BAN401: Applied Programming and Data Analysis for Business

Candidate numbers: 10, 53, 92, 149

Handed out: September 31, 2024
Deadline: November 11, 2024

Contents

1 Problem 1

1.1 Python code

```python
#function for password checker
def password_checker():
    # Variabel for the correct password
    correct_password = "friend"
    # Variabel for the number of attempts
    count = 5

    # Loop for the number of attempts
    while count > 0:

        # Input for the password
        password = input("Enter your password to access the client database: ")
        # Decrement the number of attempts
        count -= 1

        # Check if the password is correct
        if password == correct_password:
            # If the password is correct, print a message and exit the loop
            print("Access granted. Welcome, friend!")
            return

        # Partial match calculation
        matches = sum(1 for i in range(min(len(password), len(correct_password)
            )) if password[i] == correct_password[i])
        # Total length calculation
        total_length = len(correct_password)

        # Print the result
        if matches > total_length / 2:
            # Calculate the percentage of partial matches
            partial_match_percentage = (matches / total_length) * 100
            # Print the result
            print(f"Partial match: {partial_match_percentage:.1f}% Attempts
                remaining: {count}")
        else:
            # Print the result
            print(f"Incorrect password. Attempts remaining: {count}")

    # If the number of attempts is exceeded, print an error message
    print("Incorrect password supplied 5 (five) times. Access denied.")

# Run the password checker
password_checker()
```

1.2 Explanation of our solution

This password checker limits access to five attempts, checking if the input matches the correct password ("friend"). If it doesn't match, the code hints at partial matches to help the user identify if they're close to guessing the correct password. The code sets up a loop for five attempts, reducing the count with each try. If the password matches, access is granted, and the loop ends. If not, it calculates matching characters at the same positions in the correct password. If over half match, it displays the percentage match. After five failed attempts, it denies access, ensuring limited tries while offering guidance if partially correct.

1.3 Screenshost of the results

```
Enter your password to access the client database: fiend
Incorrect password. Attempts remaining: 4
Enter your password to access the client database: frend
Incorrect password. Attempts remaining: 3
Enter your password to access the client database: frren
Partial match: 66.7% Attempts remaining: 2
Enter your password to access the client database: friend
Access granted. Welcome, friend!
```

Figure 1: Password_case

2 Problem 2

2.1 Python code

```
#Validate all input data:
age = int(input("What is your age? "))
income = input("What is your income? ")
income = int(income.replace(",",""))
credit_score = int(input("What is your credit score? "))
debt = input("What is your current amount of existing debt? ")
debt = int(debt.replace(",", ""))
employment_status = input("What is your employment status? (employed/unemployed) ").lower()
loan = input("What loan amount are you requesting? ")
loan = int(loan.replace(",",  ""))
term = int(input("Over how many years would you like to repay the loan?  "))

# Check criteria for loan approval
if 18 <= age <= 70:
    if income > 65000:
        if credit_score > 700:
            debt_ratio = (debt / income) * 100
            if debt_ratio < 40:
                if employment_status == "employed":
                    half_income = income / 2
                    if loan < half_income:
                        if 1 <= term <= 30:
                            print("\n")
                            print(f"You meet all the criteria for a loan, and your request for ${loan} for a term of {term} years has been approved")
                            # If-statement for which interest rate
                            if (700 <= credit_score <= 750) and (term <= 15):
                                print("Your interest rate is 5%")
                            elif (700 <= credit_score <= 750) and (term > 15):
                                print("Your interest rate is 6%")
                            elif (751 <= credit_score <= 800) and (term <= 15):
                                print("Your interest rate is 4%")
                            elif (751 <= credit_score <= 800) and (term > 15):
                                print("Your interest rate is 5%")
                            elif (credit_score > 800) and (term <= 15):
                                print("Your interest rate is 3%")
                            elif (credit_score > 800) and (term > 15):
                                print("Your interest rate is 4%")
                        else:
                            print("Your requested loan term is not within the acceptable range for approval.")
                    else:
                        print("Your requested loan amount is too high relative to your income, making you ineligible to request a loan.")
                else:
                    print("You must be currently employed to request a loan.")
            else:
                print("Your debt-to-income ratio is too high, and you do not qualify to request a loan.")
        else:
            print("You do not meet the credit score requirement to request a loan. ")
    else:
        print("You do not meet the income requirements to request a loan")
else:
    print("You do not meet the age requirements to request a loan")
```

2.2 Explanation of our solution

The code evaluates whether an applicant meets the requirements for a loan by checking several key criteria. It first ensures the applicant's age is within an acceptable range and that their income and credit score meet minimum thresholds. Then, it calculates the debt-to-income ratio to confirm the applicant's financial stability for taking on new debt. The code also verifies employment status and checks that the requested loan amount and repayment term are reasonable in relation to the applicant's income. If all conditions are met, the loan is approved with an interest rate adjusted for the applicant's credit score and loan term.

2.3 Screenshots of the results

```
Loan Approval
What is your age? 24
What is your income? 120,000
What is your credit score? 790
What is your current amount of existing debt? 20,000
What is your employment status? (employed/unemployed) employed
What loan amount are you requesting? 40,000
Over how many years would you like to repay the loan?  10


You meet all the criteria for a loan, and your request for $40000 for a term of 10 years has been approved
Your interest rate is 4%
```

Figure 2: Scenario A

```
Loan Approval
What is your age? 35
What is your income? 45,000
What is your credit score? 680
What is your current amount of existing debt? 25,000
What is your employment status? (employed/unemployed) unemployed
What loan amount are you requesting? 30,000
Over how many years would you like to repay the loan?  20
You do not meet the income requirements to request a loan
```

Figure 3: Scenario B

```
Loan Approval
What is your age? 40
What is your income? 75,000
What is your credit score? 810
What is your current amount of existing debt? 40,000
What is your employment status? (employed/unemployed) employed
What loan amount are you requesting? 45,000
Over how many years would you like to repay the loan?  25
Your debt-to-income ratio is too high, and you do not qualify to request a loan.
```

Figure 4: Scenario C

3 Problem 3

3.1 Asking customers for feedback

3.1.1 Python code

```
1
2  # Define the feedback_for_loop function
3  def feedback_for_loop():
4      # Collect feedback from customers using a for loop
5      num_customers = int(input("How many customers do you want to collect
           feedback from? "))
6      # Initialize an empty list to store the feedback
7      feedback_list = []
8
9      # Use a for loop to collect feedback
10     for i in range(num_customers):
11         # Get feedback from the customer
12         feedback = input(f"Enter feedback for customer {i+1}: ")
13         # Add the feedback to the list
14         feedback_list.append(feedback)
15
16     # Print the collected feedback
17     print("\nCollected Feedback:")
18     # Use an enumerate function to get the customer number
19     for i, feedback in enumerate(feedback_list, 1):
20       # Print the feedback for each customer
21         print(f"Customer {i}: {feedback}")
22
23 # Run the program
24 feedback_for_loop()
25
26 \subsection{Screenshots of the results}
```

3.1.2 Screenshots of the results

```
How many customers do you want to collect feedback from? 1
Enter feedback for customer 1: good

Collected Feedback:
Customer 1: good
```

Figure 5: Feedback_for_loop

3.1.3 Explanation of our solution

The solution is designed to streamline customer feedback collection in a user-friendly manner by employing a straightforward for loop structure. This code enables the collection, storage, and display of customer feedback, making it a suitable approach for scenarios requiring repetitive input gathering.

3.2 Entering feedback into the system

3.2.1 Python code

```
1
2  # Define the feedback_while_loop function
3  def feedback_while_loop():
4    # Collect feedback from customers using a while loop
5      feedback_list = []
6      # Use a while loop to collect feedback
7      while True:
8        # Get feedback from the customer
9          feedback = input("Enter customer feedback (or 'stop' to finish): ")
10         # Add the feedback to the list
11         if feedback.lower() == 'stop':
12           # Exit the loop if the user enters 'stop'
13             break
14             # Add the feedback to the list
15         feedback_list.append(feedback)
16     # Print the collected feedback
17     print("\nCollected Feedback:")
18     # Use an enumerate function to get the customer number
19     for i, feedback in enumerate(feedback_list, 1):
20       # Print the feedback for each customer
21         print(f"Customer {i}: {feedback}")
22
23 # Run the program
24 feedback_while_loop()
```

3.2.2 Screenshots of the results

```
Enter customer feedback (or 'stop' to finish): good service
Enter customer feedback (or 'stop' to finish): stop

Collected Feedback:
Customer 1: good service
```

Figure 6: Feedback_while_loop

3.2.3 Explanation of our solution

This function gathers feedback from customers until they type ”stop.” It saves each feedback entry in a list and displays it in an organized way when finished. The code uses a while loop to keep asking for feedback. If the input is ”stop,” it exits the loop. Otherwise, it adds the feedback to a list. Once the loop stops, it prints each feedback entry with a customer number, providing an organized view of all collected feedback.

3.3 An anlysis

In the world of retail, understanding customer experiences is crucial. Many stores implement feedback collection systems to gauge customer satisfaction and identify areas for improvement. As we develop such a system, we face a fundamental programming decision: should we use a WHILE loop or a FOR loop to manage the feedback collection process?

Our initial tests showed some interesting results. The FOR loop completed 1000 feedback entries in just 0.000483 seconds, while the WHILE loop took 0.001687 seconds. At first glance, this makes the FOR loop about 71.39% faster. But in the complex environment of a busy store, is raw speed the most important factor? Let's dive deeper into this question and explore the nuances of each loop type in our specific context.

3.3.1 The Nature of Retail Feedback Collection

Before we analyze the loops themselves, it's important to understand the environment in which our program will operate. Retail stores are dynamic places. Customer traffic ebbs and flows throughout the day, influenced by factors like time, weather, local events, and more. Some customers are in a hurry, while others are happy to chat. Some may be eager to share their thoughts, while others might need a bit of encouragement.

Our feedback collection system needs to be flexible enough to handle this variability. It should be able to collect feedback when customers are available, pause when the store gets busy, and stop when we've gathered enough data or when it's time to close shop. With this context in mind, let's examine our loop options.

3.3.2 The WHILE Loop: Flexibility in Action

The WHILE loop is like a patient sales associate. It stands ready to collect feedback for as long as necessary, adapting to the rhythm of the store.

This loop will keep running as long as the 'collecting' variable is True. We can change this variable based on various factors: time of day, number of responses collected, or even a direct input from the store manager.

The strength of the WHILE loop lies in its flexibility. On a quiet day, it can keep running, collecting feedback from every available customer. On a busy day, we can easily interrupt it after each iteration. This adaptability is invaluable in a retail environment where conditions can change rapidly.

Moreover, the WHILE loop makes it easy to add additional checks or processes between each feedback collection.

This flexibility comes at a cost, though. As our tests showed, the WHILE loop is slightly slower than the FOR loop. It also requires us to manually manage the loop's continuation condition, which could potentially lead to errors if not handled carefully.

3.3.3 The FOR Loop: Speed and Simplicity

The FOR loop, on the other hand, is like a focused interviewer with a quota to meet. It's designed to run a specific number of times, collecting a predetermined amount of feedback.

The advantage of the FOR loop is its simplicity and speed. It's straightforward to write and understand, and as our tests showed, it executes more quickly than the WHILE loop.

In scenarios where we know exactly how many feedback entries we want to collect, the FOR loop shines. It's efficient and leaves little room for logical errors. The loop will run exactly the number of times we specify, no more and no less.

However, this predetermined nature can be a drawback in a retail environment. What if we set the loop to collect 100 responses, but on a particularly slow day, we only get 50 customers? The program would keep running, potentially confusing staff or skewing our data. Conversely, on a busy day, we might hit our target quickly and miss out on valuable additional feedback.

3.3.4 Practical Considerations in a Retail Environment

When we step back and look at the bigger picture of retail operations, the speed difference between our WHILE and FOR loops (about 0.001204 seconds over 1000 entries) becomes almost insignificant. In the time it takes a customer to consider and voice their feedback, or for a staff member to input that feedback into the system, those fractions of a second disappear into the background.

What becomes more important is how well our chosen loop integrates into the daily flow of the store. Can it be easily started and stopped as the store gets busy or quiet? Can it adapt if we decide to change our feedback collection strategy mid-day? Can it handle unexpected situations, like a customer changing their mind halfway through giving feedback?

The WHILE loop, with its inherent flexibility, seems to answer "yes" to all these questions. It allows for easy interruption, simple adaptation, and straightforward error handling.

This kind of adaptive behavior is much harder to implement with a FOR loop, which is designed to run for a set number of iterations.

3.3.5 Looking to the Future

While our current analysis favors the WHILE loop for its flexibility in a dynamic retail environment, it's important to consider future developments. As technology evolves, so too might our feedback collection methods.

For instance, if we implement an automated feedback system where customers input their responses directly into a kiosk or mobile app, the speed advantage of the FOR loop might become more relevant. In such a system, we might want to collect a specific number of responses each day, making the FOR loop a more attractive option.

Similarly, if we integrate our feedback system with a broader customer relationship management (CRM) system, we might find that a more structured, predictable approach using a FOR loop aligns better with the CRM's data handling processes.

3.3.6 Conclusion

In the current retail scenario, where flexibility and adaptability are key, the WHILE loop appears to be the better choice for our feedback collection system. Its ability to adjust to the ebb and flow of customer traffic and easily incorporate interruptions and checks outweighs the minor performance advantage of the FOR loop.

However, it's crucial to remember that in programming, as in retail, there's rarely a one-size-fits-all solution. The best choice depends on the specific needs of the store, the nature of the feedback being collected, and the broader systems with which our program must integrate.

As we move forward with implementing this system, we should remain open to reassessing our choice. We might find that a hybrid approach, using WHILE loops for some aspects of the program and FOR loops for others, provides the best balance of flexibility and efficiency.

Ultimately, the goal is to create a feedback collection system that not only works well technically, but also integrates seamlessly into the day-to-day operations of the store. By carefully considering the pros and cons of each loop type in the context of real-world retail operations, we can create a system that helps store managers gather valuable customer insights without adding unnecessary complexity to their already busy days.

```python

import time

def collect_feedback_for_loop():
    num_customers = 1000  # Simulate feedback for 1000 customers
    feedback_list = []

    start_time = time.time()
    for i in range(num_customers):
        feedback = f"Sample feedback for customer {i+1}"
        feedback_list.append(feedback)
    end_time = time.time()

    return end_time - start_time, feedback_list

def collect_feedback_while_loop():
    num_customers = 1000  # Simulate feedback for 1000 customers
    feedback_list = []
    count = 0

    start_time = time.time()
    while count < num_customers:
        feedback = f"Sample feedback for customer {count+1}"
        feedback_list.append(feedback)
        count += 1
    end_time = time.time()

    return end_time - start_time, feedback_list

# Measure the time for the FOR loop
for_loop_time, for_loop_feedback = collect_feedback_for_loop()

# Measure the time for the WHILE loop
while_loop_time, while_loop_feedback = collect_feedback_while_loop()

print(f"FOR loop execution time: {for_loop_time:.6f} seconds")
print(f"WHILE loop execution time: {while_loop_time:.6f} seconds")

# Verify that both loops collected the same feedback
print(f"\nBoth loops collected the same feedback: {for_loop_feedback ==
    while_loop_feedback}")
print(f"Number of feedback entries: {len(for_loop_feedback)}")
```

```
FOR loop execution time: 0.000483 seconds
WHILE loop execution time: 0.001687 seconds
FOR loop is faster by 71.39%

Both loops collected the same feedback: True
Number of feedback entries: 1000
```

Figure 7: Speed analysis

4 Problem 4

4.1 R code

```r
library(tidyverse) #for using tibble and pipes

#data frame for the routes data
df <- tibble(route = c("City A -> B", "City B -> C", "City A -> C"),
             distance = c(350, 500, 700),
             ferries = c(1, 2, 3),
             tolls = c(3, 5, 6)
             )

#list of cost data
electric <- list(0.20, 180, 40)
diesl <- list(1.5, 300, 80)


#Calculate the savings
calc <- df %>%
  mutate(et_total = distance * electric[[1]] + ferries * electric[[2]] + tolls
      * electric[[3]],
         dt_total = distance * diesl[[1]] + ferries * diesl[[2]] + tolls *
             diesl[[3]],
         savings = dt_total - et_total)

#display of results
calc
```

4.2 Explanation of our solution

The code calculates and compares travel costs for electric and diesel options on various routes. Using mutate from dplyr, it creates columns for each route's total electric and diesel costs by combining distance, ferry trips, and tolls with their respective per-unit costs from the electric and diesl lists. It then calculates the savings by subtracting the electric cost from the diesel cost, showing potential savings for each route. The final data frame displays each route's original data and calculated costs, making it easy to see the cost differences between electric and diesel travel

4.3 Screenshots of the results

```
# A tibble: 3 × 7
  route       distance ferries tolls et_total dt_total savings
  <chr>          <dbl>   <dbl> <dbl>    <dbl>    <dbl>   <dbl>
1 City A -> B      350       1     3      370     1065     695
2 City B -> C      500       2     5      660     1750    1090
3 City A -> C      700       3     6      920     2430    1510
```

Figure 8: Display results in RStudio Console

5 Problem 5

5.1 R code

```r
# Define item types and their costs
item_types <- c("small", "medium", "large", "extra-large", "luxury", "premium",
    "bulk")
costs <- c(1, 2, 5, 10, 20, 50, 100)

# Create a dataframe with item types and costs
item_costs_df <- data.frame(Item_Type = item_types, Cost = costs)

# Initialize a vector to store the number of ways to make each amount
# Index i represents amount (i-1), so we need 201 elements for 0 to 200
ways <- rep(0, 201)
ways[1] <- 1  # There's one way to make 0 (by selecting nothing)

# Use dynamic programming to calculate combinations
for (cost in costs) {
  for (amount in cost:200) {
    # Update the number of ways to make 'amount'
    # by adding the number of ways to make 'amount - cost'
    ways[amount + 1] <- ways[amount + 1] + ways[amount - cost + 1]
  }
}

# Print the result (ways[201] represents the number of ways to make 200)
print(paste("Number of possible purchase combinations:", ways[201]))
```

5.2 Explanation of our solution

This R code calculates how many different ways a retailer can spend exactly $200 on various items. It uses a method called dynamic programming to efficiently count all possible combinations of items, considering their different costs. The code accounts for repeated use of items and excludes the $200 wholesale pack. The final number, stored in ways201, shows how many unique ways the retailer can combine items to reach $200, highlighting the flexibility in stock replenishment options.

5.3 Screeshots of the results

```
>
> # Print the result (ways[201] represents the number of ways to make 200)
> print(paste("Number of possible purchase combinations:", ways[201]))
[1] "Number of possible purchase combinations: 73681"
```

Figure 9: Combinations

6 Problem 6

6.1 Database idea

Our database is created to support Norway's thriving fishing and aquaculture industry by focusing on several important areas. First, it keeps track of all the aquaculture facilities across Norway, including where they are located, how much they can produce, who owns or runs them, and their licensing information. This is important because Norway produced over 1.5 million tonnes of salmon in 2023, making up more than half of the world's total production (Fiskeridirektoratet, n.d.).

The database also monitors the fish populations at each facility. It records the types of fish, how many there are, their average weight, when they were stocked, and when they are expected to be harvested. This helps ensure that the fish are healthy and growing properly, addressing key concerns from the Norwegian Veterinary Institute (Veterinærinstituttet, n.d.).

Environmental conditions are another crucial part of the database. It tracks factors like water temperature, oxygen levels, salinity, algae concentration, and the number of lice. Monitoring these conditions helps maintain sustainable practices and tackle environmental issues such as escaped fish and salmon lice infestations (Miljødirektoratet, n.d.).

Feeding practices are also recorded, including the types of feed used, the amount consumed, and how efficiently the feed is converted into fish growth. This supports resource-efficient aquaculture, as emphasized by the Norwegian Institute of Marine Research (Havforskningsinstituttet, n.d.).

Health management is covered by tracking any diseases or health issues that arise, the treatments applied, and the outcomes. This ensures that all seafood meets the strict safety standards set by the Norwegian Food Safety Authority (Mattilsynet, n.d.).

The database manages information related to harvesting and exporting fish, such as when and how much is harvested, the quality of the catch, where it is exported, and the sale prices. This is vital for Norway's seafood exports, which reach over 140 countries (Norges sjømatråd, n.d.).

Lastly, the database includes employee information, keeping records of all workers at the facilities, their roles, hire dates, and salaries. This highlights the industry's role in creating jobs and adding value along Norway's coast (Sjømat Norge, n.d.).

In summary, our database covers all major aspects of Norway's fishing and aquaculture industries. It helps manage operations efficiently, ensures sustainability, and supports the industry's growth by providing valuable data for decision-making.

6.2 ER-model

This ER model represents the structure of a database system for the fisheries and aquaculture industry, showcasing various entities and their relationships. The core of the model is the Facilities table, which connects to different operational aspects, such as environmental data, fish stock, feeding, and export destinations. Other key tables include Employees, which handles employee details, Health Management, focusing on disease and treatment tracking, and Suppliers, providing information on suppliers of feed and other necessary resources. The model also includes licenses and owners, offering a comprehensive view of the administrative and operational infrastructure of a fish facility.

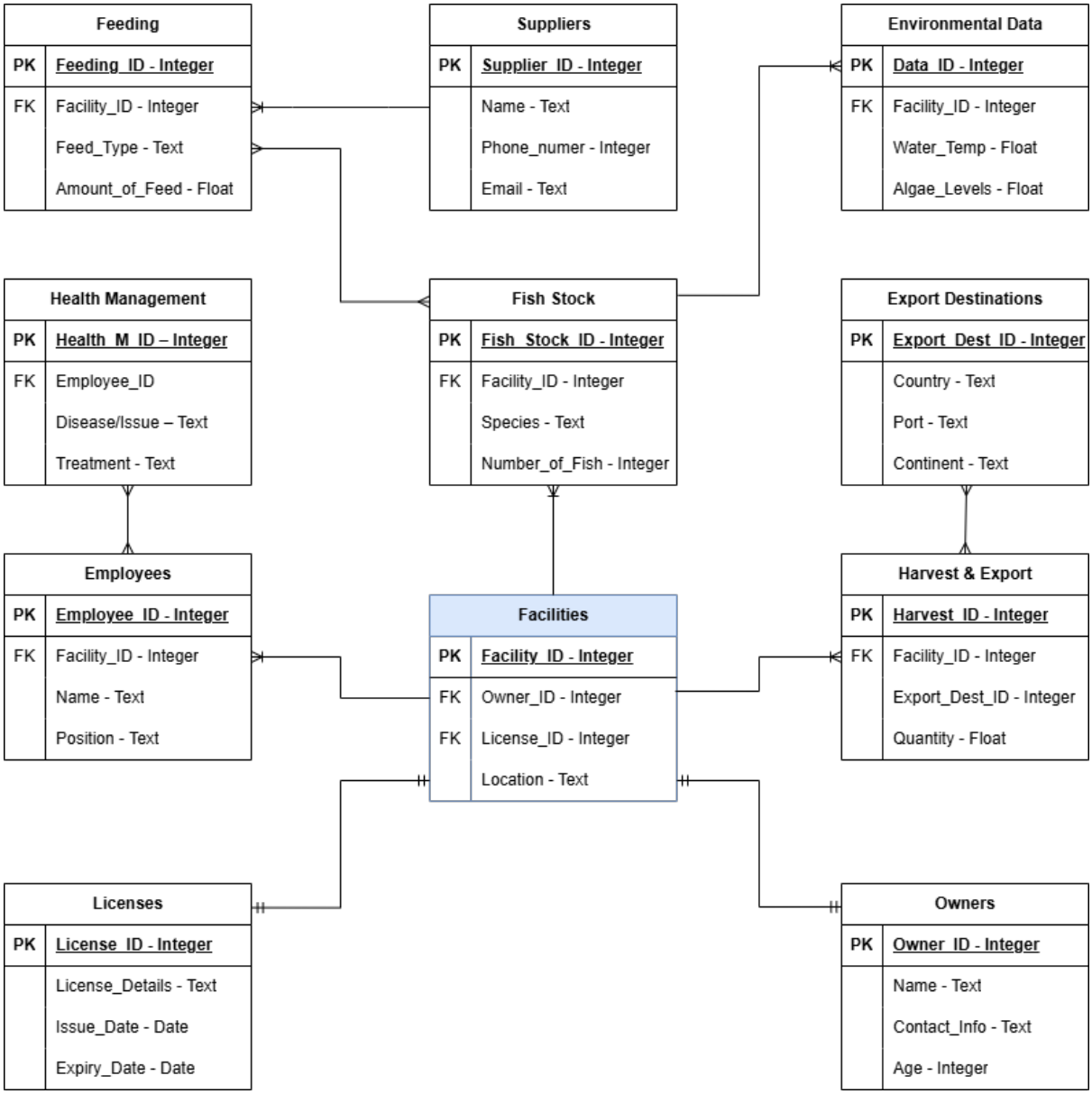

Figure 10: ER_Model

6.3 Database creation

6.3.1 SQL code

The following SQL code creates tables for our database. Each table includes a primary key and, where relevant, foreign key constraints to maintain relationships between tables. The code executed successfully without errors.

```
CREATE TABLE Facilities (
    Facility_ID INTEGER PRIMARY KEY,
    Owner_ID INTEGER,
    License_ID INTEGER,
    Location TEXT,
    FOREIGN KEY (Owner_ID) REFERENCES Owners(Owner_ID),
    FOREIGN KEY (License_ID) REFERENCES Licenses(License_ID)
);

CREATE TABLE Owners (
    Owner_ID INTEGER PRIMARY KEY,
    Name TEXT,
    Contact_Info TEXT,
    Age INTEGER
);

CREATE TABLE Licenses (
    License_ID INTEGER PRIMARY KEY,
    License_Details TEXT,
    Issue_Date DATE,
    Expiry_Date DATE
);

CREATE TABLE Employees (
    Employee_ID INTEGER PRIMARY KEY,
    Facility_ID INTEGER,
    Name TEXT,
    Position TEXT,
    FOREIGN KEY (Facility_ID) REFERENCES Facilities(Facility_ID)
);

CREATE TABLE Health_Management (
    Health_M_ID INTEGER PRIMARY KEY,
    Employee_ID INTEGER,
    Disease_Issue TEXT,
    Treatment TEXT,
    FOREIGN KEY (Employee_ID) REFERENCES Employees(Employee_ID)
);

CREATE TABLE Feeding (
    Feeding_ID INTEGER PRIMARY KEY,
    Facility_ID INTEGER,
    Supplier_ID INTEGER,
    Feed_Type TEXT,
    Amount_of_Feed FLOAT,
    FOREIGN KEY (Facility_ID) REFERENCES Facilities(Facility_ID),
    FOREIGN KEY (Supplier_ID) REFERENCES Suppliers(Supplier_ID)
);

CREATE TABLE Suppliers (
```

```
    Supplier_ID INTEGER PRIMARY KEY,
    Name TEXT,
    Phone_number INTEGER,
    Email TEXT
);

CREATE TABLE Fish_Stock (
    Fish_Stock_ID INTEGER PRIMARY KEY,
    Facility_ID INTEGER,
    Species TEXT,
    Number_of_Fish INTEGER,
    FOREIGN KEY (Facility_ID) REFERENCES Facilities(Facility_ID)
);

CREATE TABLE Environmental_Data (
    Data_ID INTEGER PRIMARY KEY,
    Facility_ID INTEGER,
    Water_Temp FLOAT,
    Algae_Levels FLOAT,
    FOREIGN KEY (Facility_ID) REFERENCES Facilities(Facility_ID)
);

CREATE TABLE Harvest_Export (
    Harvest_ID INTEGER PRIMARY KEY,
    Facility_ID INTEGER,
    Export_Dest_ID INTEGER,
    Quantity FLOAT,
    FOREIGN KEY (Facility_ID) REFERENCES Facilities(Facility_ID),
    FOREIGN KEY (Export_Dest_ID) REFERENCES Export_Destinations(Export_Dest_ID)
);

CREATE TABLE Export_Destinations (
    Export_Dest_ID INTEGER PRIMARY KEY,
    Country TEXT,
    Port TEXT,
    Continent TEXT
);
```

6.4 Creating Samples

Owners Table
Inserting data into the ‘Owners‘ table using the ‘INSERT INTO Owners‘ command.

```
INSERT INTO Owners (Owner_ID, Name, Contact_Info, Age) VALUES
(1, 'Ola Hansen', 'ola.hansen@example.com', 45),
(2, 'Kari Nordmann', 'kari.nordmann@example.com', 39),
(3, 'Per Johansen', 'per.johansen@example.com', 50),
(4, 'Ingrid Olsen', 'ingrid.olsen@example.com', 42),
(5, 'Einar Larsen', 'einar.larsen@example.com', 36);
```

Licenses Table
Inserting data into the ‘Licenses‘ table using the ‘INSERT INTO Licenses‘ command.

```
INSERT INTO Licenses (License_ID, License_Details, Issue_Date, Expiry_Date) VALUES
(1, 'Standard Lisens', '2022-01-01', '2027-01-01'),
(2, 'Spesiell Lisens', '2021-06-15', '2026-06-15'),
(3, 'Utvidet Lisens', '2023-03-10', '2028-03-10'),
```

```
(4, 'Eksport Lisens', '2020-09-20', '2025-09-20'),
(5, 'Forskning Lisens', '2019-12-05', '2024-12-05');
```

Facilities Table
Inserting data into the 'Facilities' table using the 'INSERT INTO Facilities' command.

```
INSERT INTO Facilities (Facility_ID, Owner_ID, License_ID, Location) VALUES
(1, 1, 1, 'Lofoten'),
(2, 2, 2, 'Tromsø'),
(3, 3, 3, 'Bergen'),
(4, 4, 4, 'Ålesund'),
(5, 5, 5, 'Stavanger');
```

Employees Table
Inserting data into the 'Employees' table using the 'INSERT INTO Employees' command.

```
INSERT INTO Employees (Employee_ID, Facility_ID, Name, Position) VALUES
(1, 1, 'Lars Pedersen', 'Driftsleder'),
(2, 2, 'Eva Jakobsen', 'Fiskepleier'),
(3, 3, 'Jonas Nilsen', 'Tekniker'),
(4, 4, 'Nina Kristiansen', 'Biolog'),
(5, 5, 'Kristoffer Ås', 'Kvalitetsinspektør');
```

Health Management Table
Inserting data into the 'Health_Management' table using the 'INSERT INTO Health_Management' command.

```
INSERT INTO Health_Management (Health_M_ID, Employee_ID, Disease_Issue, Treatment) VALUES
(1, 1, 'Lus', 'Lusebehandling'),
(2, 2, 'Fiskelus', 'Varmebehandling'),
(3, 3, 'Fungal Infection', 'Antifungal behandling'),
(4, 4, 'Gillsykdom', 'Antibiotika'),
(5, 5, 'Skinnsår', 'Saltbad');
```

Suppliers Table
Inserting data into the 'Suppliers' table using the 'INSERT INTO Suppliers' command.

```
INSERT INTO Suppliers (Supplier_ID, Name, Phone_number, Email) VALUES
(1, 'Fiskefôr AS', 98765432, 'kontakt@fiskefor.no'),
(2, 'Havbruk Mat', 91234567, 'post@havbrukmat.no'),
(3, 'Laksefôr Norge', 98765431, 'info@laksefor.no'),
(4, 'Norsk Fôrindustri', 92345678, 'kundeservice@norskforind.no'),
(5, 'Maritim Fiskefôr', 97654321, 'support@maritimfor.no');
```

Feeding Table
Inserting data into the 'Feeding' table using the 'INSERT INTO Feeding' command.

```
INSERT INTO Feeding (Feeding_ID, Facility_ID, Supplier_ID, Feed_Type, Amount_of_Feed) VALUES
(1, 1, 1, 'Pellet', 120.5),
(2, 2, 2, 'Våtfôr', 95.3),
(3, 3, 3, 'Tørrfôr', 110.2),
(4, 4, 4, 'Granulat', 89.7),
(5, 5, 5, 'Spesialfôr', 130.0);
```

Fish Stock Table
Inserting data into the 'Fish_Stock' table using the 'INSERT INTO Fish_Stock' command.

```
INSERT INTO Fish_Stock (Fish_Stock_ID, Facility_ID, Species, Number_of_Fish) VALUES
(1, 1, 'Laks', 50000),
(2, 2, 'Ørret', 30000),
(3, 3, 'Kveite', 25000),
(4, 4, 'Torsk', 45000),
(5, 5, 'Sei', 20000);
```

Environmental Data Table
Inserting data into the 'Environmental_Data' table using the 'INSERT INTO Environmental_Data' command.

```
INSERT INTO Environmental_Data (Data_ID, Facility_ID, Water_Temp, Algae_Levels) VALUES
(1, 1, 13.5, 0.3),
(2, 2, 14.2, 0.2),
(3, 3, 12.8, 0.4),
(4, 4, 15.0, 0.5),
(5, 5, 13.0, 0.1);
```

Export Destinations Table
Inserting data into the 'Export_Destinations' table using the 'INSERT INTO Export_Destinations' command.

```
INSERT INTO Export_Destinations (Export_Dest_ID, Country, Port, Continent) VALUES
(1, 'Norge', 'Oslo', 'Europa'),
(2, 'Sverige', 'Stockholm', 'Europa'),
(3, 'Danmark', 'København', 'Europa'),
(4, 'Tyskland', 'Hamburg', 'Europa'),
(5, 'USA', 'New York', 'Nord-Amerika');
```

Harvest & Export Table
Inserting data into the 'Harvest_Export' table using the 'INSERT INTO Harvest_Export' command.

```
INSERT INTO Harvest_Export (Harvest_ID, Facility_ID, Export_Dest_ID, Quantity) VALUES
(1, 1, 1, 2000),
(2, 2, 2, 2500),
(3, 3, 3, 1500),
(4, 4, 4, 3000),
(5, 5, 5, 3500);
```

6.4.1 Execution Result

The output confirms that the code was executed successfully:

```
Execution finished without errors.
Result: query executed successfully. Took 0ms
At line 173:
-- Insert sample data into Harvest & Export Table
INSERT INTO Harvest_Export (Harvest_ID, Facility_ID, Export_Dest_ID, Quantity) VALUES
(1, 1, 1, 2000),
(2, 2, 2, 2500),
(3, 3, 3, 1500),
(4, 4, 4, 3000),
(5, 5, 5, 3500);
```

6.4.2 For Each Relationship

Facilities - Owners (1:1): This relationship links each facility to a unique owner, providing detailed information on ownership. It is a 1:1 relationship where one facility is connected to one owner.

Facilities - Licenses (1:1): This relationship connects a facility to its specific license, ensuring regulatory compliance is tracked effectively. It is a 1:1 relationship, with each facility holding one license and each license belonging to one facility.

Facilities - Fish Stock (1:N): This relationship associates a facility with multiple fish stock records, which is essential for inventory management. It is a 1:N relationship where one facility can have many fish stocks, but each fish stock is tied to one facility.

Facilities - Employees (1:N): This relationship links a facility to its employees for efficient staff management. It is a 1:N relationship, as a facility can have many employees, but each employee is associated with only one facility.

Fish Stock - Environmental Data (1:N): This relationship connects fish stock data to environmental data entries, facilitating analysis of environmental impacts on fish stocks. It is a 1:N relationship, where one fish stock can have multiple environmental data entries associated with it.

Feeding - Fish Stock (N:N): This relationship tracks feeding regimens applied to different fish stocks, enabling detailed feeding strategy management. It is an N:N relationship, allowing multiple feeding records to be linked to multiple fish stocks and vice versa.

Employees - Health Management (N:N): This relationship links employees to health management tasks, allowing tracking of employee involvement in various health activities. It is an N:N relationship, where many employees can engage in many health management tasks.

Harvest & Export - Export Destinations (N:N): This relationship connects harvested and exported fish with different export destinations for distribution tracking. It is an N:N relationship, where many harvests can be linked to multiple destinations and each destination can receive exports from multiple harvests.

Facilities - Harvest & Export (1:N): This relationship ties facilities to their respective harvest and export records, managing the production and export activities from each facility. It is a 1:N relationship, with one facility having many related harvest and export records.

Suppliers - Feeding (1:N): This relationship links suppliers to feeding records to track the source of feed for various feeding processes. It is a 1:N relationship where one supplier can supply feed to multiple feeding records, but each feeding record is linked to a single supplier.

6.4.3 For each table

Facilities: This table represents the various fish farming facilities and holds essential information related to their operations and management. The primary key is *Facility ID*, which uniquely identifies each facility. The foreign keys *Owner ID* and *License ID* link each facility to its owner and license, creating relationships with the *Owners* and *Licenses* tables.

Owners: This table contains information about the individuals or entities that own the facilities. The primary key *Owner ID* ensures each owner is uniquely identifiable in the database, allowing clear connections to the *Facilities* table.

Licenses: This table stores details of the licenses held by facilities to maintain regulatory compliance. The primary key *License ID* uniquely identifies each license, ensuring that each license entry can be traced to a specific facility through the *Facilities* table.

Fish Stock: The *Fish Stock* table represents the various fish stocks present in facilities, including information on species and number of fish. The primary key is *Fish Stock ID*, providing unique identification for each entry, while the foreign key *Facility ID* connects fish stocks to the facilities where they are kept.

Environmental Data: This table holds environmental data affecting fish stocks, such as water temperature and algae levels. The primary key *Data ID* uniquely identifies each environmental entry. The foreign key *Facility ID* links the data to the specific facility to provide context on environmental conditions.

Feeding: The *Feeding* table records feeding processes, including the type of feed used and the amount. The primary key is *Feeding ID*, which uniquely identifies each entry. The foreign keys *Facility ID* and *Supplier ID* link the feeding processes to the facilities and suppliers involved, allowing tracking of supply sources and feeding activities.

Suppliers: This table represents the suppliers providing feed and other services to facilities. The primary key *Supplier ID* ensures that each supplier is uniquely identifiable, facilitating the association with feeding activities and facilities.

Employees: The *Employees* table includes information about employees working at the facilities, such as their names and positions. The primary key *Employee ID* uniquely identifies each employee, and the foreign key *Facility ID* links employees to the facilities where they work.

Health Management: This table records health management activities related to fish stocks, including disease treatment and issues. The primary key *Health ID* uniquely identifies each health record, while the foreign key *Employee ID* associates health activities with the responsible employees.

Harvest & Export: The *Harvest & Export* table tracks fish harvesting and export data, such as the quantity exported. The primary key *Harvest ID* uniquely identifies each entry. The foreign keys *Facility ID* and *Export Dest ID* connect the harvest data to the facilities and export destinations, respectively.

Export Destinations: This table details the export destinations for harvested fish, including country and port information. The primary key *Export Dest ID* uniquely identifies each export destination, ensuring traceability and clear connections with the *Harvest & Export* table.

References

Fiskeridirektoratet. (n.d.). *Oppdrett av laks i Norge*. Retrieved from `https://www.fiskeridir.no`

Veterinærinstituttet. (n.d.). *Fiskehelse og fiskevelferd*. Retrieved from `https://www.vetinst.no`

Miljødirektoratet. (n.d.). *Akvakultur og miljøutfordringer*. Retrieved from `https://www.miljodirektoratet.no`

Havforskningsinstituttet. (n.d.). *Ressurseffektiv matproduksjon*. Retrieved from `https://www.hi.no`

Mattilsynet. (n.d.). *Mattrygghet i sjømatnæringen*. Retrieved from `https://www.mattilsynet.no`

Norges sjømatråd. (n.d.). *Eksport av norsk sjømat*. Retrieved from `https://www.seafood.no`

Sjømat Norge. (n.d.). *Verdiskaping langs kysten*. Retrieved from `https://www.sjomatnorge.no`