Simon Fraser University

School of Computing Science

Lab Week 4

Objectives

In Lab 2, our objective is to practise:

1. Using Python building blocks learnt so far
2. Manipulating strings
3. Testing and debugging
4. Exploring if statements, which we shall further look at in our lectures soon.

Notes

1. For Week 4 Lab, we are free to work on our own or with a partner.
2. This lab is long. If we do not have enough time to finish the exercises within our lab session, no problem! We 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, or
    c. using online repl.it.
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 **Lab2**. Let's store any files we create as part of this lab into this directory.

Exercise 1 - Playing around with Python building blocks learnt so far

1. Let's do Exercises 2.1 and 2.2 found in Section 2.10 of **Chapter 2 - Variables, expressions and statements** of our online textbook, using the Python IDLE Interpreter shell for both questions.

   Note: For the above exercises, we are not asked to create Python programs. In Exercise 2.1, we are asked to "play around" with the Python IDLE Interpreter shell and notice what results we obtain, and in Exercise 2.2, we are asked to use the Python IDLE Interpreter shell as if it was a calculator to solve a few equations.

2. What is the difference between the 2 **Python code fragments** below? To answer the question, we can either **hand trace** the code or we can create a Python program using these code fragments.

   Python code fragment 1:

```
x = 12
print("original value of x is", x)
print("x/2 =", x/2)
print("printing x again ->", x)
```

Python code fragment 2:

```
x = 12
print("original value of x is", x)
print("x/2 =", x/2)
x = x/2
print("printing x again ->", x)
```

3. What do the following 2 Python code fragments produce? To answer the question, let's create a Python program using these code fragments and see what they produce.

   Python code fragment 1:

   ```
   exchangeRate = 1.23456
   print("Rounding exchangeRate %f" %exchangeRate,
   "to 2 significant figures produces", round(exchangeRate,2))
   ```

   Python code fragment 2:

   ```
   exchangeRate = 1.23654
   print("Rounding exchangeRate %f" %exchangeRate,
         "to 2 significant figures
   produces", round(exchangeRate,2))
   ```

4. What happens if we use the name of a function to name a variable? Let's try and see:

   ```
   print("round:", round)
   exchangeRate = 1.23456
   round = round(exchangeRate,2)
   print("round: %g" %round)
   # Call round() again - does it work???
   print(round(exchangeRate,2))
   ```

   ☹

5. Our task here is to create a small Python program in which we ask the user to enter a number of minutes. Our Python program then displays to the user the equivalent of these minutes expressed in hours and minutes so that the number of hours is the maximum number of hours possible, and the number of minutes is < 60.

Here are some sample runs. Hum… what is a sample run?

Definition: A **sample run** is the "snapshot" of what a program outputs on the computer monitor screen as it executes along with the result it prints on the computer monitor screen (if any) when it terminates. A sample run also contains what the user enters (often typed in bold font and/or underline – in the case below, the user input is in bold font and is underlined).

When we are given sample runs in a lab exercise, in an assignment, or in an exam question, the expectation is that we must create a Python program that produces the **exact same output** as the output represented by the sample runs if we, as user, enter the **exact same input**.

Sample Run 1: (what the user has typed is bolded and underlined)

```
Please, enter a number of minutes? 70
The equivalent time is: 1 hour(s) and 10 minute(s)
```

Sample Run 2: (what the user has typed is bolded and underlined)

```
Please, enter a number of minutes? 121
The equivalent time is: 2 hour(s) and 1 minute(s)
```

Exercise 2 – Playing around with Strings

1. Let's play around with strings. Copying the following Python code fragment into a Python program. Execute it and observe what happens. Feel free to play around with these Python statements by investigating what would happen if we were to modify any of these statements this way or that way.

```
word = "hello"
print(word.upper())
print(word)
```

```
print(word[0])
print(word[0].upper())
print(word.upper()[0])
print(word.upper()[0:2])
print(word.upper()[1:3])
aSlice = word.upper()[3:1]    # What happens here?
print(aSlice)
print(word[0:10])             # What happens here?
word = word.upper()
print(word)
print(word.isalpha( ))        # The answer is True – why?
anotherWord = "hello * 3"
print(anotherWord)
print(anotherWord[::])
print(anotherWord[2:])
print(anotherWord[0:5])
print(anotherWord[0::2])
print(anotherWord[3:len(anotherWord)-1])
theOperation = anotherWord[6]  # theOperation should
contain "*"
print(theOperation)
print(theOperation.isalpha( )) # The answer is False – why?
```

We may want to play with the other string built-in functions and methods, which we can find following the links in our lecture on strings.

2. **Incrementally Constructing a String** – Create a Python program by translating each of the following instructions of the algorithm described below (executing the instructions in the following order):

   a) Assigns the empty string to a string variable named varS

   b) Ask the user for a name

   c) Concatenate the name to the string in varS

   d) Print the content of the variable varS (at this point it should only contain the name)

   e) Ask the user for a number

   f) Concatenate the number (which should be a string) to varS, separating the previous content of varS from the number with a space

g) Display to the user the content of the variable varS (it should now show the name, a space and the number)

h) Ask the user for a code with special symbols

i) Concatenate this code to varS, separating the previous content of varS from the code with a space

j) Display the content of the variable varS

Note: Incrementally or cumulatively constructing a string is a very useful process we may encounter often while developing software.

3. Let's play with the `print()` built-in function (output statement) and escape sequences (or characters). Let's create a Python program using the **Python Code Fragment** 1 below and see what it produces.

Python Code Fragment 1:

```
name = "Louise MacLeavy"
income = 37500.2567
SIN = 123456789
age = 20
print("The employee record for %s
contains:\n\tSIN %i \tage %d\tsalary of $%0.2f" %(name,
SIN, age, income))
```

Let's make sure we understand what is happening in the above code fragment. Specifically, make sure we know what the following items do and why we would use them:

- the %s, %i, %d, %f
- the \t and the \n , which are called *escape sequences* or *escape characters* (what other escape sequences are there?)
- *old* string formatting mechanism: "$%0.2f" %(name, SIN, age, income). This string formatting mechanism is sometimes referred to as *printf* string formatting. It also has other names. Note that the 0in "$%0.2f" is the number zero 0, not the capital letter O.

While investigating the above, feel free to browse the Internet. Just remember to add “Python 3” to our query as Python 2 does printing differently than Python 3. Here is a table listing some escape sequences:

Source: https://docs.python.org/3.1/reference/lexical_analysis.html

In terms of the *old* string formatting, we may find the information described at this link very useful.

Is the Python code fragment below equivalent, i.e., outputs the same result, to the Python code fragment above?

Python Code Fragment 2:

```
name = "Louise MacLeavy"
income = 37500.2567
SIN = 123456789
age = 20
record = "\tSIN %i\tage %d\tsalary of $%0.2f" %(SIN, age,
income)
print("The employee record for %s contains:\n%s"
%(name,record))
```

Once again, let's make sure we understand what is happening in the above code fragment.

4. **Print Statement Challenge** - Our task is to write a Python program that asks the user to enter one letter, one digit and one special symbol (such as "!") using 3 separate `input()` statement, then, our program must …:

   a) display to the user each of the three values on a separate line using as many `print()` statements as needed.

   Hint: *string* formatting.

   b) display to the user the three values on the same line, each value separated by a space, using only 1 `print()` statement.

   Hint: *concatenation* operator.

   c) display to the user each of the three values on a separate line using only 1 `print()` statement.

   Hint: *newline* character and *concatenation* operator or *string* formatting (seen in Problem 3 above).

   d) display to the user the three values on the same line, however, there shall be ***no*** spaces between the three values, using only 1 `print()` statement.

   Hint: Look up the separator *sep=* argument to the `print()` built-in function.

   Note: Let's remember GPS (Good Programming Style): Let's make sure our variables are descriptively named, let's create clear and unambiguous prompts and let's clearly label our output.

Exercise 3 – Testing and Debugging

1. Let's have a look at this Python program (copy and paste it into a Python Program Editor new window):

```
# CtoF.py
# Celsius to Fahrenheit converter
#
# Anne Lavergne
# May 2015

print("Welcome to my Celsius to Fahrenheit converter!\n")

# Get temperature from user
celsius = float(input("Please, enter a Celsius temperature:
")

# Convert it to Fahrenheit
print("You have entered %0.1f celsius degrees, which is
%0.1f Fahrenheit degrees.\n" %(celsius,(( celsius * 2.4 ) +
32) ))

print("Bye!\n")
```

a) First, figure out what this program does? How will we be doing this? Hint: There are at least 2 ways we can figure out what this program does.

b) As we may have noticed, it contains a syntax error. Find the syntax error and fix it.

c) Testing

Once a program executes (i.e., is free of syntax errors), how do we test it? How do we know when a program solves the problem?

Yes! We can enter the requested number and pressed *Enter*. That is good! But, how do we know whether the output a program produces (with the number we entered) is correct?

And how many times (i.e., with how many different input values – test data) should we test our program? The answer is “at least once for each *category* of test data. What does ***category*** **of test data** mean? The test data for this program represents a Celsius temperature, which can be 0, positive or negative. Therefore, one can consider these 3 kinds of test data as the ***categories of test data***. So, when creating test cases, we would select at least one value (test data) from each of these 3 categories.

Test Cases:

As seen in class, let's come up with a few test cases. Complete the Table I (which has already been started for us):

Test Case #	Test Data	Expected Results
1	0	32
2		
3		

To test the program, we execute it, for each execution of the program, we enter one of the 3 test data and verify whether the program has produced (printed on the computer monitor screen) the expected result. If yes, youpi! and we move onto the next test data. If not, then something must be wrong with either our calculation of the expected result for this test case or something is wrong with the code of the program.

d) As we may have noticed, the program above does contain a semantic error. Find the semantic error and fix it. One way to find such error is to work backward from the `print( )` statement that displayed the erroneous output data on the screen, hand tracing the code of our program until we locate the problem.

Once we have fixed the semantic error of the program, we must **retest** it using all our 3 test cases all over again. Why would we need to do this?

Exercise 4 – Exploring Python Conditional Statements

1. Consider the following **Python code fragment**:

```
number = int(input("Please, enter a positive number :  "))
```

```
print("You have entered : ", number)
if number < 0:
    print(" ... which is not a positive number. So ... ")
    number = 0
print("Resulting number is : ", number)
calculation = number * 10
print("Result of calculation : ", calculation)
```

What does the **Python program** above accomplish when the user enters …

a) a **positive** number?

b) a **negative** number? (Even though the user is asked to enter a positive number.)

c) a **word**? (Even though the user is asked to enter a positive number.)

In order to answer the question, we need to create a Python program with the above Python code fragment and execute it 3 times, each time entering an input as described above in a), b) and c).

Note: The above Python program is not only an introduction to conditional statements but also to "guardian" code (also known as "error handling" code). More about this topic soon.