evaluate) **Condition 1** and **Condition 2** above using the **AND** and **OR Truth tables** found in our lecture notes. The results we obtain are the **expected result** for this test case. (For example, if `userGuess = 5`, then hand tracing **Condition 1** produces `True` and hand tracing **Condition 2 also** produces `True`.)

The following table may be useful to record our results. To illustrate the process, I have completed a few test cases:

Test Case #	Test data `userGuess`	Expected Result Condition 1	Expected Result Condition 2
1	5	True	True
2			
3			
4	1		
5	10		

2. In order to answer the question, have a look at the above table (once we have completed it) and observe whether both conditions produce the same expected result for each test data we assigned to `userGuess`. If this is the case, then, yes, **both conditions are equivalent**!

Exercise 2 – Conditional Statements

1. Some of the following Python code fragments are not semantically correct, i.e., the code does not correspond to the message it prints. Can we fix the problem?

   In order to answer this question, we may copy and paste the Python code fragments below into a Python program. If we do so, let's make sure the

indentation is preserved when the code fragments are pasted into a Python program.

Python code fragment 1:

```
# The user is well-behaved and will enter a positive integer
num = int(input("Please, enter a positive integer: "))
if num < 10:
    print("num is just 1 digit long.")
else:
    if num < 100:
        print("num is just 2 digit long.")
    else:
        print("num has more than 2 digits.")
```

Python code fragment 2:

```
# The user is well-behaved and will enter a positive integer
num = int(input("Please, enter a positive integer: "))
if num < 10:
    print("num is just 1 digit long.")
if num < 100:
    print("num is just 2 digit long.")
else:
    print("num has more than 2 digits.")
```

Python code fragment 3:

```
# The user is well-behaved and will enter a positive integer
num = int(input("Please, enter a positive integer: "))
if num < 100:
    print("num is just 2 digit long.")
else:
     if num < 10:
        print("num is just 1 digit long.")
     else:
        print("num has more than 2 digits.")
```

Python code fragment 4:

```
# The user is well-behaved and will enter a positive integer
num = int(input("Please, enter a positive integer: "))
if num < 10:
     print("num is just 1 digit long.")
elif num < 100:
     print("num is just 2 digit long.")
else:
```

```
        print("num has more than 2 digits.")
```

2. **Tricky:** Modify the following Python code fragment such that it is semantically correct.

   Hint: Have a look at the prompt!

   Python code fragment:

```
# The user is well-behaved and will enter an integer
num = int(input("Please, enter an integer: "))
if num < 10:
    print("num is just 1 digit long.")
else:
    if num < 100:
        print("num is just 2 digit long.")
    else:
        print("num has more than 2 digits.")
```

3. Write a Python program which must satisfy the following requirements:

   a) Our program asks the user for a Celsius temperature (an integer) value.

   b) If the temperature is **<=** 0 then our program prints "`Cold!`", if it is **>** 0 and **<** 10, our program prints "`Cool!`", if it is **>=** 10, our program prints "`Warm!`.

   We will create **two** versions of our program: **version 1** will use a nested conditional statement and our **version 2** will use a chained conditional statement.

   What are our test cases (test data and our expected results) for our programs? What is the minimum number of test cases we shall need in order to feel confident in our Python program?

4. Let's modify our Python program we wrote in the previous problem above (we can use either version 1 or 2, it is up to us) such that once it has asked the user for a temperature, it then asks the user whether there is precipitation or not (suggestion: "Is there precipitation?" and the user enters *yes* or *no* – careful, what if the user enters *Yes* or *No* or *YES* or *NO* or something that is not *yes* or *no*?).

When our program prints its message "`Cold!`", "`Cool!`" or "`Warm!`", if there is some precipitation, it also prints "`Snow!`" (if temperature <= 0) or "`Rain!`" otherwise.

Exercise 3 – Iterative Statements

1. Let's create a Python program by copying and pasting Python code fragment 1 below in a Python program. When we paste the fragments into a Python program, lets' make sure the indentation remains as it is in the Python code fragment below. Execute it and see what happens:

   Python code fragment 1:

```
for number in range(10):
    print(123)
    print("abc")

for number in range(10):
    print(123)
print("abc")
```

2. What is the result of the following Python code fragments?

   Python Code Fragment 1:

```
# with a string
word = 'Greetings'
for char in word:
    print(char, end=" ")
```

   Python Code Fragment 2:

```
# with a list
aList = [1,'b',300]
for element in aList:
    print(element, end=" ")
```

What is different between the Python code fragment in Problem1 and the Python code fragments in Problem 2? **Hint**: Look at the *sequence* used in the `for` loops.

Exercise 4 – Manipulating Lists and Building them using `for` loop

1. Write a Python code fragment that, given a list of integers, …

   a) extends this list by adding the value 100 at the end. For example, if `aList = [10,20,30]`, our code transforms it to `[10,20,30,100]`.

   b) extends this list by adding the value 100 in a certain position in the list. For example, if `aList = [10,20,30]`, and `position` (i.e., index) `= 2`, our code transforms it to `[10,20,100,30]`.

   c) adds 10 to every element of the list. For example, if `aList = [1,2,3]`, our code transforms it to `[11,12,13]`.

   d) add the position (index) to the value in each element in the list. For example, if `aList = [10,20,30]`, our code transforms it to `[10,21,32]`.

   e) removes the first element from a list.

   f) eliminates one element in a certain position (index) in the list. For example, given `aList = [10,20,30,40]`, and `position` (i.e., index) `= 2`, our code transforms it to `[10,20,40]`.

2. Construction of a list repeatedly

   To construct a list repeatedly, we could start with an empty list and append/insert repeatedly.

   Execute the following Python code fragments and see what they do:

   Python Code Fragment 1:

```
n = 6
newList = list( )   # or, newList = []
for element in range(n):
    newList.append(element)  # append at the end of list
print(newList)
```

Python Code Fragment 2:

```
n = 6
newList = list( )   # or, newList = []
for element in range(n):
    newList.insert(element, element) # insert at specified position
print(newList)
newList.insert(0,99)
print(newList)
newList.insert(20,26)
print(newList)
```

Python Code Fragment 3:

```
n = 4
newList = list( )   # or, newList = []
for element in range(n):
    newList.append(input("Please, enter a colour: "))
print(newList)
```

3. **None**

Because lists are mutable, we mentioned in class that it was important for us to become aware of which list functions and methods did modify the content of a list (and which ones did not), whether the list was an argument to the function or the object used to call the method. In this problem, we illustrate one of the reasons why being aware of these functions/methods is important.

As we can see from the above Python code fragments in Problem 2, the list methods `append( )` and `insert( )` modify `newList` without us having to reassign the result of these methods to `newList`. This is to say that, to insert an element into `newList`, all we have to do is this:

```
newList.insert(element, element)
```

We do not have to do this:

```
newList = newList.insert(element, element)
```

Actually, the above statement will destroy our `newList` i.e., its content.

To see why that is, execute the following Python code fragment and see what it does:

Python Code Fragment 3:

```
newList = [1,2,3,4,5]
newList = newList.append(8)
print(newList)
```

List methods such as **`insert()`**, **`append()`**, and **`remove()`** modify the list that calls them, i.e., the list on the left hand side of the "." (access or dot operator). These methods do not return anything (such as a list, a number, etc…) since they do not have to (because they have already modified the list).

When Python functions and methods do not return any value, the value `None` is returned. In the above Python code fragment, the `newList` is assigned the value `None` and here is why:

a. When `newList.append(8)` is executed, `8` is appended to the end of the `newList` and `None` is returned.
b. `None` is then assigned to `newList` when this part of the statement is executed: `newList =`, overwriting the previous content of `newList`.
c. The end result of the whole statement is that `newList` now contains `None`.
d. `None` is printed on the computer monitor screen when the last statement, i.e., the print statement `print(newList)` is executed.

Again, because lists are mutable, it is important for us to know which list functions and methods modify the content of a list (and therefore return `None`), and which ones do not (as we may need to assign their results to a list variable using the assignment operator).

Have fun!

Anne Lavergne – June 2017