National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2020/2021

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.

Instructor notes: When demonstrating the operators, highlight the following:

- That `//` is floor division, not integer division. Integer division truncates the decimal portion, while floor division round down (floor) the number. Seems similar, but ask them to think about which situation would there be a difference.
- `%` works with floats in Python, but not necessary the case in other programming languages.
- Demonstrate the imprecision on floats. I like to use `a = 1/5` and show `a+a+a == 3/5` is False. Explanation is basically there is no precise way to represent all real numbers in any base system. Just like we cannot represent 1/3 in our decimal system, 1/3 as 0.3333333333 suppose given 10 digits. Then adding this 3 times will give you 0.9999999999. Take away is just to make them aware of this imprecision when dealing with real numbers.
- For string comparison, it is not just dictionary order, but ASCII order. I usually Google and show them an ASCII table and from there it is obvious that `"Z" < "a"`. So it's important to note that capitals come first because that might trip them up when sorting with strings later.
- As mentioned above, `*` works with string and integer.

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

Instructor notes: Usually there is time for me to go down the rows and get students to provide the answer.

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

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

Instructor notes: Here I will quiz them if they know the order of precedence of the operators. Truth is I also cannot remember so just use brackets to remove any doubt.

```
>>> 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
```

Instructor notes: Some students will find `if False:` confusing. Better to clarify after class than interrupt the flow.

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

>>> if 0:
    1
else:
    2
2

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

Instructor notes: Also demonstrate what happens if `x` is `False`. This will result in an error because `what-happended-here` is actually a subtraction of 3 variables. `-` is not a valid character in a variable name, so it is actually the minus operator.

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

Instructor notes: Again, here `wake-up` is `wake` minus `up`. It's meant to catch sleeping students and ask them to wake up.

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.

   Instructor notes: First show how it can be verbosely done:

   ```
   def is_biggie_size(combo):
       if combo > 4:
           return True
       else:
           return False
   ```

   Then show how `True` is returned when `combo > 4` is `True`, we can simply return the value of the expression `combo > 4` as below:

   ```
   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.

Instructor notes: Again, first show the "non-SLAP" way:

```
def combo_price(combo):
    if combo > 4:
        return 0.5 + 1.17 * (combo - 4)
    else:
        return 1.17 * combo
```

Then show how we can reuse functions previously defined. And extract out the "magic numbers" if there is time.

Emphasise that for our class, we will accept "unoptimal" code as they are still beginners.

```
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`.

Instructor notes: I would point out that there isn't actually any fixed order in an order (pun intended), and we can add the new combo to the left of the order. But doing so is actually not easy.

I like to say Math students will think of this way:

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

Instructor notes: And non-Math students will think of this way:

```
def add_to_order(order, combo):
    return int(str(order) + str(combo))
```

Then ask if this way will still work when `order` is initially `empty_order`. Answer is it will only work if `combo` is added to the right of `order`