CSC108 NOTES TEMPLATE 3 – MORE BASICS AND FUNCTIONS

1 Booleans

In **bio_calc.py**, we saw an example of an **if**-statement. The **if**-statement needs a condition: a yes/no value that determines which statements to perform next (e.g., **my_average >= 90**). We call a yes/no value a Boolean value. The 'yes' value is **True** and the 'no' value is **False**.

1.1 Comparison operators

Comparison operators: compare two values and give back a Boolean value.

```
Remark the "type" part last week!

>>>my_average >= 100
     True
>>>my_average >= 100
     False
>>># True and False are boolean values.
>>>type(True)
     <type 'bool'>
>>># Comparison operator: compare two values and give back a
Boolean value.
```

```
>>>3 < 4
     True
>>>3 < 8
     True
>>>3 > 8
     False
>>>3.8 < 2.5
     False
>>>x = 7
     y = 7.0
>>>x <= y
     True
>>>7 <= 7.0
     True
```

```
>>>7 == 7.0 # == is equality
     True
>>>7 == 7.1
     False
>>>7 != 7.1
     True
>>># != read as "not equal to"
```

1.2 Logical operators

Logical operators: operators that have two Boolean operands and give back a Boolean value.

```
>>># Logical operators: and, or ,not
>>>sunny = True
>>>snowing = False
>>>
>>>not sunny
     False
>>>not snowing
     True
>>>sunny and snowing
     False
>>># and: evaluates to True if and only if both operands are True
>>>True and True
     True
>>>True and False
     False
>>>False and True
     False
>>>False and False
     False
```

```
>>># or:evaluates to True if at least one of the operands is True
>>>True or True
     True
>>>True or False
     True
>>>False or True
     True
>>>False or False
     False
>>>a = False
     b = True
>>>not a or b # precedence?
    True
>>>not b or a
     False
>>>(not a ) or b
     True
>>>not (a or b)
     False
>>># not has highest precedence
>>>b or not a
    True
>>># not a or b is equilvant to b or not a
>>># -3 + y is equilvant to y + -3
```

```
>>>x = 6
>>>x < 0 or x > 5
    True
>>>x < 0 and x > 5
    False
>>>x <= 5
    False
>>>not (x > 6)
    True
>>>not (x > 5)
    False
>>>not (x >= 5)
    False
```

1.3 Comparison and Logical operators combined

To verify whether your two comparison operators are equal, like:
whether "x <= 5" is the same as "not (x > 5)", we should use three different x values to check, a value of x which is bigger than 5, a value of x which is 5 and a value of x which is smaller than 5. IF THE TO COMPARISON OPERATORS GET THE SAME RESULT UNDER THREE SITUATIONS EACH TIME, it is right.

2 Assignment vs Equality

[Basics slides 5-9]

THIS WEEK'S SLIDE (PAGE 9) shows the rule!

To test your understanding, predict what this does:

```
i = 50
j = -9
# Swap i and j
i = j
j = i
print i, j
```

-9 -9

Now predict what this does:

```
a = 87
b = 68
# Find the average of a and b
a + b = total
print total / 2
```

```
a = 87
b = 68
a + b = total
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
can't assign to operator: <string>, line 1
print total/2
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
NameError: name 'total' is not defined
```

why is wrong?

IT SHOULD BE
TOTAL = A + B!!!

caution: it is 77 instead of a 77.5 as it is a int not float!!!

3 Naming variables

[Basics slides 10-13]

4 Expressions vs. Statements

[Basics slides 14-15]

5 Textual input and output

[Basics slides 14-15]

5.1 print

Example program [**`expression.py`**]:

```
1. 4 * 24 / 2.5
```

When we run the module **expression.py**, we don't see the value of **4 * 24 / 2.5**. Python did evaluate the expression **4 * 24 / 2.5**, but we did not ask Python to print it!
In the shell, if we ask Python to evaluate the same expression, it shows us a value:

```
>>> 4 * 24 / 2.5
```

The shell does that as a courtesy. If we want to make a program (written in the editor) show the value of an expression, we need to tell it to **print**.

5.2 raw_input

raw_input is a function that prompts to user to enter textual input and gets what the user types.

```
"This is a string."
'This is a string.'
raw_input("What is your name?")
What is your name? Jen
' Jen'
name = raw_input("What is your name?")
What is your name? Jen
name
' Jen'
name = raw_input("What is your name?")
What is your name?Jen
name = raw_input("What is your name? ")
What is your name? 2012
name
'2012'
name + 1
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: cannot concatenate 'str' and 'int' objects
int(name)+1
2013
int(name)
2012
name + str(1)
'20121'
```

6 return vs. print

[Recap: Functions slides 2-5] [Functions slides 6-7]

[**hello.py**]

```
def say_hello():
    name = raw_input("What is your name? ")
     print "Hello " + name


def say_hello2():
    name = raw_input("What is your name? ")
    return "Hello " + name     # return exits this function.
    print "test" #this line is not executed.

say_hello()
say_hello2()
```

This will not show the string until we set a " message2 = say_hello2 ()". Comparatively, the same thing "message = say_hello" for the previous def has no effects.

7 Make sunset: new version using functions

[**make_sunset_functions.py**]

```
import media

def get_picture():
   filename = media.choose_file()
   pic = media.load_picture(filename)
   return pic

def get_sunset_pic():
   sunset_pic = media.copy(pic)

   for pixel in sunset_pic:
      value = media.get_green(pixel)  # Note: it should be pixel NOT
sunset_pic!
      new_green = int(value * 0.7)
      media.set_green(pixel, new_green)

      value = media.get_blue(pixel)   # Note: it should be pixel NOT
sunset_pic!
      new_blue = int(value * 0.7)
      media.set_blue(pixel, new_blue)
   return sunset_pic

pic = get_picture()
media.show(pic)
new_pic = get_sunset_pic(pic)
media.show(new_pic)
```

8 Nesting function calls

Example program [nesting_functions.py]:

```
def f(x):
     return x ** 2

def g(x):
     return x + 5
```

this part is just like the composition of functions!

9 `docstring`

We've used the **`help`** function to find out about built-in and media functions. We can provide information about the functions we write using a docstring. The docstring will be displayed when someone calls help on our function.

Let's add a docstrings to some of the functions we've already written. The notation is `'''`.

10 Practice writing functions

Let's write a function that figures out the total amount of green in a picture. [**`total_green.py`**]

```
import media
```

```
def total_green():
    '''(Picture) -> int
    Return the total amount of green in a Picture.'''

    total = 0
    for pixel in pic:
        total = total + media.get_green(pixel)
    return total
```

11 Reusing functions by importing them

Now we have several Python programs that manipulate images including **`make_sunset_functions.py`** and **`total_green.py`**. The **`get_picture`** function from **`make_sunset_functions.py`** might be useful in some other python program, such as **`total_red.py`**. We can import this file (called a module) into another program in the same way that we imported **`media`**. [**`importing.py`**]

```
import media
import make_sunset_functions

print 'some code that will call some functions from make_sunset_functions'
```

a small problem here???

12 Exercise: trace this code

```
def f(x):
    result = (x + y) ** 2
    return result

if __name__ == ¨__main__:̈
    a = 11
    b = 54
    answer = f(a, b / 8)
    print answer
```

13 Namespaces

There can be several variables with the same name in different places. In the code (**`namespaces.py`**) below, there are three different **`x`**'s: **`f`**'s **`x`**, **`g`**'s **`x`**, and **`main`**'s **`x`**. Let's trace this code.

```
def f(x):
    return x ** 2

def g(x):
    return x * (x + 1) / 2

if __name__ == ¨__main__:̈
    x = 13
    y = f(x)
    z = g(x)
    print x, y, z
```

```
def f(x) :
   return x ** 2

def g(x) :
   return x * (x + 1) / 2

x = 13
y = f(x)
z = g(x)
print x, y, z
```

[Functions slides 8-10]

14 Designing programs with functions

[Function slides 11-13]