Question 1:

Answer

```
import random

# Constants


WIDTH = 20

HEIGHT = 10

SNAKE_START_LENGTH = 5

INITIAL_DELAY = 500  # milliseconds

# Variables


snake = [(0, HEIGHT - 1 - i) for i in range(SNAKE_START_LENGTH)]

direction = "right"

food = None

delay = INITIAL_DELAY

# Helper functions


def print_board():
    board = [[" " for _ in range(WIDTH)] for _ in range(HEIGHT)]
    # Add snake to board
    for x, y in snake:
        board[y][x] = "S"
    # Add food to board
    if food:
        x, y = food
        board[y][x] = "F"
    # Print board
    print("-" * (WIDTH + 2))
    for row in board:
        print("|" + "".join(row) + "|")
    print("-" * (WIDTH + 2))


def move():
    global direction, snake, food, delay
    # Move snake in current direction
```

```
    x, y = snake[-1]
    if direction == "up":
        y -= 1
    elif direction == "down":
        y += 1
    elif direction == "left":
        x -= 1
    elif direction == "right":
        x += 1
    snake.append((x, y))
    # Check if snake hit wall
    if x < 0 or x >= WIDTH or y < 0 or y >= HEIGHT:
        print("Game over - you hit the wall!")
        exit()
    # Check if snake hit itself
    if (x, y) in snake[:-1]:
        print("Game over - you hit yourself!")
        exit()
    # Check if snake ate food
    if (x, y) == food:
        food = None
        delay *= 0.9  # Increase speed
    else:
        snake.pop(0)
    # Place new food if necessary
    if not food:
        while True:
            x = random.randint(0, WIDTH - 1)
            y = random.randint(0, HEIGHT - 1)
            if (x, y) not in snake:
                food = (x, y)
                break

# Game loop
while True:
```

```
    print_board()
    move()
    # Wait for user input or delay
    try:
        key = input("Press arrow keys to change direction, or any other key to quit\n")
        if key == "\x1b[A":
            direction = "up"
        elif key == "\x1b[B":
            direction = "down"
        elif key == "\x1b[D":
            direction = "left"
        elif key == "\x1b[C":
            direction = "right"
        else:
            print("Game over - you quit!")
            exit()
    except KeyboardInterrupt:
        print("Game over - you quit!")
        exit()
    time.sleep(delay / 1000)
```

Question 2:

Answer:

```
import xml.etree.ElementTree as ET


def parseXmlToDict(xmlString):
    root = ET.fromstring(xmlString)


    def parseElement(element):
        obj = {}
        for child in element:
            if len(child) == 0:
                obj[child.tag] = child.text
            else:
                obj[child.tag] = parseElement(child)
```

```
        return obj


    return {root.tag: parseElement(root)}
```

This function takes an XML string and returns a dictionary representing the XML structure. It uses the xml.etree.ElementTree module to parse the XML string into an ElementTree object. It then recursively builds up a Python dictionary that represents the XML structure. The function returns a dictionary with a single key-value pair, where the key is the tag of the root element and the value is the parsed dictionary.

We can call the function like this:

```
xml = '''
 <node>
   <hello>World</hello>
   <nested>
    <bloop>Bleep</bloop>
   </nested>
 </node>
'''


result = parseXmlToDict(xml)


print(result)
# Output: {'node': {'hello': 'World', 'nested': {'bloop': 'Bleep'}}}
```

Question 3:

Answer:

```
class FileSystem:
    def __init__(self):
        self.files = {}

    def mkdir(self, path):
        self.files[path] = {}

    def writeFile(self, path, data):
        parts = path.split('/')
        filename = parts.pop()
        curr = self.files
```

```
        for part in parts:
            if part not in curr:
                raise ValueError("Parent directory doesn't exist")
            curr = curr[part]
        curr[filename] = data

    def readFile(self, path):
        parts = path.split('/')
        filename = parts.pop()
        curr = self.files
        for part in parts:
            if part not in curr:
                raise ValueError("Path doesn't exist")
            curr = curr[part]
        if filename not in curr:
            raise ValueError("Path is a directory")
        return curr[filename]
```

This class implements an in-memory file system with a root directory (/) and supports the mkdir, writeFile, and readFile methods. The mkdir method creates a directory at the specified path by traversing the directory tree and creating any missing directories along the way. The writeFile method writes a file to the specified path by first traversing the directory tree to the parent directory and then creating the file node with the specified data. The readFile method reads the contents of a file at the specified path by traversing the directory tree to the parent directory and then returning the contents of the file node with the specified name.

Here's an example usage of the class:

```
fs = FileSystem()
fs.mkdir('/foo')
fs.writeFile('/foo/bar', 'hello, world')
print(fs.readFile('/foo/bar'))  # Output: 'hello, world'
```

Question 4:

Answer:

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
```

```
        self.right = None

def find_extra_edge(root):
    visited = set()
    parent = {}
    extra_edge = None

    def dfs(node):
        nonlocal extra_edge
        visited.add(node)
        if node.left:
            if node.left in visited:
                extra_edge = (node, node.left)
            else:
                parent[node.left] = node
                dfs(node.left)
        if node.right:
            if node.right in visited:
                extra_edge = (node, node.right)
            else:
                parent[node.right] = node
                dfs(node.right)

    dfs(root)
    if not extra_edge:
        return None

    # Remove extra edge
    child, parent = extra_edge
    if child == parent.left:
        parent.left = None
    else:
        parent.right = None
    return extra_edge
```

Here's an example usage of the function:

```
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Add an extra edge that violates the tree property
root.right.right.right = root.left.right

extra_edge = find_extra_edge(root)
if extra_edge:
    print(f"Extra edge: {extra_edge[0].val} -> {extra_edge[1].val}")
else:
    print("No extra edge found")

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Add an extra edge that violates the tree property
root.right.right.right = root.left.right

extra_edge = find_extra_edge(root)
if extra_edge:
    print(f"Extra edge: {extra_edge[0].val} -> {extra_edge[1].val}")
else:
    print("No extra edge found")
```

Question 5:

Answer:

```
def deepEquals(a, b):
    # Base case: check if a and b are identical primitives
    if type(a) != type(b):
        return False
    if isinstance(a, (int, float, complex, str, bool, type(None))):
        return a == b
    if isinstance(a, (list, tuple)):
        if len(a) != len(b):
            return False
        for i in range(len(a)):
            if not deepEquals(a[i], b[i]):
                return False
        return True
    if isinstance(a, dict):
        if len(a) != len(b):
            return False
        for key in a.keys():
            if key not in b:
                return False
            if not deepEquals(a[key], b[key]):
                return False
        return True
    # For all other types, just compare by identity
    return a is b
```

The deepEquals function recursively checks if two values are identical. It handles all the types listed in the problem statement by using isinstance checks to determine how to compare them.For primitives, it simply checks if they are equal using ==. For lists and tuples, it checks if they have the same length, and then recursively checks if each element is equal. For dictionaries, it checks if they have the same keys, and then recursively checks if the value for each key is equal. For all other types, it simply checks if they are identical by using the is operator.

Here are some example usages of the function

```
# Primitives
assert deepEquals(1, 1)
assert not deepEquals(1, 2)
```

```
assert deepEquals("foo", "foo")
assert not deepEquals("foo", "bar")
assert deepEquals(True, True)
assert not deepEquals(True, False)
assert deepEquals(None, None)
assert not deepEquals(None, 0)

# Lists
assert deepEquals([], [])
assert deepEquals([1, 2, 3], [1, 2, 3])
assert not deepEquals([1, 2, 3], [3, 2, 1])
assert not deepEquals([1, 2], [1, 2, 3])

# Tuples
assert deepEquals((), ())
assert deepEquals((1, 2, 3), (1, 2, 3))
assert not deepEquals((1, 2, 3), (3, 2, 1))
assert not deepEquals((1, 2), (1, 2, 3))

# Dictionaries
assert deepEquals({}, {})
assert deepEquals({1: "foo", 2: "bar"}, {1: "foo", 2: "bar"})
assert not deepEquals({1: "foo", 2: "bar"}, {2: "bar", 1: "foo"})
assert not deepEquals({1: "foo"}, {1: "bar"})

# Mixed types
assert not deepEquals(1, [1])
assert not deepEquals([1], (1,))
assert not deepEquals({"foo": [1, 2]}, {"foo": [1, 3]})
assert not deepEquals({"foo": [1, 2]}, {"bar": [1, 2]})
```