Carleton University
Department of Systems and Computer Engineering
ECOR 1041 - Computation and Programming

Lab 7 - Code that Makes Decisions

Objectives

- To gain experience developing Python functions and programs that perform different computations, depending on whether or not a condition is fulfilled.
- Introduction to automated unit testing

Overview

As usual, you will be writing a series of function definitions that are tested by call expressions in the test script. We will be changing the requirements for those call expressions though – we now want automated testing. Let me explain.

In previous labs: You had *manual* testing.

```
def silly_function( arg:int ) -> int :
    """ Returns the incremented value of arg
    >>> silly_function( 3 )
    4
    """
    return arg + 1

actual = silly_function( 3 )
print (actual)
```

Expected Value

Actual Value

It is “manual” testing because you yourself must stare at the screen in order to visually compare and verify in your head that the *actual* printed value to the *expected* value given in the docstring.

Now that you know about conditionals, you can write *automated* testing.

```
def silly_function( arg:int ) -> int :
    """ Returns the incremented value of arg
    >>> silly_function( 3 )
    4
    """
    return arg + 1

actual = silly_function( 3 )
print( "Testing silly_function(3) ")
print( "Expected result: 4, Actual result:”, actual)
if actual == 4 :
    print ("Test passed")
else:
    print ("Test failed”)
```

Expected Value

Automatic testing (i.e. comparison of expected with actual)

Launch Wing 101. All of the code for all the following exercises should be placed in the same file called lab7.py using the common file layout we have taught you (import, functions, main script).

As always, functions must be written with full docStrings and type annotations (The marking rubric requires them). **The main script must now demonstrate automated testing for each of the function**s – the print() output should readily pinpoint which test passed/failed, and if it failed, why.

Exercise 1 (…/10)

The purpose of this first exercise is to demonstrate your understanding of automated testing.

The factorial *n*! of a positive integer *n* is defined as:

$$n! \equiv 1 \times 2 \times \ldots \times (n - 1) \times n$$

Here is an incorrect definition of a function that calculates the factorial of its argument. Don't worry if you don't understand the function body: it uses Python constructs that have not yet been covered in class (i.e. for-loop), but that's not a problem, because you won't be correcting the function in this exercise.

```
def factorial(n: int) -> int:
    """Return n! for positive values of n.

    >>> factorial(1)
    1
    >>> factorial(2)
    2
    >>> factorial(3)
    6
    >>> factorial(4)
    24
    """
     fact = 1
     for i in range(2,n+1):
        fact = fact * n

    return fact
```

Copy-paste the above code into your program. Write the automated tests for this function as described in the Overview. Call the function `factorial` four times, with arguments 1, 2, 3, and 4, respectively. For each call, your code should determine whether the actual result (that is, the value returned by the call to the function `factorial`) matches the expected result (i.e., the value a correct implementation of `factorial` should return). The output produced by your program should look like this:

```
Testing factorial(1)
Expected result: 1 Actual result: 1
Test passed
```

```
Testing factorial(2)
Expected result: 2 Actual result: 2
Test passed
Testing factorial(3)
Expected result: 6 Actual result: 9
Test failed
Testing factorial(4)
Expected result: 24 Actual result: 64
Test failed
2 tests passed for Exercise 1
2 tests failed for Exercise 1
```

Notice the concluding two outputs. Somehow you have to keep track of how many tests passed and how many failed for this exercise (It's a programming problem that you have to solve)

Hints on Printing: By default, `print` displays its arguments separated by one space; for example,

```
>>> print('R', 2, '-', 'D', 2)
R 2 - D 2
```

If one of the arguments is `sep = "a_string"`, the values will be separated by `a_string`. In this example, three periods are printed between the values:

```
>>> print('R', 2, '-', 'D', 2, sep ='...')
R...2...-...D...2
```

If `sep` is assigned the empty string, `''`, no spaces are printed between the values:

```
>>> print('R', 2, '-', 'D', 2, sep ='')
R2-D2
```

Exercise 2 (…/35)

Use the function design recipe (FDR) to develop a function named `triangle_type` to check if a triangle is `acute`, `obtuse,` or `right`. The function takes three integers arguments which are the angles of the triangle. The function returns the triangle type, calculated as follows:

- A right triangle has one angle that is equal to 90°
- An obtuse triangle has one angle that is greater than 90°
- An acute triangle has all the angles that are less than 90°

Follow the same procedures to write and automatically test this function.

Exercise 3 (…/35)

A product cost is discounted based on the age of the customers. For example, if a customer's age is 60 years, the customer gets a discount worth 16% of the product price. The following table shows the percentage used to calculate the discount awarded for different customer's ages:

Customer age	**Discount Percentage**
less than 18 years	No discount
Between 18 (included) and 40 (not included) years old	5%
Between 40 and 60 years old	13%
Between 60 and 80 years old	16%
80 or more years old	20%

Use the function design recipe to develop a function named `discount`. The function has two parameters, the first parameter is the customer age and the second parameter is the product price. The function returns the value of the discount in dollars.

Hint: you need to code the conditions that determine the discount percentage. If you think carefully about the order in which the conditions will be executed, you'll see that you don't need to use the Boolean operators ("and", "or","not") in the conditions.

Final Exercise (…/20)

The term ***refactoring*** refers to the process of reviewing your code. AFTER you have written and successfully executed your code, ask yourself : Could I make it better?

In this lab, when you were writing the code for the automated testing of the previous exercises, did you end up writing a lot of repetitive code? Perhaps you copy-pasted the code to be more efficient. Did it seem repetitive?

Repetitive code is a symptom in program design that it is time to write a function.

Look again at your automated test code as well as the given example (copied again below). The code in red is meant to be a hint about what code is repetitive and can be moved into a function. The code is blue is meant to be a hint about the arguments that this function should have.

```
def silly_function( arg:int ) -> int :
    """ Returns the incremented value of arg
    >>> silly_function( 3 )
    4
    """
    return arg + 1

actual = silly_function( 3 )
print( "Testing silly_function(3)")
print( "Expected result: 4, Actual result:”, actual)
if actual == 4 :
    print ("Test passed")
else:
    print ("Test failed”)
```

While the syntax needed to solve this problem is simple, designing the function may be a difficult thought exercise. You may need help from your peers and the TAs.

1. Write the automated test function definition.
   - Use the name `test_int`
   - There should be three arguments (the blue clues in the sample above).
   - NOT SHOWN ABOVE – The function must return the integer 1 if the test passed, and return the integer 0 if the test failed. Can you figure out why this might be useful when calling it?
2. Call your new test function. Return to the code that you wrote for the previous exercises, starting with Exercise 1. Replace some of this code with calls to your new `test_int()` function. Ideally, the program runs exactly as it did before, printing out exactly the same messages. This is called *re-factoring your code*.

You may also replace the automated testing code for Exercise 2 and 3, but it is not mandatory. The functions in these two exercises return float numbers so you will need to write another version of the test function, but you have done enough work for this lab. We will save that work for another day. It would be a great exam question to ask you WHY a different version is needed.

Keep this function because you will need to re-use it in future labs.

Wrap Up

Ensure that your code meets the posted marking rubrics for the labs.

- Make sure that you included your name and student number
- Check proper use constants (UPPER_CASE) and variables (lower_case) (There is a 10/100 deduction for misuse of UPPPER & lower case)
- Check the indents of the function bodies. (There is a 10/100 deduction for misuse of indentation)
- Check file organization: (1) imports, (2) all function definitions; (2) Main Script (There is a 10/100 deduction for not organizing the file according to the instructions)
- Confirm that your filename matches exactly.
- Confirm that your .py script runs properly, otherwise, the TA will also assign a zero.
- Submit the file on Brightspace.

You are required to keep a backup copy of (all) your work for the duration of the term.

Last edited: January 25, 2022