0. Imports and Setting up Anthropic API Client

```
from google.colab import drive

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install python-dotenv

import os
import dotenv

dotenv.load_dotenv('/content/drive/MyDrive/.env')
```

```
Collecting python-dotenv
  Downloading python_dotenv-1.0.1-py3-none-any.whl (19 kB)
Installing collected packages: python-dotenv
Successfully installed python-dotenv-1.0.1
True
```

```
# Load Prompts and Problem Description
prompt1_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt1_MathematicalModel.txt'
prompt2_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt2_PyomoCode.txt'
problem_desc_path = '/content/drive/MyDrive/Thesis/ProblemDescriptions/LP/LP4.txt'

prompt1_file = open(prompt1_path, "r")
prompt2_file = open(prompt2_path, "r")
problem_desc_file = open(problem_desc_path, "r")

prompt1 = prompt1_file.read()
print("Prompt 1:\n", prompt1)

prompt2 = prompt2_file.read()
print("Prompt 2:\n", prompt2)

problem_desc = problem_desc_file.read()
print("Problem Description:\n", problem_desc)
```

```
Prompt 1:
 Please write a mathematical optimization model for this problem. Include parameters, decision variables, the objective
Prompt 2:
 Please write a python pyomo code for this optimization problem.
Use sample data where needed.
Indicate where you use sample data.
Problem Description:
 Consider a farmer who specializes in raising wheat, corn, and sugar beets on her 500 acres of land. During the winter,

The farmer knows that at least 200 tons (T) of wheat and 240 T of corn are needed for cattle feed. These amounts can be
Any production in excess of the feeding requirement would be sold.

Over the last decade, mean selling prices have been $170 and $150 per ton of wheat and corn, respectively.
The purchase prices are 40% more than this due to the wholesaler’s margin and transportation costs.

Another profitable crop is sugar beet, which she expects to sell at $36/T. However, the European Commission imposes a qu

Based on past experience, the farmer knows that the mean yield on her land is roughly 2.5 T, 3 T, and 20 T per acre for

The farmer can decide how much of her land she will use to grow each of the three products: wheat, corn and sugar beets.
```

```
!pip install anthropic
```

Show hidden output

```
# Importing Anthropic & Setting Headers
import anthropic

client = anthropic.Anthropic(
    # defaults to os.environ.get("ANTHROPIC_API_KEY")
    api_key=os.environ.get("ANTHROPIC_API_KEY"),
)
```

1. Prompt 1: Create Mathematical Model

```
message = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt1,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": problem_desc}
    ]
)

response_p1 = message.content[0].text


# Print response
print(response_p1)
```

```
Let's formulate the mathematical optimization model for the farmer's problem.

Parameters:
- W_min = 200: minimum requirement of wheat (in tons)
- C_min = 240: minimum requirement of corn (in tons)
- P_w = 170: selling price of wheat ($/ton)
- P_c = 150: selling price of corn ($/ton)
- P_wb = 1.4 * P_w: purchase price of wheat ($/ton)
- P_cb = 1.4 * P_c: purchase price of corn ($/ton)
- P_sb = 36: selling price of sugar beet ($/ton)
- P_sbx = 10: selling price of excess sugar beet ($/ton)
- Q_sb = 6000: quota for sugar beet production (in tons)
- Y_w = 2.5: yield of wheat (tons/acre)
- Y_c = 3: yield of corn (tons/acre)
- Y_sb = 20: yield of sugar beet (tons/acre)
- C_w = 150: cost to plant an acre of wheat ($/acre)
- C_c = 230: cost to plant an acre of corn ($/acre)
- C_sb = 260: cost to plant an acre of sugar beet ($/acre)
- L = 500: total available land (in acres)

Decision Variables:
- w: acres of land allocated to wheat
- c: acres of land allocated to corn
- sb: acres of land allocated to sugar beet
- w_b: tons of wheat bought from the wholesaler
- c_b: tons of corn bought from the wholesaler
- sb_x: tons of sugar beet produced in excess of the quota

Objective Function:
minimize    C_w * w + C_c * c + C_sb * sb + P_wb * w_b + P_cb * c_b - P_w * max(Y_w * w - W_min, 0) - P_c * max(Y_c * c

Constraints:
1. w + c + sb <= L                     (land constraint)
2. Y_w * w + w_b >= W_min              (wheat requirement)
3. Y_c * c + c_b >= C_min              (corn requirement)
4. Y_sb * sb = Q_sb + sb_x             (sugar beet quota)
5. w, c, sb, w_b, c_b, sb_x >= 0       (non-negativity constraints)

The objective function minimizes the total costs, considering planting costs, purchase costs, and sales revenues. The ma

The constraints ensure that:
1. The total land allocated to the three crops does not exceed the available land.
2. The minimum requirements for wheat and corn are met through production and/or purchase.
3. The sugar beet production is equal to the quota plus any excess production.
4. All decision variables are non-negative.
```

2. Prompt 2: Write Code based on Mathematical Model

```
message2 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt2,
    max_tokens=4069,
    messages=[
        {"role": "user", "content": response_p1}
    ]
)

response_p2 = message2.content[0].text


# Print response
print(response_p2)
```

```
Here's the Python code for the farmer's optimization problem using Pyomo:

```python
from pyomo.environ import *

# Create a Concrete Model
```
```

```
model = ConcreteModel()

# Define the parameters
model.W_min = 200
model.C_min = 240
model.P_w = 170
model.P_c = 150
model.P_wb = 1.4 * model.P_w
model.P_cb = 1.4 * model.P_c
model.P_sb = 36
model.P_sbx = 10
model.Q_sb = 6000
model.Y_w = 2.5
model.Y_c = 3
model.Y_sb = 20
model.C_w = 150
model.C_c = 230
model.C_sb = 260
model.L = 500

# Define the decision variables
model.w = Var(domain=NonNegativeReals)
model.c = Var(domain=NonNegativeReals)
model.sb = Var(domain=NonNegativeReals)
model.w_b = Var(domain=NonNegativeReals)
model.c_b = Var(domain=NonNegativeReals)
model.sb_x = Var(domain=NonNegativeReals)

# Define the objective function
def obj_rule(model):
    return (
        model.C_w * model.w + model.C_c * model.c + model.C_sb * model.sb +
        model.P_wb * model.w_b + model.P_cb * model.c_b -
        model.P_w * max(model.Y_w * model.w - model.W_min, 0) -
        model.P_c * max(model.Y_c * model.c - model.C_min, 0) -
        model.P_sb * min(model.Y_sb * model.sb, model.Q_sb) -
        model.P_sbx * model.sb_x
    )
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define the constraints
def land_constraint_rule(model):
    return model.w + model.c + model.sb <= model.L
model.land_constraint = Constraint(rule=land_constraint_rule)

def wheat_requirement_rule(model):
    return model.Y_w * model.w + model.w_b >= model.W_min
model.wheat_requirement = Constraint(rule=wheat_requirement_rule)

def corn_requirement_rule(model):
    return model.Y_c * model.c + model.c_b >= model.C_min
```

4. Input Problem Data and Test Model Code

```
# Installing pyomo nd solver
!pip install -q pyomo
!apt-get install -y -qq glpk-utils
!pip install glpk
```

Show hidden output

```
from pyomo.environ import *

# Create a Concrete Model
model = ConcreteModel()

# Define the parameters
model.W_min = 200
model.C_min = 240
model.P_w = 170
model.P_c = 150
model.P_wb = 1.4 * model.P_w
model.P_cb = 1.4 * model.P_c
model.P_sb = 36
model.P_sbx = 10
model.Q_sb = 6000
model.Y_w = 2.5
model.Y_c = 3
model.Y_sb = 20
model.C_w = 150
model.C_c = 230
model.C_sb = 260
model.L = 500

# Define the decision variables
model.w = Var(domain=NonNegativeReals)
model.c = Var(domain=NonNegativeReals)
model.sb = Var(domain=NonNegativeReals)
model.w_b = Var(domain=NonNegativeReals)
model.c_b = Var(domain=NonNegativeReals)
model.sb_x = Var(domain=NonNegativeReals)

# Define the objective function
def obj_rule(model):
    return (
        model.C_w * model.w + model.C_c * model.c + model.C_sb * model.sb +
        model.P_wb * model.w_b + model.P_cb * model.c_b -
        model.P_w * max(model.Y_w * model.w - model.W_min, 0) -
        model.P_c * max(model.Y_c * model.c - model.C_min, 0) -
        model.P_sb * min(model.Y_sb * model.sb, model.Q_sb) -
        model.P_sbx * model.sb_x
    )
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define the constraints
def land_constraint_rule(model):
    return model.w + model.c + model.sb <= model.L
model.land_constraint = Constraint(rule=land_constraint_rule)

def wheat_requirement_rule(model):
    return model.Y_w * model.w + model.w_b >= model.W_min
model.wheat_requirement = Constraint(rule=wheat_requirement_rule)

def corn_requirement_rule(model):
    return model.Y_c * model.c + model.c_b >= model.C_min
model.corn_requirement = Constraint(rule=corn_requirement_rule)

def sugar_beet_quota_rule(model):
    return model.Y_sb * model.sb == model.Q_sb + model.sb_x
model.sugar_beet_quota = Constraint(rule=sugar_beet_quota_rule)

# Solve the optimization problem
solver = SolverFactory('glpk')
results = solver.solve(model)

# Print the results
print("Objective value:", value(model.obj))
print("Acres of wheat:", value(model.w))
print("Acres of corn:", value(model.c))
print("Acres of sugar beet:", value(model.sb))
print("Tons of wheat bought:", value(model.w_b))
print("Tons of corn bought:", value(model.c_b))
print("Tons of excess sugar beet:", value(model.sb_x))
```

```
ERROR:pyomo.core:Rule failed when generating expression for Objective obj wit
PyomoException: Cannot convert non-constant Pyomo expression (2.5*w - 200  <
This error is usually caused by using a Var, unit, or mutable Param in a
Boolean context such as an "if" statement, or when checking container
membership or equality. For example,
    >>> m.x = Var()
    >>> if m.x >= 1:
    ...     pass
and
    >>> m.y = Var()
    >>> if m.y in [m.x, m.y]:
    ...     pass
would both cause this exception.
ERROR:pyomo.core:Constructing component 'obj' from data=None failed:
    PyomoException: Cannot convert non-constant Pyomo expression (2.5*w - 200
This error is usually caused by using a Var, unit, or mutable Param in a
Boolean context such as an "if" statement, or when checking container
membership or equality. For example,
    >>> m.x = Var()
    >>> if m.x >= 1:
    ...     pass
and
    >>> m.y = Var()
    >>> if m.y in [m.x, m.y]:
    ...     pass
would both cause this exception.
---------------------------------------------------------------------------
PyomoException                            Traceback (most recent call last)
<ipython-input-11-6608f9dd2da9> in <cell line: 42>()
     40         model.P_sbx * model.sb_x
     41     )
---> 42 model.obj = Objective(rule=obj_rule, sense=minimize)
     43
     44 # Define the constraints

5 frames
/usr/local/lib/python3.10/dist-packages/pyomo/core/expr/relational_expr.py in
__bool__(self)
     45         if self.is_constant():
     46             return bool(self())
---> 47         raise PyomoException(
     48             """
     49 Cannot convert non-constant Pyomo expression (%s) to bool.

PyomoException: Cannot convert non-constant Pyomo expression (2.5*w - 200  <
0) to bool.
This error is usually caused by using a Var, unit, or mutable Param in a
Boolean context such as an "if" statement, or when checking container
membership or equality. For example,
    >>> m.x = Var()
    >>> if m.x >= 1:
    ...     pass
and
    >>> m.y = Var()
    >>> if m.y in [m.x, m.y]:
    ...     pass
```

5. Correct The Model Code to Test Mathematical Model (if applicable)

```
%%capture
import sys
import os

if 'google.colab' in sys.modules:
    !pip install idaes-pse --pre
    !idaes get-extensions --to ./bin
    os.environ['PATH'] += ':bin'


from pyomo.environ import *

# Create a Concrete Model
model = ConcreteModel()

# Define the parameters
model.W_min = 200
model.C_min = 240
model.P_w = 170
model.P_c = 150
model.P_wb = 1.4 * model.P_w
model.P_cb = 1.4 * model.P_c
model.P_sb = 36
model.P_sbx = 10
model.Q_sb = 6000
```

```
model.Y_w = 2.5
model.Y_c = 3
model.Y_sb = 20
model.C_w = 150
model.C_c = 230
model.C_sb = 260
model.L = 500

# Define the decision variables
model.w = Var(domain=NonNegativeReals)
model.c = Var(domain=NonNegativeReals)
model.sb = Var(domain=NonNegativeReals)
model.w_b = Var(domain=NonNegativeReals)
model.c_b = Var(domain=NonNegativeReals)
model.sb_x = Var(domain=NonNegativeReals)

# MODIFIED INTRODUCE BINARY VARIABLES TO LINEARIZE MAX/ MIN IN OBJECTIVE
model.Y1 = Var(domain=Binary)
model.Y2 = Var(domain=Binary)

# Define the objective function
def obj_rule(model):
    return (
        model.C_w * model.w + model.C_c * model.c + model.C_sb * model.sb +
        model.P_wb * model.w_b + model.P_cb * model.c_b -
        model.P_w * (model.Y_w * model.w - model.W_min) * model.Y1 - # MODIFIED max(model.Y_w * model.w - model.W_min, 0)
        model.P_c * (model.Y_c * model.c - model.C_min) * model.Y2 - # MODIFIED max(model.Y_c * model.c - model.C_min, 0)
        model.P_sb * (model.Y_sb * (model.sb - model.sb_x)) - # MODIFIED min(model.Y_sb * model.sb, model.Q_sb)
        model.P_sbx * model.sb_x
    )
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define the constraints
def land_constraint_rule(model):
    return model.w + model.c + model.sb <= model.L
model.land_constraint = Constraint(rule=land_constraint_rule)

def wheat_requirement_rule(model):
    return model.Y_w * model.w + model.w_b >= model.W_min
model.wheat_requirement = Constraint(rule=wheat_requirement_rule)

def corn_requirement_rule(model):
    return model.Y_c * model.c + model.c_b >= model.C_min
model.corn_requirement = Constraint(rule=corn_requirement_rule)

def sugar_beet_quota_rule(model):
    return model.Y_sb * model.sb == model.Q_sb + model.sb_x
model.sugar_beet_quota = Constraint(rule=sugar_beet_quota_rule)

# MODIFIED INTRODUCE CONSTRAINTS TO LINEARIZE MAX/ MIN IN OBJECTIVE
def wheat_sold_rule1(model):
  return model.Y_w * model.w - model.W_min <= 1250 * model.Y1
model.wheat_sold_rule1 = Constraint(rule=wheat_sold_rule1)

def wheat_sold_rule2(model):
  return  - model.Y_w * model.w + model.W_min <= 1250 * (1-model.Y1)
model.wheat_sold_rule2 = Constraint(rule=wheat_sold_rule2)

def corn_sold_rule1(model):
  return model.Y_c * model.c - model.C_min <= 1500 * model.Y2
model.corn_sold_rule1 = Constraint(rule=corn_sold_rule1)

def corn_sold_rule2(model):
  return  - model.Y_c * model.c + model.C_min <= 1500 * (1-model.Y2)
model.corn_sold_rule2 = Constraint(rule=corn_sold_rule2)

# Solve the optimization problem
solver = SolverFactory('couenne')
results = solver.solve(model)

# Print the results
print("Objective value:", value(model.obj))
print("Acres of wheat:", value(model.w))
print("Acres of corn:", value(model.c))
print("Acres of sugar beet:", value(model.sb))
print("Tons of wheat bought:", value(model.w_b))
print("Tons of corn bought:", value(model.c_b))
print("Tons of excess sugar beet:", value(model.sb_x))
```