Assignment 1

B351 / Q351

Due: September 3, 2019 at 11:59 PM

This assignment is intended to bring the class to a baseline of proficiency with git, Python tools, syntax, and built in data structures.

We will be using Python 3 so be sure you aren't using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the above deadline without incurring a late penalty. No collaboration is allowed on this assignment.

1 Summary

- Familiarize self with the course tools and methodologies
- Refresh or learn basic Python syntax

2 Background

Before beginning to work on this assignment, please be sure that you have thoroughly read and understood the following files on the Canvas webpage:

- Git Installation and Configuration
- Python Installation and Configuration
- Assignment Overview

3 Programming Component

3.1 Problem 1 (5%)

The Fibonacci sequence is a sequence of numbers, where each number is the sum of the previous two numbers. the mathematical definition is as follows:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

Objective: Complete the `fib(n)` function to return the nth fibonacci number using recursion.

3.2 Problem 2 (10%)

Sequences are a family of data structures in Python including lists, tuples, and strings. A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ]. A tuple is

an immutable ordered sequence of elements, defined with values between parentheses ( ).

All sequences support the len built-in function and indexing using brackets:
`len((1,2)) => 2`
`'b351'[2] => '5'`

Objective: Complete the `firstLast(seq)` function to return the first and last items in a given sequence, seq. Items should be returned in a tuple format. If given a sequence of length 1, a tuple containing only one item should be returned.

3.3 Problem 3 (15%)

Objects in Python are useful for many things. One common reason you might write a Python class is to build a custom data structure.
For example, the following class defines a tree.

```
# A Node is an object
# - value : Number
# - subnodes : List of Nodes
class Node:
    def __init__(self, value, subnodes):
        self.value = value
        self.subnodes = subnodes
```

Here's an example of creating a tree object:

```
exTree = Node(1,[
Node(2,[]),Node(3,[
Node(4,[Node(5,[]),Node(6,[
Node(7,[])])])])]))
```

This is how `exTree` looks when drawn.

Objective: Complete the `sumNodesRec(root)` function so it returns the sum of the values at each node in the tree using recursion. For example, `sumNodesRec(exTree)` will return 28.

3.4 Problem 4 (20%)

This is a continuation of Problem 3, so you will use the `Node` class again.

Objective: Complete the `sumNodesNoRec(root)` function so it returns the sum of the values at each node in the tree without using recursion.

3.5 Problem 5 (15%)

There is another way besides `def` in Python to define a function. Here's an example:

```
f = lambda x: x**2
```

Now we can use f as follows:

f(2) = 4
f(10) = 100

We can do some interesting things with lambdas, such as having functions take functions as arguments, or return new functions. Here's a function which returns a function:

```
# makePowerFunction takes a number, and returns a
# function which will raise its argument to that power
makePowerFunction = lambda x: lambda y: y**x
```

This function can now be used to make new functions like this:

```
square = makePowerFunction(2)
cube = makePowerFunction(3)
square(3)
> 9
cube(3)
> 27
```

Objective: Complete the function `compose(f_outer, f_inner)`, which takes an outer function and an inner function, and returns a function which applies the outer function to the inner function to an argument.

3.6 Problem 6 (15%)

The `yield` keyword works similarly to `return`, in that it passes values to the code that called a function. However, it is used for generator functions (a type of iterable) that return a sequence of values without storing the entire sequence in memory. Every time `yield` is used, control flow passes back to the code that is iterating over the generator. Using yield is beneficial when you want a function to easily return multiple values without using too much memory.

Objective: Complete the `twice(iterable)` function, which takes any iterable (like a list, generator, or any data structure you can iterate through) and yields each element of the iterable twice. Your implementation should utilize the `yield` keyword.

For example, `twice([1, 2, 3])` would yield `1, 1, 2, 2, 3, 3`

3.7 Problem 7 (20%)

In a similar fashion to try/catch statements in Java, Python uses try/except for error handling.

Objective: Complete the `valid(iterable, function)` function, which takes an iterable and a black-box function and yields the returned value for each element of the iterable after applying the function to it, while ignoring any that raise a `ValueError` (do not handle any other exceptions). Your implementation should utilize try/except statements and the `yield` keyword.

For example, using the following black-box function `toHex`, which takes an integer and returns a string of its hexadecimal representation,

```
def toHex(value, minbytes=0, maxbytes=-1):
    if type(value) != int:
        raise ValueError('Integer expected.')
    hexValues = '0123456789abcdef'
    hexString = ''
    while (value or minbytes > 0) and maxbytes != 0:
        hexString = hexValues[value % 16] + hexString
        value //= 16
        minbytes -= .5
        maxbytes -= .5
    return hexString
```

`valid([255, 16, 'foo', 3], toHex)` would yield `'ff', '10', '3'`

3.8 Bonus Problem (10%)

Objective: Suppose you are given a tree. Write a function `treeToString(root)` which constructs a string with the values at each level of the tree on a new line. For example, using exTree from Problem 3, we would see the following:

```
print(treeToString(exTree))
> 1
> 23
> 4
> 56
> 7
```

4 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

Finally, remember that this is an individual assignment. The objective of this assignment is to ensure that you have enough programming knowledge to be successful in the later portions of this course. If you find this assignment particularly difficult, you may want to reach out to the instructors to discuss the best plan of action.

4.1 Problem 1 (5%)

Criteria	**Points**
Returns the correct values for base cases	2
Makes the correct recursion calls for input > 1	3
Total	5

4.2 Problem 2 (10%)

Criteria	**Points**
If the given sequence has one element, returns the tuple containing only that element	4
If the given sequence has more than one element, returns the tuple containing only its first and last elements	6
Total	10

4.3 Problem 3 (15%)

Criteria	**Points**
Adds the node's value to the sum	3
Recursively gets the totals for each of the node's subnodes	5
Sums up the subnodes' totals with the node's value	7
Total	15

4.4 Problem 4 (20%)

Criteria	**Points**
Does NOT use recursion	
For each node that is considered, adds its value to the total	3
For each node that is considered, also considers its subnodes	7
Returns the correct total sum of all the node's values	10
Total	20

4.5 Problem 5 (15%)

Criteria	Points
Returns a lambda function	5
The returned function invokes both `f_outer` and `f_inner`	3
The returned function applies `f_inner` before `f_outer`	4
When the returned function is applied to any input, it returns the result of composing `f_outer` with `f_inner`	3
Total	15

4.6 Problem 6 (15%)

Criteria	Points
Uses a `for` loop over the iterable	3
Uses the yield statement at least once in the `for` loop	7
Yields each element of the list exactly twice	5
Total	15

4.7 Problem 7 (20%)

Criteria	Points
Applies the function exactly once to each element of the iterable	5
Catches any `ValueErrors` that result from applying the function	5
Does NOT catch any other errors	5
Yields the result of applying the function to each valid input	5
Total	20

4.8 Bonus (10%)

Criteria	Points
Considers every node by generation – first the root, then its subnodes, then their subnodes...	2
Produces the desired output	8
Total	10