Exercise 1

Here is the step-by-step explanation of the implementation, linking the Python code to the mathematical theory of Gradient Descent.

1. Defining the Optimization Problem

First, we define the mathematical landscape we want to navigate. In Machine Learning, this corresponds to the Loss Function $\mathcal{L}(\Theta)$, which measures the error of our model.

```
def loss_fn_1(theta):
    return (theta - 3)**2 + 1

def gradient_fn_1(theta):
    return 2 * (theta - 3)
```

- **Function Parameters & Types:**
- Both functions accept `theta`. In the context of the loop, `theta` is a **scalar (float)** representing the current parameter value. However, thanks to NumPy's polymorphism, `loss_fn_1` can also accept a **NumPy array**, which allows for element-wise calculation later.
- **Code & Theory Connection:**
- `loss_fn_1` implements the objective function $\mathcal{L}(\theta) = (\theta - 3)^2 + 1$. This is a convex function with a global minimum at $\theta = 3$.
- `gradient_fn_1` implements the gradient $\nabla\mathcal{L}(\theta)$. Analytically, the derivative is $2(\theta - 3)$. The gradient points in the direction of the steepest ascent. To find the minimum, the algorithm must move in the opposite direction of this value.

---

2. The Gradient Descent Algorithm

This is the core engine. The function `GD` encapsulates the iterative logic required to minimize the loss function.

```
def GD(loss_function, gradient_function, start_theta, learning_rate=0.01, n_iterations=100):
    theta = start_theta
    theta_history = [theta]
```

- **Parameters & Types:**
- `loss_function`, `gradient_function`: **Callables** (Python functions). This makes the code modular; you can pass any differentiable function here.
- `start_theta`: **Float**. The initial guess $\Theta^{(0)}$.

- `learning_rate`: **Float**. Represents η (eta), the step size.
- `n_iterations`: **Int**. The stopping criterion, defined as the maximum number of steps (`maxit`).
- **Initialization:** We initialize `theta` at the starting point and create a list `theta_history` to store the trajectory.

---

3. The Iterative Update Loop

We now enter the loop that performs the actual optimization steps.

```
for k in range(n_iterations):
    grad = gradient_function(theta)

    theta = theta - learning_rate * grad
    theta_history.append(theta)
```

Code & Theory Connection:

- **Gradient Calculation:** `grad = gradient_function(theta)` computes $\nabla\mathcal{L}(\Theta^{(k)})$.
- **The Update Rule:** The line `theta = theta - learning_rate * grad` is the direct translation of the Gradient Descent update formula found in the theory:

$$\Theta^{(k+1)} = \Theta^{(k)} - \eta\nabla\mathcal{L}(\Theta^{(k)})$$

- By subtracting the gradient (scaled by η), we move θ towards the minimum. If the learning rate is too small, convergence is slow; if too large, it may oscillate or diverge.
- By subtracting the gradient (scaled by), we move towards the minimum. If the learning rate is too small, convergence is slow; if too large, it may oscillate or diverge.

---

4. Vectorized Loss Calculation

After the loop finishes, we perform a crucial optimization for efficiency and analysis.

```
theta_history = np.array(theta_history)
loss_history = loss_function(theta_history)
return theta_history, loss_history
```

- **Code Explanation:**

- Instead of calculating the loss value inside the loop (which would require calling the function N times scalar-wise), we convert the list of positions into a NumPy array.
- We then pass this entire array to loss_function. Thanks to NumPy's broadcasting, the mathematical operation $(\theta - 3)^2 + 1$ is applied to every element in the vector simultaneously.
- **Return Types:** The function returns two **NumPy arrays** containing the sequence of parameters and their corresponding loss values.

---

5. Execution and Hyperparameter Selection

Finally, we run the algorithm using different learning rates to observe how step size affects convergence.

```
start_theta = 0.0
n_iter = 50
etas = [0.05, 0.2, 1.0]

results = {}

for eta in etas:
    thetas, losses = GD(loss_fn_1, gradient_fn_1, start_theta, eta, n_iter)
    results[eta] = {
        'thetas': thetas,
        'losses': losses
    }
```

- **Code & Theory Connection:**
- We define a starting point $\Theta^{(0)} = 0.0$. Since the function is convex, the choice of $\Theta^{(0)}$ is less critical as it will ideally converge to the global minimum regardless.
- We iterate through different values of η (`etas`). This allows us to empirically verify the theoretical behavior of step sizes:
- Small η (0.05) leads to slow, steady convergence.
- Moderate η (0.2) leads to efficient convergence.
- Large η (1.0) might cause the parameter to bounce back and forth around the minimum.
- The results are stored in a dictionary to facilitate the comparison of trajectories and loss reduction over time.