1. Water Jug Problem

```python
def water_jug_problem(jug1, jug2, target):
    x = 0
    y = 0
    x = jug1
    while True:
        t = min(x, jug2 - y)
        y += t
        x -= t
        print(f"Jug1: {x}L, Jug2: {y}L")
        if x == target or y == target:
            return True
        if y == jug2:
            y = 0
        if x == 0:
            x = jug1
    return False

jug1 = 4
jug2 = 3
target = 2

if water_jug_problem(jug1, jug2, target):
    print("Solution exists!")
else:
    print("No solution exists.")
```

Sure, let's break down the code:

1. `def water_jug_problem(jug1, jug2, target):`: This line defines a function called `water_jug_problem`, which takes three arguments: `jug1` and `jug2` are the capacities of the two jugs, and `target` is the desired amount of water.
2. `x = 0 # Initial amount of water in jug1`: This line initializes a variable `x` to represent the current amount of water in the first jug. Initially, this jug is empty, so `x` is set to 0.

3. `y = 0 # Initial amount of water in jug2`: Similarly, this line initializes a variable `y` to represent the current amount of water in the second jug. This jug is also initially empty, so `y` is set to 0.
4. `x = jug1`: This line fills the first jug to its maximum capacity.
5. `while True:`: This line starts an infinite loop. The loop will continue until it is explicitly broken out of.
6. `t = min(x, jug2 - y)`: This line calculates the amount of water to be transferred from the first jug to the second. This is the minimum of the amount of water currently in the first jug and the remaining capacity of the second jug.
7. `y += t`: This line adds the amount of water `t` to the second jug.
8. `x -= t`: This line subtracts the amount of water `t` from the first jug.
9. `print(f"Jug1: {x}L, Jug2: {y}L")`: This line prints the current state of the jugs.
10. `if x == target or y == target: return True`: This line checks if either jug contains the target amount of water. If so, the function returns `True` to indicate that a solution has been found.
11. `if y == jug2: y = 0`: If the second jug is full, this line empties it.
12. `if x == 0: x = jug1`: If the first jug is empty, this line fills it.
13. `return False`: If the function has not returned `True` after all possible states have been explored, it returns `False` to indicate that no solution exists.
14. The last few lines of the script call the `water_jug_problem` function with specific values and print the result.

---

2. Tic-Tac-Toe

```
def tic_tac_toe():
    board = [' '] * 9
    player = 'X'

    while True:
        print(' '.join(board[0:3]))
        print(' '.join(board[3:6]))
        print(' '.join(board[6:9]))
```

```
        print(f"Player {player}, enter your move (1-9):")
        move = int(input()) - 1
        if board[move] != ' ':
            print("Invalid move, try again.")
            continue
        board[move] = player
        if board[0] == board[1] == board[2] != ' ' or \
           board[3] == board[4] == board[5] != ' ' or \
           board[6] == board[7] == board[8] != ' ' or \
           board[0] == board[3] == board[6] != ' ' or \
           board[1] == board[4] == board[7] != ' ' or \
           board[2] == board[5] == board[8] != ' ' or \
           board[0] == board[4] == board[8] != ' ' or \
           board[2] == board[4] == board[6] != ' ':
            print(f"Player {player} wins!")
            break
        player = 'O' if player == 'X' else 'X'
        if ' ' not in board:
            print("It's a draw.")
            break

tic_tac_toe()
```

Sure, let's break down the code:

1. `def tic_tac_toe():`: This line defines a function called `tic_tac_toe`. This function will run the game.
2. `board = [' '] * 9`: This line creates a list of 9 spaces. This list represents the game board.
3. `player = 'X'`: This line sets the initial player to 'X'.
4. `while True:`: This line starts an infinite loop. The game will continue until a break statement is encountered.
5. The next three lines print the game board. The `join` function is used to convert the list of characters into a string.
6. `move = int(input()) - 1`: This line asks the current player to enter their move. The move is a number from 1 to 9, which corresponds to the positions on the board from top left to bottom right. The `- 1` is used because list indices in Python start at 0.
7. `if board[move] != ' ':`: This line checks if the chosen position on the board is already occupied. If it is, the player is asked to enter their move again.

8. `board[move] = player`: This line places the current player's mark at the chosen position on the board.
9. The next several lines check if the current player has won the game. The game is won if any row, column, or diagonal contains three of the same mark.
10. `player = 'O' if player == 'X' else 'X'`: This line switches the current player.
11. `if ' ' not in board:`: This line checks if the board is full. If it is, the game is a draw, and the loop is broken.
12. `tic_tac_toe()`: This line runs the game.

---

3. Constraint Satisfaction Problem (CSP)

```
from constraint import *
```

```
def map_coloring():
    problem = Problem()

    # Define the variables (regions)
    regions = ['A', 'B', 'C', 'D']
    problem.addVariables(regions, ['red', 'green', 'blue'])

    # Define the constraints (adjacent regions cannot have the
same color)
    problem.addConstraint(AllDifferentConstraint(), ['A', 'B'])
    problem.addConstraint(AllDifferentConstraint(), ['A', 'C'])
    problem.addConstraint(AllDifferentConstraint(), ['B', 'C'])
    problem.addConstraint(AllDifferentConstraint(), ['B', 'D'])
    problem.addConstraint(AllDifferentConstraint(), ['C', 'D'])

    # Get and print solutions
    solutions = problem.getSolutions()
    for i, solution in enumerate(solutions):
        print(f"Solution {i+1}: {solution}")

map_coloring()
```

Sure, let's break down the code:

1. `from constraint import Problem, NotEqualsConstraint`: This line imports the necessary classes from the `constraint` module. `Problem` is used to create a new constraint satisfaction problem, and `NotEqualsConstraint` is a type of constraint that ensures two variables do not have the same value.
2. `def map_coloring():`: This line defines a function called `map_coloring` that will contain our problem.
3. `problem = Problem()`: This line creates a new `Problem` instance which we will use to define our problem.
4. `regions = ['A', 'B', 'C', 'D']`: This line defines a list of regions that need to be colored. Each region is represented by a letter.
5. `problem.addVariables(regions, ['red', 'green', 'blue'])`: This line adds our regions as variables to our problem. The second argument is a list of possible colors that each region can take.
6. The next five lines add constraints to our problem. Each constraint is a `NotEqualsConstraint` between two regions, meaning two regions cannot have the same color. For example, `problem.addConstraint(NotEqualsConstraint(), ['A', 'B'])` means region 'A' cannot have the same color as region 'B'.
7. `solutions = problem.getSolutions()`: This line solves the problem and gets a list of all possible solutions.
8. The last two lines of the function print out each solution.
9. Finally, `map_coloring()` calls our function and starts the program.

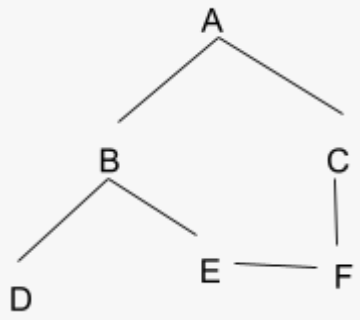
---

4. BFS & DFS

```
# Graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
```

```
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# DFS Implementation
def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for n in graph[node]:
            dfs(graph, n, visited)
    return visited

visited = dfs(graph, 'A', [])
print("DFS:", visited)

# BFS Implementation
def bfs(graph, node):
    visited = []
    queue = [node]

    while queue:
        n = queue.pop(0)
        if n not in visited:
            visited.append(n)
            queue.extend(graph[n])
    return visited

visited = bfs(graph, 'A')
print("BFS:", visited)
```

Here’s a breakdown of the code:



1. **Graph Representation**: The graph is represented as an adjacency list, which is a dictionary where each key is a node in the graph, and the value is a list of nodes that the key node is connected to.
2. **DFS Implementation**: The DFS function is a recursive function that visits a node and all its unvisited adjacent nodes. It keeps track of visited nodes in a list.
3. **BFS Implementation**: The BFS function uses a queue to visit a node's adjacent nodes before moving on to the next level of nodes. It also keeps track of visited nodes in a list.

---

5. A* Algorithm

```
import heapq

def astar(graph, start, goal):
    frontier = [(0, start)]  # Priority queue: (f-score, node)
    came_from = {}
    cost_so_far = {start: 0}

    while frontier:
        _, current = heapq.heappop(frontier)

        if current == goal:
            break

        for next_node in graph[current]:
            new_cost = cost_so_far[current] +
graph[current][next_node]
            if next_node not in cost_so_far or new_cost <
cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                priority = new_cost + heuristic(next_node, goal)
                heapq.heappush(frontier, (priority, next_node))
                came_from[next_node] = current

    path = []
    while current != start:
        path.append(current)
        current = came_from[current]
```

```
    path.append(start)
    path.reverse()

    return path

def heuristic(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

if __name__ == "__main__":
    graph = {
        (0, 0): {(0, 1): 1, (1, 0): 1},
        (0, 1): {(0, 0): 1, (0, 2): 1},
        (0, 2): {(0, 1): 1, (1, 2): 1},
        (1, 0): {(0, 0): 1, (1, 1): 1},
        (1, 1): {(1, 0): 1, (1, 2): 1},
        (1, 2): {(0, 2): 1, (1, 1): 1}
    }
    start = (0, 0)
    goal = (1, 2)
    path = astar(graph, start, goal)
    print("Path found by A* Search:")
    print(path)
```

Sure, this code is an implementation of the A* search algorithm, which is a pathfinding algorithm often used in computer science and game development. Here's a breakdown of the code:

1. **Importing heapq**: `heapq` is a Python module which provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.
2. *Defining the A function**: The `astar` function is defined with three parameters: `graph`, `start`, and `goal`. `graph` is a dictionary representing the graph on which we're finding a path. `start` and `goal` are the nodes where the path starts and ends, respectively.
3. **Initializing variables**: `frontier` is a list of tuples, each containing a node and its corresponding f-score (the sum of the cost to reach the node and the heuristic cost from the node to the goal). `came_from` is a dictionary that will be used to keep track of the path. `cost_so_far` is a dictionary that keeps track of the cost to reach each node from the start node.

4. **Main loop**: The algorithm continues until there are no more nodes to explore (i.e., the frontier is empty) or until the goal has been reached.
5. **Exploring neighbors**: For the current node, the algorithm calculates the cost to reach each of its neighbors. If a neighbor has not been visited yet, or if the new cost to reach it is less than the previously calculated cost, the neighbor is added to the frontier with its updated priority score (f-score).
6. **Building the path**: Once the goal is reached, the algorithm builds the path from the start to the goal by following the `came_from` links from the goal to the start.
7. **Heuristic function**: The `heuristic` function calculates the estimated cost to reach the goal from a given node. In this case, it uses the Manhattan distance (sum of the absolute values of the differences in the goal's coordinates and the current node's coordinates).
8. **Main execution**: If the script is run (not imported), a graph is defined, the A* function is called with this graph, a start node, and a goal node, and the resulting path is printed out.

The A* algorithm is efficient and finds the shortest path if one exists. However, its performance heavily depends on the heuristic used. In this case, the Manhattan distance is used, which works well for grid-based maps where movement is allowed only in four directions. If movement were allowed in eight directions (including diagonals), a different heuristic, such as the Euclidean distance, would be more appropriate.

---

6. Best First Search

```python
import heapq

def best_first_search(graph, start, goal):
    queue = [(0, start)]
    visited = set()

    while queue:
        priority, current_node = heapq.heappop(queue)

        if current_node == goal:
            return True
```

```
        if current_node not in visited:
            visited.add(current_node)

            for neighbor in graph[current_node]:
                if neighbor not in visited:
                    heapq.heappush(queue,
(graph[current_node][neighbor], neighbor))

    return False

# example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 5},
    'C': {'A': 4, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

print(best_first_search(graph, 'A', 'D'))  # returns True
```

This Python code implements the Best-First Search algorithm, which is a search algorithm used to traverse or search through a graph. It's often used in pathfinding and graph traversal problems, the classic example being finding the shortest path from one point to another.

Here's a breakdown of the code:

- **Importing the heapq module:** heapq is a Python module which provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

```
import heapq
```

- **Defining the best_first_search function:** This function takes a graph, a start node, and a goal node as arguments. The graph is represented as a dictionary where the keys are the nodes and the values are dictionaries of the neighbors and their respective costs.

```
def best_first_search(graph, start, goal):
```

- **Initializing the queue and visited set:** The queue is implemented as a list of tuples, where each tuple contains a node and its priority. The visited set is used to keep track of the nodes that have already been visited.

```
queue = [(0, start)]
```

```
visited = set()
```

- **Main loop:** The algorithm continues until there are no more nodes left in the queue.

```
while queue:
```

- **Choosing the next node:** The node with the lowest cost (highest priority) is chosen and removed from the queue.

```
priority, current_node = heapq.heappop(queue)
```

- **Checking if the goal has been reached:** If the current node is the goal, the function returns True.

```
if current_node == goal:
    return True
```

- **Exploring the neighbors:** If the current node has not been visited before, it's added to the visited set. Then, for each of its neighbors, if the neighbor has not been visited before, it's added to the queue with its cost as the priority.

```
if current_node not in visited:
    visited.add(current_node)

    for neighbor in graph[current_node]:
        if neighbor not in visited:
            heapq.heappush(queue, (graph[current_node][neighbor], neighbor))
```

- **Returning False if the goal is not reachable:** If the function hasn't returned True after exploring all nodes, it means the goal is not reachable from the start node, so it returns False.

```
return False
```

- **Example usage:** The function is then used with an example graph and start and goal nodes.

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 5},
```

```
    'C': {'A': 4, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

print(best_first_search(graph, 'A', 'D'))  # returns True
```

---

7. Monty Hall Problem

```python
import random

def monty_hall(simulations=1000):
    switch_wins = 0
    stay_wins = 0

    for _ in range(simulations):
        doors = [0, 0, 1]  # 0 represents a goat, 1 represents a car
        random.shuffle(doors)  # The prizes behind the doors are randomized

        # The player makes a choice
        choice = random.randint(0, 2)

        # Monty opens a door
        monty_opens = [i for i in range(3) if doors[i] == 0 and i != choice][0]

        # The player switches their choice
        switch_choice = [i for i in range(3) if i != choice and i != monty_opens][0]

        # Check if the player would win by switching or staying
        if doors[choice] == 1:
            stay_wins += 1
        if doors[switch_choice] == 1:
            switch_wins += 1
```

```
            print(f"Winning   by   staying   with   original   door:
{stay_wins/simulations}")
                    print(f"Winning     by     switching     doors:
{switch_wins/simulations}")

monty_hall()
```

Sure, let's break down the Monty Hall problem simulation code line by line:

1. `import random`: This line imports the `random` module, which contains functions for generating random numbers.
2. `def monty_hall(simulations=1000):`: This line defines a function called `monty_hall` that takes one argument, `simulations`, which defaults to 1000 if no value is provided.
3. `switch_wins = 0` and `stay_wins = 0`: These lines initialize two counters to keep track of the number of wins when the player switches doors and when the player stays with their initial choice.
4. `for _ in range(simulations):`: This line starts a loop that will run for the number of times specified by `simulations`.
5. `doors = [0, 0, 1]`: This line creates a list representing the three doors. A `0` represents a goat and a `1` represents a car.
6. `random.shuffle(doors)`: This line randomly shuffles the `doors` list to simulate placing the prizes behind the doors randomly.
7. `choice = random.randint(0, 2)`: This line simulates the player's initial choice of door.
8. `monty_opens = [i for i in range(3) if doors[i] == 0 and i != choice][0]`: This line determines which door Monty Hall opens. Monty always opens a door that the player did not pick and that has a goat behind it.
9. `switch_choice = [i for i in range(3) if i != choice and i != monty_opens][0]`: This line determines the door the player would switch to if they decide to switch. It's the remaining unopened door.
10. `if doors[choice] == 1:` and `if doors[switch_choice] == 1:`: These lines check if the player's final choice (either stay or switch) is the door with the car. If it is, the corresponding win counter is incremented.
11. `print(f"Winning by staying with original door: {stay_wins/simulations}")` and `print(f"Winning by switching doors: {switch_wins/simulations}")`: These lines print the probabilities of winning for both strategies.
12. `monty_hall()`: This line calls the `monty_hall` function to run the simulation.

---

8. Unification & Resolution

```python
def unify_var(var, x, s):
    if var in s:
        return unify(s[var], x, s)
    else:
        return extend(s, var, x)

def unify(x, y, s):
    if s is None:
        return None
    elif x == y:
        return s
    elif is_var(x):
        return unify_var(x, y, s)
    elif is_var(y):
        return unify_var(y, x, s)
    else:
        return None

def is_var(x):
    return isinstance(x, str) and x.startswith('?')

def extend(s, var, val):
    s2 = s.copy()
    s2[var] = val
    return s2
s = {}
x = '?x'
y = 'John'
result = unify(x, y, s)
print(result)  # Output: {'?x': 'John'}
```

—----------

9. Unsupervised Learning

```
# Import required libraries
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
np.random.seed(0)
points = np.random.randn(100, 2)

# Create a KMeans instance with 2 clusters (we choose 2 arbitrarily)
kmeans = KMeans(n_clusters=2)

# Fit the model to the data
kmeans.fit(points)

# Get the cluster assignments for each data point
labels = kmeans.labels_

# Plot the data points colored by their cluster assignments
plt.scatter(points[:, 0], points[:, 1], c=labels)
plt.show()
```

Sure, I'd be happy to explain the code line by line:

# Import required libraries
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

These lines import the necessary libraries for the code. `KMeans` is a clustering algorithm from the `sklearn` library, `numpy` is used for numerical operations, and `matplotlib.pyplot` is used for plotting the data.

Generate random data

```
np.random.seed(0)
points = np.random.randn(100, 2)
```

Here, we're generating a dataset of 100 points, each with two features (so it's 2-dimensional and can be easily visualized). The data is randomly generated from a normal distribution. The `np.random.seed(0)` line ensures that the random numbers generated are the same every time the code is run.

```
# Create a KMeans instance with 2 clusters (we choose 2 arbitrarily)
kmeans = KMeans(n_clusters=2)
```

This line creates an instance of the KMeans algorithm. We're setting it to create 2 clusters, but this number can be changed depending on the specific problem.

```
# Fit the model to the data
kmeans.fit(points)
```

This line fits the KMeans model to our data. This is where the actual clustering happens.

```
# Get the cluster assignments for each data point
labels = kmeans.labels_
```

After fitting the model, we can get the cluster assignments for each data point. These are stored in `kmeans.labels_`.

```
# Plot the data points colored by their cluster assignments
plt.scatter(points[:, 0], points[:, 1], c=labels)
plt.show()
```

Finally, we plot the data points, coloring them by their cluster assignments. `points[:, 0]` and `points[:, 1]` are the x and y coordinates of the points, and `c=labels` sets the color of each point according to its cluster assignment.

—----------

10. Natural Language Processing

```
import nltk
import string
corpus="Any text can be entered"

lowerc=corpus.lower()
```

```
print(lowerc)
puncrem="".join(c for c in lowerc if c not in string.punctuation)
print(puncrem)
tokend=nltk.word_tokenize(puncrem)
print(tokend)

from nltk.corpus import stopwords
stp_wrds=set(stopwords.words("english"))
filter_tokend=[i for i in tokend if i not in stp_wrds]
print(filter_tokend)

from nltk.stem import PorterStemmer
stemmer=PorterStemmer()
stemmed_tokens=[stemmer.stem(token) for token in filter_tokend]
print(stemmed_tokens)

from nltk.tag import pos_tag
pos_tag(stemmed_tokens)
```

Sure, I'd be happy to explain the code line by line:

```
import nltk
import string
```

These lines import the necessary libraries for the code. `nltk` is the Natural Language Toolkit, a library that provides tools for Natural Language Processing (NLP). `string` is a built-in Python module for common string operations.

```
corpus="Any text can be entered"
```

This line defines a string variable `corpus` which contains the text that will be processed.

```
lowerc=corpus.lower()
print(lowerc)
```

This converts all the text in `corpus` to lowercase and prints it. This is often done in NLP to ensure that the same word in different cases is not considered as two different words.

```
puncrem="".join(c for c in lowerc if c not in string.punctuation)
print(puncrem)
```

This removes all punctuation from the text and prints it. This is done because punctuation usually doesn't carry meaning and can interfere with the processing of the text.

```
tokend=nltk.word_tokenize(puncrem)
print(tokend)
```

This tokenizes the text, i.e., splits it into individual words, and prints the tokens. This is one of the most basic steps in NLP.

```
from nltk.corpus import stopwords
stp_wrds=set(stopwords.words("english"))
filter_tokend=[i for i in tokend if i not in stp_wrds]
print(filter_tokend)
```

This removes stop words (common words like 'is', 'the', 'and', etc. that do not contain important meaning) from the tokens and prints the filtered tokens.

```
from nltk.stem import PorterStemmer
stemmer=PorterStemmer()
stemmed_tokens=[stemmer.stem(token) for token in filter_tokend]
print(stemmed_tokens)
```

This performs stemming, which is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. The stemmed tokens are then printed.

```
from nltk.tag import pos_tag
pos_tag(stemmed_tokens)
```

This performs part-of-speech tagging, which is the process of marking each word in the text as corresponding to a particular part of speech (e.g., noun, verb, adjective, etc.), based on both its definition and its context.

---

11. Deep Learning

```
import tensorflow as tf
```

```
from tensorflow.keras import layers, models, datasets
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32,
3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10, validation_data=(x_test,
y_test))

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print("Test accuracy:", test_acc)

num_images = 5
plt.figure(figsize=(10, 10))
for i in range(num_images):
    plt.subplot(1, num_images, i + 1)
    plt.imshow(x_test[i])
    plt.title(class_names[np.argmax(model.predict(x_test[i:i+1]))])
    plt.axis('off')
plt.show()
```

Sure, I'd be happy to explain the code line by line:

import tensorflow as tf
from tensorflow.keras import layers, models, datasets

These lines import the necessary libraries for the code. `tensorflow` is the main library used for creating and training neural networks. `layers`, `models`, and `datasets` are modules within `tensorflow.keras` that provide tools for building neural network layers, models, and loading datasets, respectively.

```
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
```

This line loads the CIFAR-10 dataset, which is a set of 60,000 32x32 color images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

This line normalizes the image data to be between 0 and 1. Image data is typically composed of pixel values that range from 0 to 255. Dividing by 255 scales this data to the range 0-1.

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])
```

This block of code defines the architecture of the neural network model. It's a Sequential model, which means it's a linear stack of layers. The model consists of three convolutional layers (Conv2D), each followed by a max pooling layer (MaxPooling2D), a flattening layer (Flatten), and two dense (fully connected) layers (Dense).

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

This line compiles the model with the Adam optimizer, the sparse categorical crossentropy loss function, and accuracy as the metric to monitor.

```
model.fit(x_train, y_train, epochs=50, validation_data=(x_test, y_test))
```

This line trains the model for 50 epochs on the training data and evaluates it on the test data after each epoch.

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print("Test accuracy:", test_acc)
```

These lines evaluate the model on the test data and print the test accuracy.

```
num_images = 5
plt.figure(figsize=(10, 10))
for i in range(num_images):
    plt.subplot(1, num_images, i + 1)
    plt.imshow(x_test[i])
    plt.title(class_names[np.argmax(model.predict(x_test[i:i+1]))])
    plt.axis('off')
plt.show()
```

These lines plot the first 5 images from the test set, along with the model's predicted class for each image. Note that `class_names` is not defined in the provided code, so you would need to define it before running this part of the code. For the CIFAR-10 dataset, `class_names` would be defined as follows:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```