Question 1. [8 MARKS]

Complete the following function without using a loop. You can use `len`, but no other built-in functions are allowed.

You *are* allowed to index into lists, take slices, and call `list` methods.

You can use `%` to get the remainder of a division in order to find out if a number is evenly divisible by another number.

Here are three solutions (all three have the same docstring):

```
def multiples(L, v):
    '''(list of ints, int) -> list of ints
    Return a list of values in L that are multiples of v, in the
    same order as they appear in L. L is not modified.'''
    res = []
    _multiples(L, v, 0, res)
    return res

def _multiples(L, v, i, res):
    '''(list of ints, int, int, list of ints) -> NoneType
    Append to res all the values in L[i:] that are multiples of v,
    in the same order as they appear in L. L is not modified.'''
    if i != len(L):
        if L[i] % v == 0:
            res.append(L[i])

        _multiples(L, v, i + 1, res)

def multiples2(L, v):
    if not L:
        return []
    elif L[0] % v == 0:
        return [L[0]] + multiples2(L[1:], v)
    else:
        return multiples2(L[1:], v)

def multiples3(L, v):
    res = []
    if not L:
        return res
    elif L[0] % v == 0:
        res.append(L[0])

    res.extend(multiples3(L[1:], v))
    return res

if __name__ == '__main__':
    for m in [multiples, multiples2, multiples3]:
        assert m([2, 4, 6], 2) == [2, 4, 6]
        assert m([1, 3, 5], 2) == []
        assert m([], 2) == []
        assert m([1, 2, 3, 4, 5, 6], 2) == [2, 4, 6]
```

Question 2. [10 MARKS]

You will be writing some functions and methods in this question. Write docstrings indicating the correct parameter types and return type of every function and method. You do *not* need to write the rest of the docstrings: you only need to indicate the various types.

Part (a) [5 MARKS]

Write a class called `Point` that represents a 2-D Cartesian point. `Point` objects should have two instance variables, `x` and `y`, which are set when each object is created. If either of the initial values passed to the `Point` class constructor is not an integer, the constructor should raise a `TypeError`. `Point` objects should show up in print statements as follows: `(<value for x>, <value for y>)`. For example, a `Point` whose `x` value is 3 and whose `y` value is 8 should print as `(3, 8)`.

Your `Point` class should work with the following testing code:

```
p = Point(3, 8)
assert p.x == 3
assert p.y == 8
assert '%s' % p == '(3, 8)'
try:
    q = Point('foo', 4)
    assert False
except TypeError:
    pass
```

```
class Point(object):
    def __init__(self, a, b):
        '''(Point, int, int) -> NoneType'''

        if type(a) != int or type(b) != int:
            raise TypeError
        self.x = a
        self.y = b

    def __str__(self):
        '''(Point) -> str'''

        return '(%d, %d)' % (self.x, self.y)
```

(Question continued on next page.)

Assume you are writing a class called `Triangle`. It has an instance variable `vertices`, which is a `list` of three `Point` objects. Also assume that you have written a function called `length` that takes two `Point` objects as parameters and returns the length of the line segment that joins them.

Part (b) [3 MARKS]

Write a method called `side_lengths` (to go inside your `Triangle` class) that returns a list of the lengths of the line segments formed by the `Point`s in the `Triangle`. Notice that you do not need to write the whole class; just write the `side_lengths` method.

```
def side_lengths(self):
    '''(Triangle) -> list'''

    return [length(self.vertices[i], self.vertices[(i + 1) % 3])
        for i in range(3)]

    # or, more verbosely:
    return [length(self.vertices[0], self.vertices[1]),
        length(self.vertices[1], self.vertices[2]),
        length(self.vertices[0], self.vertices[2])]
```

Part (c) [2 MARKS]

A set of three points forms a triangle if and only if the length of the longest side is strictly less than the sum of the lengths of the other two sides. Use this fact to write a method called `is_valid` (to go inside your `Triangle` class) that returns `True` if and only if the three `Point`s form a geometrically valid triangle.

```
def is_valid(self):
    '''(Triangle) -> bool'''

    s = self.lengths()
    s.sort()
    return s[2] < s[1] + s[0]
```

Part (d) [1 MARK]

Write an `InvalidTriangleError` exception class as simply as you can.

```
class InvalidTriangleError(Exception):
    pass
```

Part (e) [2 MARKS]

Write a function called `perimeter` that takes a `Triangle` as a parameter and returns the sum of the sides of the `Triangle`. If the triangle is not a valid triangle, raise an `InvalidTriangleError`; you do not need to supply an error message.

```
    def perimeter(self):
        '''(Triangle) -> number'''

        if self.is_valid():
            return sum(self.lengths())
        else:
            raise InvalidTriangleError()
```

Question 3. [8 MARKS]

Two functions `f` and `g` are *mutually recursive* if the body of `f` depends on a call to `g` and the body of `g` depends on a call to `f`.

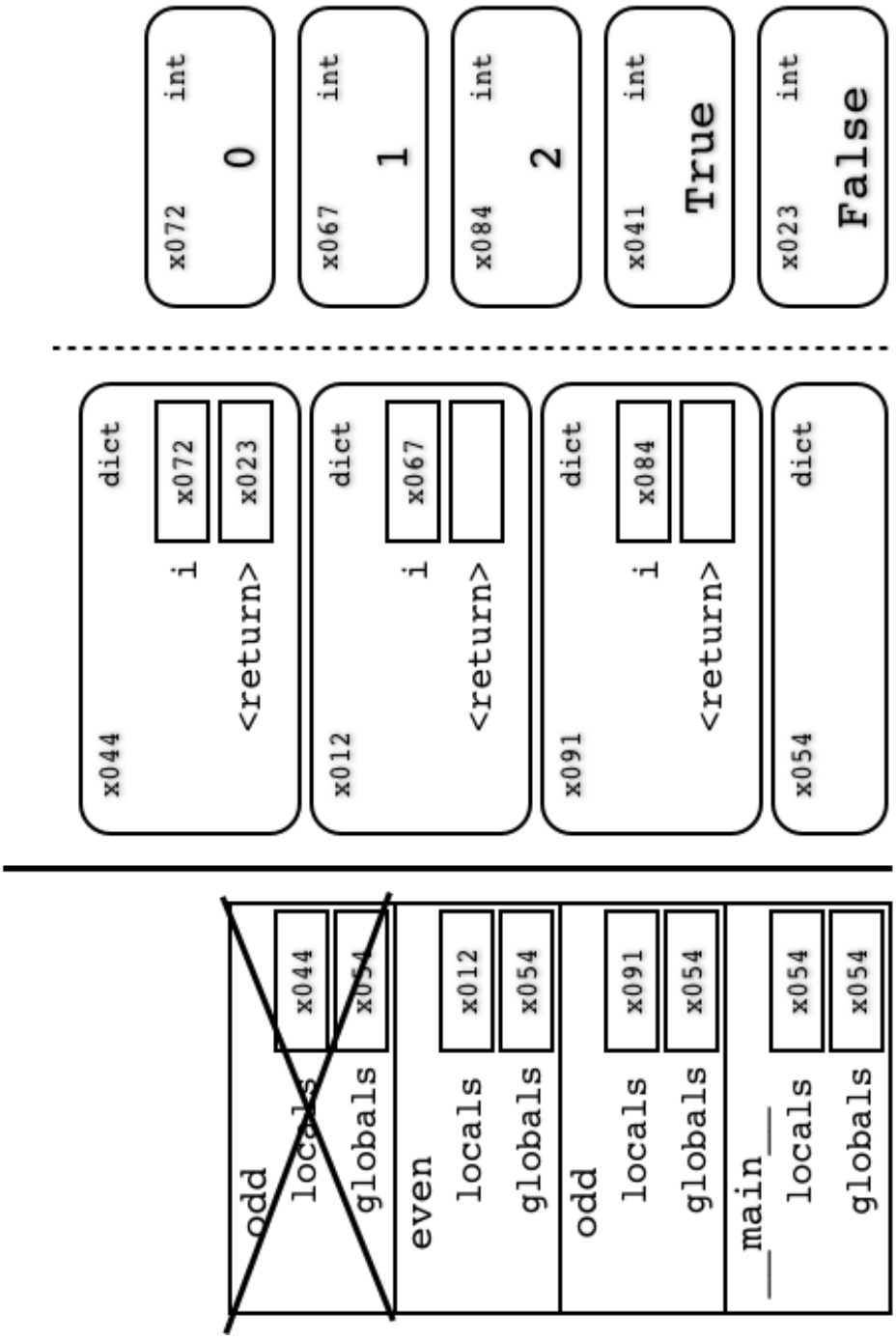
Consider this code:

```
def even(i):
    if i == 0:
        return True
    else:
        return odd(i - 1)

def odd(i):
    if i == 0:
        return False
    else:
        return even(i - 1)

odd(2)
```

On the opposite page is a memory model diagram in the style we have used in lecture. The diagram represents the contents of memory just after `odd` has been called. The `if` statement in `odd` is about to be executed.

Complete the memory model diagram for the call to `odd(2)` up until one return statement has fully executed. This statement will pop the top frame off the call stack; indicate this by drawing a big "X" through the frame. As soon as you draw that "X", stop tracing.

Create any boxes that you think are needed. (There are no global variables, so we have not shown the global namespace.)


x072 int 0
x067 int 1
x084 int 2
x041 int True
x023 int False
x044 dict i x072 <return> x023
x012 dict i x067 <return>
x091 dict i x084 <return>
x054 dict
odd locals x044 globals x054
even locals x012 globals x054
odd locals x091 globals x054
__main__ locals x054 globals x054