Question 1.  [5 marks]

Read over the definition of this Python function:

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

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

1. `c(5)`
   - 5

2. `c([])`
   - 0

3. `c([1, 2, 3.5])`
   - 6.5

4. `c([1, [2, 3], 4, [5, 6]])`
   - 21

5. `c([1, [2, 3], 4, [5, [5.5, 42], 6]])`
   - 68.5

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:
```

```
```
raise UltraSpecialException
else:
    b = 1 / int(s)

def notice(s: str) -> str:
    """Return messages appropriate to raiser(s).
    >>> notice("123456")
    'ok'
    >>> notice("000000")
    'exception'
    >>> notice("12345")
    'ultraspecialexception'
    >>> notice("123")
    'extraspecialexception'
    >>> notice("1")
    'specialexception'
    ""
    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_whether. Then complete
the implementation of two_whether below. It may be helpful to know that the Python builtin function
any(L) returns True if list L contains at least one True element, and False otherwise.

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 WHETHER(t: Tree) -> bool:
    """Return whether at least one value in tree t is 2

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(7)])
>>> tn1 = Tree(1, [tn2, tn3])
>>> two WHETHER(tn1)
True
>>> two WHETHER(tn3)
False
""

    return t.value == 2 or any([two WHETHER(c) for c in t.children])

Question 4. [5 MARKS]

Complete the implementation of push in the class DescendingStack, 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.

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 DescendingStack(Stack):
    """A stack of integers in descending order.""

def push(self: 'DescendingStack', n: int) -> None:
    """Place n on top of stack self provided it is smaller than
its predecessor. Otherwise, raise an Exception and leave stack self as it was before.

precondition - possibly empty self contains only integers

>>> s = DescendingStack()
>>> s. push(12)
>>> s.push(4)
>>> # now s.push(5) should raise Exception
""

if not self.is_empty():
    last = self.pop()
    Stack.push(self, last)
    if not last > n:
        raise Exception('{} is not smaller than {}'.format(n, last))
    Stack.push(self, n)