Question 1. [5 marks]
Read over the definition of this Python function:

```python
def c(s):
    """Docstring (almost) omitted.""
    return sum([c(i) for i in s]) if isinstance(s, list) else 1
```

Work out what each function call produces, and write it in the space provided.

1. c(5)
   1
2. c([])
   0
3. c(["one", 2, 3.5])
   3
4. c(["one", [2, "three"], 4, [5, "six"]])
   6
5. c(["one", [2, "three"], 4, [5, [5.5, 42], "six"]])
   8

Question 2. [5 marks]
Read over the declarations of the three Exception classes, the definition of raiser, and the supplied code for notice below. Then complete the code for notice, using only except blocks, and perhaps an else block.

```python
class SpecialException(Exception):
    pass

class ExtraSpecialException(SpecialException):
    pass

class UltraSpecialException(ExtraSpecialException):
    pass

def raiser(s: str) -> None:
    """Raise exceptions based on length of s.""
    if len(s) < 2:
        raise SpecialException
    elif len(s) < 4:
        raise ExtraSpecialException
    elif len(s) < 6:
        pass
```
raise UltraSpecialException
else:
    b = 1 / int(s)

def notice(s: str) -> str:
    """Return messages appropriate to raiser(s)."""
    try:
        raiser(s)
        # Write some "except" blocks and perhaps an "else" block
        # below that makes notice(...) have the behaviour shown in the docstring above
        except UltraSpecialException:
            return 'ultraspecialexception'
        except ExtraSpecialException:
            return 'extraspecialexception'
        except SpecialException:
            return 'specialexception'
        except Exception:
            return 'exception'
    else:
        return 'ok'

Question 3. [5 MARKS]

Read over the declaration of the class Tree and the docstring of the function two_count. Then complete the implementation of two_count

class Tree:
    """Bare-bones Tree ADT"""
    def __init__(self: 'Tree',
        value: object =None, children: list =None):
"""Create a node with value and any number of children""

    self.value = value
    if not children:
        self.children = []
    else:
        self.children = children[:]  # quick-n-dirty copy of list

def two_count(t: Tree) -> int:
    """Return number of times 2 occurs as a value in any node of t.
    precondition - t is a non-empty tree with number values
    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(2), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(2)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> two_count(tn1)
    3
    """
    return (1 if t.value == 2 else 0) + sum([two_count(c) for c in t.children])

Question 4.  [5 MARKS]

Complete the implementation of push in the class DividingStack, a subclass of Stack. Notice that you may use push, pop, and is_empty, the public operations of Stack, but you may not assume anything about Stack's underlying implementation. You may find it useful to know that if n1 and n2 are integers, then n1 % n2 == 0 if and only if n2 divides n1 evenly.

    from csc148stack import Stack
    """
    Stack operations:
    pop(): remove and return top item
    push(item): store item on top of stack
    is_empty(): return whether stack is empty.
    """

class DividingStack(Stack):
    """A stack of integers that divide predecessors."""

    def push(self: 'DividingStack', n: int) -> None:
        """Place n on top of self provided it evenly divides its predecessor.
        Otherwise, raise an Exception and leave self as it was before
        """
precondition – possibly empty self contains only integers

```python
>>> s = DividingStack()
>>> s.push(12)
>>> s.push(4)
>>> # now s.push(3) should raise Exception

```  
if not self.is_empty():
    last = self.pop()
    Stack.push(self, last)
    if not last % n == 0:
        raise Exception('{} does not divide {}'.format(n, last))
Stack.push(self, n)
```