National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2016/2017

Solutions for Recitation 1
Introduction to CS1010X, Python & Functional Abstraction

Python

1. *if - elif - else:*

```
if expression:
    statement(s)
elif expression:
    statement(s)
else:
    statement(s)
```

   Consider each pre-condition *in sequence*, if the value of the any expression is not `False`, evaluate the corresponding statement(s). Otherwise evaluate the statement(s) under else.

2. Ternary Operator Form [To read only. Not expected to write code like that.]

   `[on_true] if [expression] else [on_false]`

   is equivalent to

```
if [expression]:
    [on_true]
else:
    [on_false]
```

Problems

1. Python supports a large number of different binary operators. Experiment with each of these, using arguments that are both integer, both floating point, and both string. Not all operators work with each argument type. In the following table, put a cross in the appropriate boxes corresponding to the argument and operator combinations that result in error.

Operator	Integer	Floating point	String
+			
-			X
*			X
/			X
**			X
//			X
%			X
<			
>			
<=			
>=			
==			
!=			

Note: * does work with one string input and one integer input. When the integer is non-positive, the result is an empty string.

2. Evaluate the following expressions assuming x is bound to 3, y is bound to 5 and z is bound to -2:

```
>>> x + y / z
```
0.5

```
>>> x ** y % x
```
0

```
>>> y <= z
```
False

```
>>> x > z * y
```
True

```
>>> y // x
```
1

```
>>> x + z != z + x
```
False

```
>>> if True:
    1 + 1
else:
    17
```
2

```
>>> if False:
    False
else:
```

```
    42
```
42

```
>>> if (x > 0):
    x
else:
    (-x)
```
3

```
>>> if 0:
    1
else:
    2
```
2

```
>>> if x:
    7
else:
    what-happened-here
```
7

```
>>> if True:
    1
elif (y>1):
    False
else:
    wake-up
```
1

3. Suppose we're designing an point-of-sale and order-tracking system for a new burger joint. It is a small joint and it only sells 4 options for combos: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie_sized* to acquire a larger box of fries and drink. A *biggie_sized* combo is represented by 5, 6, 7, and 8 respectively, for combos 1, 2, 3, and 4 respectively.

   (a) Write a function called `biggie_size` which when given a regular combo returns a *biggie_sized* version.

   ```
   def biggie_size(combo):
       return combo + 4
   ```

   (b) Write a function called `unbiggie_size` which when given a *biggie_sized* combo returns a non-*biggie_sized* version.

   ```
   def unbiggie_size(combo):
       return combo - 4
   ```

(c) Write a function called `is_biggie_size` which when given a combo, returns `True` if the combo has been *biggie_sized* and `False` otherwise.

```
def is_biggie_size(combo):
    return combo > 4
```

(d) Write a function called `combo_price` which takes a combo and returns the price of the combo. Each patty costs $1.17, and a *biggie_sized* version costs $.50 extra overall.

```
def combo_price(combo):
    extra = 0.5
    per_patty = 1.17
    if is_biggie_size(combo):
        return extra + per_patty * unbiggie_size(combo)
    else:
        return per_patty * combo
```

(e) An order is a collection of combos. We'll encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie_sized* Triple. Write a function called `empty_order` which takes no arguments and returns an empty order which is represented by 0.

```
def empty_order():
    return 0
```

(f) Write a function called `add_to_order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `add_to_order(1,2) -> 12`.

```
def add_to_order(order, combo):
    return order * 10 + combo
```