Midterm 1 Review

- Important control structures
  - Functions
  - Loops
  - Conditionals
- Important things to review
  - Binary numbers
  - Boolean operators (and, or, not)
  - String operations: len, ord, +, *, slice, index
  - List operations
  - Libraries (import math vs from math import *)
  - Math operations (math.sqrt, %, /, //, **)

Material to Review

- Review text:
  - Part 1: chapters 3-6,
  - Part 2: chapters 8-9
- Go over your Pre Labs
- Go over your Lab Solutions
- Understand the results of the various clicker questions

= vs ==

- In Python we use “=“ for assignment
  - Example: x = 5
  - This sets the variable x to the value 5
- In Python we use “==“ to test equality
  - Example: print(x==5)
  - This will check if x is equal to 5 and then print True if it is, or False if it is not
  - Typically used in conjunction with an if statement

Functions

- Functions allow us to “name” a region of code in a similar fashion that a variable allows us to name a value
- Functions provide us a mechanism to reuse code

```
def name(input) :
    code to execute when function is called
    return (output)
```

Abstract:

```
def name(input) :
    code to execute when function is called
    return output
```

Concrete:

```
x = 3
y = 4
def myFun(a, b) :
    z = a + b            <--- this creates a local variable
    return z
x = myFun(x, y)          <--- this is the call site
print (x)                  <--- this prints 7
print (y)                  <--- this prints 4
```

Local vs Global Variables

- Functions introduce a new “scope”
  - This scope defines the lifetime of local variables
  - The scope is the function body

```
def name(input) :
    code to execute    <--- scope of local variables
    return output
```

Local vs Global Variables

```
a = 3
b = 4
def myFun(a, b) :
    b = a + b          <--- b is a local variable
    return b
a = myFun(a, b)        <--- this is the call site
print (a)                <--- this prints 7
print (b)                <--- this prints 4
```

Local vs Global Variables

```
a = 3
b = 4
def myFun(x, y) :
    global b
    b = x + y          <--- this assigns to the global variable
    return b
a = myFun(a, b)        <--- this is the call site
print (a)                <--- this prints 7
print (b)                <--- this prints 7
```

Important Concepts

- Only ONE return is ever executed
  - The return ends execution of the function
  - If there are statements after the executed return they are ignored!
- If there is no return that is executed the function returns the special python value: None
- The return specifies the value that the function "outputs"
  - If you return a variable the function outputs the value stored in that variable

Examples

```
def noReturn(a):
    x = a + 5
    y = x + 5
    print (y)
x = noReturn(10)
print (x)
```

```
def noReturn(a):
    if a > 30:
        return -1
    x = a +5
    y = x + 5
    print (y)
x = noReturn(10)
print (x)
```

Loops

- Loops allow us to execute a “set of actions” some number of times
  - The number of times is specified differently for the two types of loops
- There are two types of loops
  - For Loops and While Loops
- We call the execution of the body of the loop an ***iteration***

For Loops

- For loops allow us to execute the body of the loop once for each element in a sequence
  - A sequence can be either a Range, a List, or a String
- For loops assign to the variable each element of the sequence:
  - For a list each element of the list
  - For a string each character of the string
  - For a range each number in the range
  - The value of the variable changes with each iteration of the loop

for *variable* **in** *sequence* :
    code to execute for each element in the sequence

Abstract:

```
for variable in sequence :
    code to execute for each element in the sequence
```

Concrete:

```
str = “hello”
for char in str :
    print(char)
```

```
for num in [0,1,2,3,4,5] :
    print(num)
```

```
for p in range(10, 0, -2) :
    print(p)
```

While Loops

- While loops allow us to execute the body of the loop until the loop condition is false
- The loop condition is checked **prior** to the execution of the loop body (and on every iteration)

```
while (condition) :
    code to execute until condition is false
```

Abstract:

```
while (condition) :
    code to execute until condition is false
```

Concrete:

```
str = “hello”
i = 0
while (i < len(str)) :
    print(str)
    i = i +1
```

Nested Loops

- We know that each statement in the body of a loop is executed for each iteration
  - What happens if we have a loop nested within another loop?

```
for x in range(0, 10) :
    for y in range(0, 100):   < --- this loop is the body
        print (y)                          of the outer loop
    print (x)
```

Conditionals

- Conditionals allow us to test for a condition and execute based on whether the condition is True or False

```
if condition :
    code to execute if condition is True
else:
    code to execute if condition is False
```

Abstract:

```
if condition :
    code to execute if condition is True
else:
    code to execute if condition is False
```

Concrete:

```
If x==5 :
    x = 4
else:
    x = 3
print (x)
```

Things to remember

- The else clause is optional
- There MUST be a condition to check after an elif
  - The else clause is still optional here too
- Anything we can express with elif we can express with a nested if

```
if x < 10:
    print(“Hello”)
elif y > 30:
    print(“World”)
```

```
if x < 10:
    print(“Hello”)
else:
    if y > 30:
        print(“World”)
```

- Python interprets non-Boolean expressions when they appear in a conditional:

```
if x:
    <statements>
```

- [ ], 0, “” are all considered False
- Nonempty lists, nonempty strings, nonzero numbers are understood as True

Strings and Lists

- Strings are defined by quotation marks
  - Example: “this is a string”
  - Example: “””this is a string”””
- List are defined by []
  - Example: [0,1,2,3,4,5]
- Strings and lists are 0 indexed
  - We mean that the logically first element in either the string or list occurs at the 0^{th} position

Example: Reversing Strings

```
def reverse(str):
    output = ""
    for i in str:
        output = i + output
    print (output)
```

Indexing

- X[k] means:
  - Element k+1 in a list
  - Character k+1 in a string
- If lists are nested, we can refer to them by multiple indexing from outside in:

```
X = [1,2, [3,[4]],5]
X[2] == [3,[4]]
X[2][1] == [4]
X[2][1][0] == 4
```

Negative indexing

- For S a string or list:
  - S[0] == S[-len(S)]
  - S[1] == S[-len(S)+1]
  - …
  - S[len(S)-1] == S[-1]

Slice

- If S is a string, then S[3:7] is the substring beginning at character S[3] and ending with S[6] as the last character.
- len(S[3:7]) is 4
- If the lower bound is omitted, it is assumed to be 0
  S[:5] is the substring of length5 starting with S[0]
- If the upper bound is omitted, it is assumed to be len(S)

CQ: What is S[:] ?

A. S

B. S[0:0]

C. S[0:len(S)]

Slicing Lists

- Works just as slicing strings

List operations

- Let L be a list, L1 another list
- L.append(x) adds x as additional element to L
- L.reverse() reverses list L
- L[k] = v assigns value v as replacement value of L[k]
- L.insert(p,x) inserts x as additional element into L at position [p]
- L.remove(p) deletes L[p]
- L.index(k) is the same as X[k]

String operations

- Let S be a string
- S.upper() converts all alphabetic characters to upper case
- S.lower() converts all alphabetic characters to lower case
- S.capitalize() converts the first character to upper case and all other alphabetic characters to lower case
- S.reverse() throws an error