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/MIP/MIP3.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:
 You are tasked with scheduling the power output of 6 electric power thermal units over the timespan of 15 periods. Ther
```

```
!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)
```

```
Here's the mathematical optimization model for the given power scheduling problem:

Parameters:
- N: Number of power units (N = 6)
- T: Number of time periods (T = 15)
- SUC[i]: Startup cost for power unit i
- SDC[i]: Shutdown cost for power unit i
- FC[i]: Fixed cost for power unit i
- VC[i,t]: Variable cost for power unit i at time period t
- PMIN[i]: Minimum power output for power unit i
- PMAX[i]: Maximum power output for power unit i
- RU[i]: Maximum power increment (ramp-up) for power unit i
- RD[i]: Maximum power decrement (ramp-down) for power unit i
- D[t]: Power demand at time period t

Decision Variables:
- u[i,t]: Binary variable indicating if power unit i is on (1) or off (0) at time period t
- p[i,t]: Continuous variable representing the power output of unit i at time period t
- v[i,t]: Binary variable indicating if power unit i is started up (1) or not (0) at time period t
- w[i,t]: Binary variable indicating if power unit i is shut down (1) or not (0) at time period t

Objective Function:
minimize ∑[i=1 to N] ∑[t=1 to T] (SUC[i] * v[i,t] + SDC[i] * w[i,t] + FC[i] * u[i,t] + VC[i,t] * p[i,t])

Constraints:
1. Power output limits:
   PMIN[i] * u[i,t] ≤ p[i,t] ≤ PMAX[i] * u[i,t], ∀i, ∀t

2. Ramp-up and ramp-down constraints:
   p[i,t] - p[i,t-1] ≤ RU[i] * u[i,t-1] + PMAX[i] * v[i,t], ∀i, ∀t ≥ 2
   p[i,t-1] - p[i,t] ≤ RD[i] * u[i,t] + PMAX[i] * w[i,t], ∀i, ∀t ≥ 2

3. Startup and shutdown constraints:
   v[i,t] - w[i,t] = u[i,t] - u[i,t-1], ∀i, ∀t ≥ 2
   v[i,t] + w[i,t] ≤ 1, ∀i, ∀t

4. Demand satisfaction constraint:
   ∑[i=1 to N] p[i,t] ≥ D[t], ∀t

5. Security constraint (10% excess capacity):
   ∑[i=1 to N] PMAX[i] * u[i,t] ≥ 1.1 * D[t], ∀t

6. Binary and non-negativity constraints:
   u[i,t], v[i,t], w[i,t] ∈ {0, 1}, ∀i, ∀t
   p[i,t] ≥ 0, ∀i, ∀t

The objective function minimizes the total cost, which includes startup costs, shutdown costs, fixed costs, and variable

The constraints ensure that the power output limits, ramp-up and ramp-down limits, startup and shutdown logic, demand sa
```

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)
```

```
# Define objective function
def objective_rule(model):
    return sum(SUC[i-1] * model.v[i,t] + SDC[i-1] * model.w[i,t] +
               FC[i-1] * model.u[i,t] + VC[i-1][t-1] * model.p[i,t]
               for i in model.N for t in model.T)
model.objective = pyo.Objective(rule=objective_rule, sense=pyo.minimize)

# Define constraints
def power_output_limits_rule(model, i, t):
    return (PMIN[i-1] * model.u[i,t], model.p[i,t], PMAX[i-1] * model.u[i,t])
model.power_output_limits = pyo.Constraint(model.N, model.T, rule=power_output_limits_rule)

def ramp_up_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.p[i,t] - model.p[i,t-1] <= RU[i-1] * model.u[i,t-1] + PMAX[i-1] * model.v[i,t]
model.ramp_up_constraints = pyo.Constraint(model.N, model.T, rule=ramp_up_rule)

def ramp_down_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.p[i,t-1] - model.p[i,t] <= RD[i-1] * model.u[i,t] + PMAX[i-1] * model.w[i,t]
model.ramp_down_constraints = pyo.Constraint(model.N, model.T, rule=ramp_down_rule)

def startup_shutdown_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.v[i,t] - model.w[i,t] == model.u[i,t] - model.u[i,t-1]
model.startup_shutdown_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_rule)

def startup_shutdown_limit_rule(model, i, t):
    return model.v[i,t] + model.w[i,t] <= 1
model.startup_shutdown_limit_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_limit_rule)

def demand_satisfaction_rule(model, t):
    return sum(model.p[i,t] for i in model.N) >= D[t-1]
model.demand_satisfaction_constraints = pyo.Constraint(model.T, rule=demand_satisfaction_rule)

def security_rule(model, t):
    return sum(PMAX[i-1] * model.u[i,t] for i in model.N) >= 1.1 * D[t-1]
model.security_constraints = pyo.Constraint(model.T, rule=security_rule)

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

# Print the results
print(results)
```

In this code, I used sample data for the parameters such as `SUC`, `SDC`, `FC`, `VC`, `PMIN`, `PMAX`, `RU`, `RD`, and

The code follows the mathematical formulation closely. It defines the sets, decision variables, objective function, an

The objective function is defined using the `objective_rule`, and the constraints are defined using separate rule func

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
!pip install pandas
```

Show hidden output

```
import pyomo.environ as pyo

# Sample data (replace with actual data)
N = 6
T = 15
SUC = [10324, 5678, 7802, 12899, 4596, 9076]
SDC = [2673, 5893, 982, 6783, 2596, 3561]
FC = [2000, 3000, 2500, 4000, 3500, 4500]
VC = [[20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
      [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
      [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32],
      [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
      [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36],
      [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]
      ]
PMIN = [50, 40, 30, 60, 55, 65]
PMAX = [500, 600, 550, 700, 650, 750]
RU = [100, 120, 110, 130, 125, 140]
RD = [90, 110, 100, 120, 115, 130]
D = [1000, 1200, 1300, 1100, 1500, 1400, 1600, 1300, 1700, 1800, 1900, 1600, 2000, 1800, 1700]

# Create a Pyomo model
model = pyo.ConcreteModel()

# Define sets
model.N = pyo.RangeSet(1, N)
model.T = pyo.RangeSet(1, T)

# Define decision variables
model.u = pyo.Var(model.N, model.T, domain=pyo.Binary)
model.p = pyo.Var(model.N, model.T, domain=pyo.NonNegativeReals)
model.v = pyo.Var(model.N, model.T, domain=pyo.Binary)
model.w = pyo.Var(model.N, model.T, domain=pyo.Binary)

# Define objective function
def objective_rule(model):
    return sum(SUC[i-1] * model.v[i,t] + SDC[i-1] * model.w[i,t] +
               FC[i-1] * model.u[i,t] + VC[i-1][t-1] * model.p[i,t]
               for i in model.N for t in model.T)
model.objective = pyo.Objective(rule=objective_rule, sense=pyo.minimize)

# Define constraints
def power_output_limits_rule(model, i, t):
    return (PMIN[i-1] * model.u[i,t], model.p[i,t], PMAX[i-1] * model.u[i,t])
model.power_output_limits = pyo.Constraint(model.N, model.T, rule=power_output_limits_rule)

def ramp_up_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.p[i,t] - model.p[i,t-1] <= RU[i-1] * model.u[i,t-1] + PMAX[i-1] * model.v[i,t]
model.ramp_up_constraints = pyo.Constraint(model.N, model.T, rule=ramp_up_rule)

def ramp_down_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.p[i,t-1] - model.p[i,t] <= RD[i-1] * model.u[i,t] + PMAX[i-1] * model.w[i,t]
model.ramp_down_constraints = pyo.Constraint(model.N, model.T, rule=ramp_down_rule)

def startup_shutdown_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.v[i,t] - model.w[i,t] == model.u[i,t] - model.u[i,t-1]
model.startup_shutdown_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_rule)

def startup_shutdown_limit_rule(model, i, t):
    return model.v[i,t] + model.w[i,t] <= 1
model.startup_shutdown_limit_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_limit_rule)

def demand_satisfaction_rule(model, t):
    return sum(model.p[i,t] for i in model.N) >= D[t-1]
model.demand_satisfaction_constraints = pyo.Constraint(model.T, rule=demand_satisfaction_rule)

def security_rule(model, t):
    return sum(PMAX[i-1] * model.u[i,t] for i in model.N) >= 1.1 * D[t-1]
model.security_constraints = pyo.Constraint(model.T, rule=security_rule)

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

# Print the results
print(results)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-fef2bf6c7e3b> in <cell line: 79>()
     77 # Solve the optimization problem
     78 solver = pyo.SolverFactory('glpk')
---> 79 results = solver.solve(model)
     80
     81 # Print the results

                              11 frames
/usr/local/lib/python3.10/dist-packages/pyomo/core/base/constraint.py in
_get_range_bound(self, range_arg)
    205         bound = self._expr.arg(range_arg)
    206         if not is_fixed(bound):
--> 207             raise ValueError(
    208                 "Constraint '%s' is a Ranged Inequality with a "
    209                 "variable %s bound.  Cannot normalize the "

ValueError: Constraint 'power_output_limits[1,1]' is a Ranged Inequality with
a variable lower bound.  Cannot normalize the constraint or send it to a
```

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

```
import pyomo.environ as pyo

# Sample data (replace with actual data)
N = 6
T = 15
SUC = [10324, 5678, 7802, 12899, 4596, 9076]
SDC = [2673, 5893, 982, 6783, 2596, 3561]
FC = [2000, 3000, 2500, 4000, 3500, 4500]
VC = [[20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
      [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
      [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32],
      [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
      [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36],
      [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]
      ]
PMIN = [50, 40, 30, 60, 55, 65]
PMAX = [500, 600, 550, 700, 650, 750]
RU = [100, 120, 110, 130, 125, 140]
RD = [90, 110, 100, 120, 115, 130]
D = [1000, 1200, 1300, 1100, 1500, 1400, 1600, 1300, 1700, 1800, 1900, 1600, 2000, 1800, 1700]

# Create a Pyomo model
model = pyo.ConcreteModel()

# Define sets
model.N = pyo.RangeSet(1, N)
model.T = pyo.RangeSet(1, T)

# Define decision variables
model.u = pyo.Var(model.N, model.T, domain=pyo.Binary)
model.p = pyo.Var(model.N, model.T, domain=pyo.NonNegativeReals)
model.v = pyo.Var(model.N, model.T, domain=pyo.Binary)
model.w = pyo.Var(model.N, model.T, domain=pyo.Binary)

# Define objective function
def objective_rule(model):
    return sum(SUC[i-1] * model.v[i,t] + SDC[i-1] * model.w[i,t] +
               FC[i-1] * model.u[i,t] + VC[i-1][t-1] * model.p[i,t]
               for i in model.N for t in model.T)
model.objective = pyo.Objective(rule=objective_rule, sense=pyo.minimize)

# Define constraints MODIFIED
def power_output_limits_rule_lower(model, i, t): # MODIFIED THE CONSTRAINT FOR PROPER SYNTAX
    return PMIN[i-1] * model.u[i,t] <= model.p[i,t]
model.power_output_limits_lower = pyo.Constraint(model.N, model.T, rule=power_output_limits_rule_lower)

def power_output_limits_rule_upper(model, i, t): # MODIFIED THE CONSTRAINT FOR PROPER SYNTAX
    return model.p[i,t] <= PMAX[i-1] * model.u[i,t]
model.power_output_limits_upper = pyo.Constraint(model.N, model.T, rule=power_output_limits_rule_upper)

def ramp_up_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
    return model.p[i,t] - model.p[i,t-1] <= RU[i-1] * model.u[i,t-1] + PMAX[i-1] * model.v[i,t]
model.ramp_up_constraints = pyo.Constraint(model.N, model.T, rule=ramp_up_rule)

def ramp_down_rule(model, i, t):
    if t == 1:
        return pyo.Constraint.Skip
```

```
            return pyo.Constraint.Skip
        return model.p[i,t-1] - model.p[i,t] <= RD[i-1] * model.u[i,t] + PMAX[i-1] * model.w[i,t]
    model.ramp_down_constraints = pyo.Constraint(model.N, model.T, rule=ramp_down_rule)

    def startup_shutdown_rule(model, i, t):
        if t == 1:
            return pyo.Constraint.Skip
        return model.v[i,t] - model.w[i,t] == model.u[i,t] - model.u[i,t-1]
    model.startup_shutdown_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_rule)

    def startup_shutdown_limit_rule(model, i, t):
        return model.v[i,t] + model.w[i,t] <= 1
    model.startup_shutdown_limit_constraints = pyo.Constraint(model.N, model.T, rule=startup_shutdown_limit_rule)

    def demand_satisfaction_rule(model, t):
        return sum(model.p[i,t] for i in model.N) >= D[t-1]
    model.demand_satisfaction_constraints = pyo.Constraint(model.T, rule=demand_satisfaction_rule)

    def security_rule(model, t):
        return sum(PMAX[i-1] * model.u[i,t] for i in model.N) >= 1.1 * D[t-1]
    model.security_constraints = pyo.Constraint(model.T, rule=security_rule)

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

    # Print the results
    print(results)
```

```
Problem:
- Name: unknown
  Lower bound: 734011.0
  Upper bound: 734011.0
  Number of objectives: 1
  Number of constraints: 552
  Number of variables: 360
  Number of nonzeros: 1728
  Sense: minimize
Solver:
- Status: ok
  Termination condition: optimal
  Statistics:
    Branch and bound:
      Number of bounded subproblems: 1667
      Number of created subproblems: 1667
  Error rc: 0
  Time: 1.3121862411499023
Solution:
- number of solutions: 0
```