

Carleton University
Department of Systems and Computer Engineering
ECOR 1051 - Fundamentals of Engineering I

Lab 6 - 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

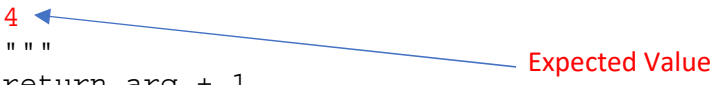
Learning outcomes: 2, 4, 5; Graduate attributes: 1.3, 5.3 (see the course outline)

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.

In previous labs: You had *manual* testing

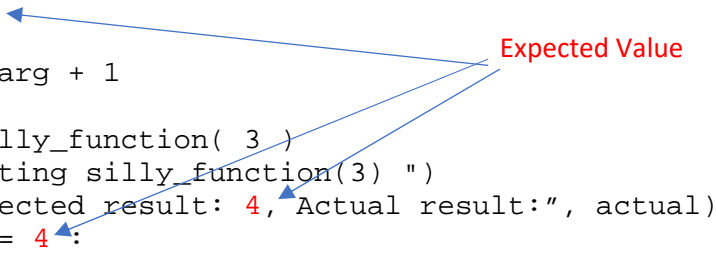
```
def silly_function( arg:int ) -> int :  
    """ Returns the incremented value of arg  
    >>> silly_function( 3 )  
    4  
    """  
    return arg + 1  
  
actual = silly_function( 3 )  
print (actual)
```

A blue arrow points from the red number '4' in the docstring to the text 'Expected Value'.

It is “manual” testing because you yourself have to stare at the screen to 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" )
```

Two blue arrows point from the text 'Expected Value' to the red number '4' in the docstring and the red number '4' in the conditional statement 'if actual == 4:'. A third blue arrow points from the text 'Expected Value' to the number '4' in the print statement 'print("Expected result: 4, Actual result:", actual)'.

Getting Started

Launch Wing 101. All of the code for all the following exercises should be placed in the same file 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 main script must now demonstrate automated testing for each of the functions – the `print()` output should readily pinpoint which test passed/failed, and if it failed, why.

Begin by creating a new file within Wing 101. Save it as `lab6.py`

Exercise 1

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:

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

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, 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 provided 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 (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` prints 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 (Based on an exercise by Cay Horstmann and Rance Necaise)

Calculating the tip when you go to a restaurant is not difficult, but a restaurant wants to suggest a tip based on the diners' satisfaction with the level of service they receive. Use the function design recipe (FDR) to develop a function named `tip`. The function's first argument is the cost of the meal. The second argument is satisfaction level. (Use these ratings: 1 = Totally satisfied, 2 = Somewhat satisfied, 3 = Dissatisfied). The function returns the amount of the tip, calculated as follows:

- If the diners are totally satisfied, calculate a 20% tip.
- If the diners are somewhat satisfied, calculate a 15% tip.
- If the diners are dissatisfied, calculate a 5% tip.

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

Exercise 3 (Based on an exercise by Cay Horstmann and Rance Necaise)

A supermarket awards coupons depending on how much a customer spends on groceries. For example, if you spend \$50, you'll get a coupon worth 8% of that amount (\$4). The following table shows the percentage used to calculate the coupon awarded for different amounts spent:

Amount Spent	Coupon Percentage
Less than \$10	No coupon
From \$10 to \$60	8%
More than \$60 to \$150	10%
More than \$150 to \$210	12%
More than \$210	14%

Use the function design recipe to develop a function named `coupon`. The function's argument is the amount spent on groceries. It returns the value of the coupon in dollars.

Hint: you need to code the conditions that determine the coupon 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 and not) in the conditions.

Final Exercise

The term *refactoring* refers to the process of reviewing your code – AFTER you’ve written and successfully executed your code – and asking yourself : Could I make it better?

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

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 put into a function. The code in 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" )
```

This may be a difficult exercise to do, and you may need help from your peers and the TAs.

1. Write the automated test function
 - 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. Replace your automated testing code for the factorial question in Exercise 1 with calls to your new `test_int()` function. Ideally, the program runs exactly like 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’ve done enough work for this lab. We’ll 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.

Submit file lab6.py.

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

Last edited: April 23, 2020