```python
# (zero-shot) Instruction: Turn on a faucet by rotating its handle. The task
is finished when qpos of faucet handle is larger than target qpos.
import numpy as np
from scipy.spatial.distance import cdist

def compute_dense_reward(self, action):
  # Here I define the reward and penalty weights for different aspects of the task
  handle_reach_weight = 0.1
  grasp_handle_weight = 0.2
  rotation_weight = 0.7
  action_penalty_weight = 0.01

  # Here I define the grasp success reward
  grasp_success_reward = 0.1

  # Here I compute the distance between the robot's end effector and the faucet handle
  handle_pcd = self.faucet.handle.get_world_pcd()
  lfinger_cords = np.array([self.robot.lfinger.pose.p, ])
  rfinger_cords = np.array([self.robot.rfinger.pose.p, ])
  dist_lfinger_handle = cdist(lfinger_cords, handle_pcd).min(-1)[0]
  dist_rfinger_handle = cdist(rfinger_cords, handle_pcd).min(-1)[0]
  dist_handle_reach = max(dist_lfinger_handle, dist_rfinger_handle)
  handle_reach_reward = - handle_reach_weight * dist_handle_reach

  # Here I check if the robot has successfully grasped the faucet handle
  grasp_handle = self.robot.check_grasp(self.faucet.handle)
  grasp_handle_reward = grasp_handle_weight * grasp_handle if grasp_handle else 0

  # Here I calculate the rotation reward based on the difference between
  # the current and target joint position
  rotation_diff = max(0, self.faucet.handle.target_qpos - self.faucet.handle.qpos)
  rotation_reward = - rotation_weight * rotation_diff

  # Here I calculate the penalty for the robot's action
  action_penalty = - action_penalty_weight*np.linalg.norm(action)

  # The total reward is the sum of all the individual rewards and penalties
  reward = handle_reach_reward + grasp_handle_reward + rotation_reward + action_penalty

  if grasp_handle and rotation_diff == 0:
    reward += grasp_success_reward

  return reward
```

```python
# (few-shot) Instruction: A dual-arm mobile robot needs to push a swivel chair
to a target location on the ground and prevent it from falling over.
import numpy as np
import scipy.spatial.distance as sdist

def compute_dense_reward(self, action):
  reward = -20.0

  actor = self.chair
  ee_coords = np.array(self.robot.get_ee_coords())
  ee_mids = np.array([ee_coords[:2].mean(0), ee_coords[2:].mean(0)])
  chair_pcd = self.chair.get_pcd()

  # EE approach chair
  dist_ees_to_chair = sdist.cdist(ee_coords, chair_pcd) # [4, N]
  dist_ees_to_chair = dist_ees_to_chair.min(1) # [4]
  dist_ee_to_chair = dist_ees_to_chair.mean()
  log_dist_ee_to_chair = np.log(dist_ee_to_chair + 1e-5)
  reward += - dist_ee_to_chair - np.clip(log_dist_ee_to_chair, -10, 0)

  # Penalize action
  action_norm = np.linalg.norm(action)
  reward -= action_norm * 1e-6

  # Keep chair standing
  z_axis_world = np.array([0, 0, 1])
  z_axis_chair = self.chair.pose.to_transformation_matrix()[:3, 2]
  chair_tilt = np.arccos(z_axis_chair[2])
  log_chair_tilt = np.log(chair_tilt + 1e-5)
  reward += -chair_tilt * 0.2

  # Chair velocity
  chair_vel = actor.velocity
  chair_vel_norm = np.linalg.norm(chair_vel)
  disp_chair_to_target = self.chair.pose.p[:2] - self.target_xy
  chair_vel_dir = sdist.cosine(chair_vel[:2], disp_chair_to_target)
  chair_ang_vel_norm = np.linalg.norm(actor.angular_velocity)

  # Stage reward
  stage_reward = 0

  dist_chair_to_target = np.linalg.norm(self.chair.pose.p[:2]-self.target_xy)

  if dist_ee_to_chair < 0.2:
    stage_reward += 2
    if dist_chair_to_target <= 0.3:
      stage_reward += 2
      reward += (np.exp(-chair_vel_norm * 10) * 2)
      if chair_vel_norm <= 0.1 and chair_ang_vel_norm <= 0.2:
        stage_reward += 2
        if chair_tilt <= 0.1 * np.pi:
          stage_reward += 2
    else:
      reward_vel = (chair_vel_dir - 1) * chair_vel_norm
      reward += np.clip(1-np.exp(-reward_vel),-1,np.inf)*2-dist_chair_to_target*2

  if chair_tilt > 0.4 * np.pi:
    stage_reward -= 2

  reward = reward + stage_reward
  return reward
```

```python
# (zero-shot) Instruction: Pick up cube A and move it to the 3D goal position.
import numpy as np

def compute_dense_reward(self, action):
    # Normalize action
    action = np.clip(action, -1, 1)

    # Calculate distance between gripper and cube
    gripper_pos = self.robot.ee_pose.p
    cube_pos = self.cubeA.pose.p
    dist_gripper_cube = np.linalg.norm(gripper_pos - cube_pos)

    # Calculate distance between cube and goal
    goal_pos = self.goal_position
    dist_cube_goal = np.linalg.norm(goal_pos - cube_pos)

    # Check if the robot is grasping the cube
    grasping_cube = self.robot.check_grasp(self.cubeA)

    # Define reward components
    reward_dist_gripper_cube = -1.0 * dist_gripper_cube
    reward_dist_cube_goal = -1.0 * dist_cube_goal
    reward_grasping_cube = 1.0 if grasping_cube else -1.0

    # Define weights for reward components
    weight_dist_gripper_cube = 0.3
    weight_dist_cube_goal = 0.5
    weight_grasping_cube = 0.2

    # Calculate total reward
    reward = weight_dist_gripper_cube * reward_dist_gripper_cube \
            + weight_dist_cube_goal * reward_dist_cube_goal \
            + weight_grasping_cube * reward_grasping_cube

    # Regularization on action
    reward -= 0.01 * (action ** 2).sum()

    return reward
```

```python
# (few-shot) Instruction: Pick up cube A and move it to the 3D goal position.
import numpy as np

def compute_dense_reward(self, action):
    reward = 0.0

    # Check if the cube is placed at the goal position and the robot is static
    cube_at_goal = np.linalg.norm(self.cubeA.pose.p - self.goal_position) \
                                                    <= self.cube_half_size
    is_robot_static = np.max(np.abs(self.robot.qvel)) <= 0.2

    # If the cube is placed at the goal and the robot is static, return a high reward
    if cube_at_goal and is_robot_static:
        reward += 2.25
        return reward

    # reaching reward, encourages the robot to reach the cube
    gripper_pos = self.robot.ee_pose.p
    obj_pos = self.cubeA.pose.p
    dist_to_obj = np.linalg.norm(gripper_pos - obj_pos)
    reaching_reward = 1 - np.tanh(5 * dist_to_obj)
    reward += reaching_reward

    # grasp reward, encourages the robot to grasp the cube
    is_grasped = self.robot.check_grasp(self.cubeA, max_angle=30)
    if is_grasped:
        reward += 0.25

    # placement reward, encourages the robot to place the cube at the goal
    if is_grasped:
        dist_to_goal = np.linalg.norm(self.cubeA.pose.p - self.goal_position)
        placement_reward = 1 - np.tanh(5 * dist_to_goal)
        reward += placement_reward

    # regularization term on robot's action
    action_reg = -np.sum(np.square(action)) / len(action)
    reward += 0.1 * action_reg

    return reward
```

```
# (few-shot interactive) Instruction: Pick up cubeA and place it on cubeB. The task is finished
when cubeA is on top of cubeB stably (i.e. cubeA is static) and isn't grasped by the gripper.
import numpy as np

def compute_dense_reward(self, action):
  reward = 0.0
  # Check if cube A is on top of cube B and whether it is stable
  is_obj_on_target = np.linalg.norm(self.cubeA.pose.p - (self.cubeB.pose.p \
                      + np.array([0, 0, self.cube_half_size*2]))) <= 0.025
  is_obj_static = self.cubeA.check_static()
  is_grasped = self.robot.check_grasp(self.cubeA)
+ gripper_openness = self.robot.gripper_openness

- success = is_obj_on_target and is_obj_static and not is_grasped
+ success = is_obj_on_target and is_obj_static

  if success:
    reward += 5
-   return reward
+   if not is_grasped:
+     reward += 5
+     return reward

  # Compute distance from the robot's end-effector to cube A
  tcp_to_obj_pos = self.cubeA.pose.p - self.robot.ee_pose.p
  tcp_to_obj_dist = np.linalg.norm(tcp_to_obj_pos)
  reaching_reward = 1 - np.tanh(5 * tcp_to_obj_dist)
  reward += reaching_reward

  # Check if the robot has grasped cube A
  reward += 1 if is_grasped else 0.0
  if is_grasped:
    # Compute distance from cube A to the top of cube B
    obj_to_goal_pos = self.cubeB.pose.p + np.array([0, 0, self.cube_half_size*2]) \
                                                            - self.cubeA.pose.p
    obj_to_goal_dist = np.linalg.norm(obj_to_goal_pos)
    place_reward = 1 - np.tanh(5 * obj_to_goal_dist)
    reward += place_reward

+   if is_obj_on_target and is_obj_static:
+     # Encourage robot to release grasp when cube A is on top of cube B and stable
+     release_reward = 3 if not is_grasped and gripper_openness > 0.5 else -3
+     reward += release_reward
+   else:
+     # Encourage the robot to keep the grasp if it's not on target
+     holding_reward = 1 if is_grasped else -1
+     reward += holding_reward
+ else:
+   if not is_obj_on_target:
+     # Encourage the robot to move closer to the target when not grasping
+     tcp_to_target_pos = self.cubeB.pose.p + np.array([0, 0, self.cube_half_size*2]) \
+                                                          - self.robot.ee_pose.p
+     tcp_to_target_dist = np.linalg.norm(tcp_to_target_pos)
+     target_reward = 1 - np.tanh(5 * tcp_to_target_dist)
+     reward += target_reward

  return reward
```