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
# Variables Prompt
prompt11_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt11_MathematicalModel.txt'

# Objective Prompt
prompt12_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt12_MathematicalModel.txt'

# Constraint Prompt
prompt13_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt13_MathematicalModel.txt'

# Code Prompt
prompt2_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt2_PyomoCode.txt'
problem_desc_path = '/content/drive/MyDrive/Thesis/ProblemDescriptions/IP/IP3.txt'

prompt11_file = open(prompt11_path, "r")
prompt12_file = open(prompt12_path, "r")
prompt13_file = open(prompt13_path, "r")
prompt2_file = open(prompt2_path, "r")
problem_desc_file = open(problem_desc_path, "r")

prompt11 = prompt11_file.read()
print("Prompt 1.1 (Variables):\n", prompt11)

prompt12 = prompt12_file.read()
print("Prompt 1.2 (Objctive):\n", prompt12)

prompt13 = prompt13_file.read()
print("Prompt 1.3 (Constraints):\n", prompt13)

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

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

```
Prompt 1.1 (Variables):
 Please formulate only the variables for this mathematical optimization problem.
Prompt 1.2 (Objctive):
 Please formulate only the objective function for this mathematical optimization problem.
Prompt 1.3 (Constraints):
 Please formulate only the constraints for this mathematical optimization problem.
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 the person in charge of packing in a large company. Your job is to skillfully pack items of various weights in
```

```
!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 1.1: Create Variables for Mathematical Model

```
message11 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt11,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": problem_desc}
    ]
)

response_p11 = message11.content[0].text


# Print response
print(response_p11)
```

```
Let's define the following variables for the mathematical optimization problem:

- Let I be the set of items to be packed, indexed by i.
- Let J be the set of available boxes, indexed by j.
- Let w_i be the weight of item i.
- Let C_j be the capacity of box j.
- Let M be the upper bound on the number of boxes that can be used.
- Let x_ij be a binary variable that equals 1 if item i is packed into box j, and 0 otherwise.
- Let y_j be a binary variable that equals 1 if box j is used, and 0 otherwise.

These are the key variables and parameters needed to formulate the optimization model for the packing problem, with the
```

1. Prompt 1.2: Create Objective for Mathematical Model

```
message12 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt12,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": (problem_desc + response_p11)}
    ]
)

response_p12 = message12.content[0].text


# Print response
print(response_p12)
```

```
Great! Now that we have defined the variables and parameters, let's formulate the objective function for this mathematic

Objective Function:
Minimize ∑(j ∈ J) y_j

This objective function aims to minimize the total number of boxes used. By summing up the binary variables y_j over all

The complete mathematical optimization model would also include the following constraints:

1. Capacity constraints: For each box j, the total weight of items packed into it should not exceed its capacity C_j.
2. Assignment constraints: Each item i must be packed into exactly one box.
3. Linking constraints: If any item is packed into a box j (i.e., if x_ij = 1 for any i), then that box must be counted
4. Upper bound constraint: The total number of boxes used should not exceed the upper bound M.

These constraints, along with the binary restrictions on the variables x_ij and y_j, complete the formulation of the mat
```

1. Prompt 1.3: Create Constraints for Mathematical Model

```
message13 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt13,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": (problem_desc + response_p11 + response_p12)}
    ]
)

response_p13 = message13.content[0].text


# Print response
print(response_p13)
```

```
Great! Let's formulate the constraints for this mathematical optimization problem.

1. Capacity constraints:
   For each box j ∈ J, the total weight of items packed into it should not exceed its capacity C_j.

   ∑(i ∈ I) w_i * x_ij ≤ C_j * y_j, ∀j ∈ J

2. Assignment constraints:
   Each item i ∈ I must be packed into exactly one box.

   ∑(j ∈ J) x_ij = 1, ∀i ∈ I

3. Linking constraints:
   If any item is packed into a box j (i.e., if x_ij = 1 for any i), then that box must be counted as used (y_j = 1).

   x_ij ≤ y_j, ∀i ∈ I, ∀j ∈ J

4. Upper bound constraint:
   The total number of boxes used should not exceed the upper bound M.

   ∑(j ∈ J) y_j ≤ M

5. Binary restrictions:
   The variables x_ij and y_j are binary, i.e., they can only take values 0 or 1.

   x_ij ∈ {0, 1}, ∀i ∈ I, ∀j ∈ J
   y_j ∈ {0, 1}, ∀j ∈ J

These constraints, along with the objective function to minimize the total number of boxes used, complete the formulatio
```

2. Prompt 2: Write Code based on Mathematical Model

In case more than the component of each step was generated, use "last instance"/ "newest version" of the components.

Example: Prompt 1.1 (Variables): Model generates only variables Prompt 1.2 (Objectvie): Model generates objective and constraints Prompt 1.3 (Constraints): Model generates only constraints

Use Variables from step 1.1, use objective from 1.2 and use constraints from 1.3 (since constraints version 1.3 is more recent than constraints version 1.2).

This set up allows for potential self-correction of the model throughout the three prompts.

Correctness of each component is still generated in the respective step, so self-correction is independent of this.

```
message2 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt2,
    max_tokens=4069,
    messages=[
        {"role": "user", "content": (response_p11 + response_p12 + response_p13)}
    ]
)

response_p2 = message2.content[0].text


# Print response
print(response_p2)
```

```
# Define the model
model = ConcreteModel()

# Define the sets
```

```
model.M = Param(initialize=3)  # Upper bound on the number of boxes (sample data)

# Define the variables
model.x = Var(model.I, model.J, domain=Binary)  # Binary variable for item-box assignment
model.y = Var(model.J, domain=Binary)  # Binary variable for box usage

# Define the objective function
model.obj = Objective(expr=sum(model.y[j] for j in model.J), sense=minimize)

# Define the constraints
model.capacity = ConstraintList()
for j in model.J:
    model.capacity.add(sum(model.w[i] * model.x[i, j] for i in model.I) <= model.C[j] * model.y[j])

model.assignment = ConstraintList()
for i in model.I:
    model.assignment.add(sum(model.x[i, j] for j in model.J) == 1)

model.linking = ConstraintList()
for i in model.I:
    for j in model.J:
        model.linking.add(model.x[i, j] <= model.y[j])

model.upper_bound = Constraint(expr=sum(model.y[j] for j in model.J) <= model.M)

# Solve the model
solver = SolverFactory('glpk')  # You can use any solver available in Pyomo
results = solver.solve(model)

# Print the results
print("Number of boxes used:", value(model.obj))
for j in model.J:
    if value(model.y[j]) == 1:
        print("Box", j, "is used.")
        for i in model.I:
            if value(model.x[i, j]) == 1:
                print("  Item", i, "is packed into box", j)
```

In this code, we first define the model using `ConcreteModel()`. Then, we define the sets `I` and `J` for items and bo

Next, we define the binary variables `x` and `y` for item-box assignment and box usage, respectively. We then define t

After that, we define the constraints using `ConstraintList()`. The capacity constraints ensure that the total weight

We solve the model using the GLPK solver (you can use any solver available in Pyomo) and print the results, including

Note that this code uses sample data for the sets, parameters, and upper bound. You can replace them with your actual

4. Input Problem Data and Test Model Code

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

Show hidden output

```
from pyomo.environ import *

# Define the model
model = ConcreteModel()

# Define the sets
model.I = Set(initialize=['item1', 'item2', 'item3', 'item4', 'item5', 'item6', 'item7', 'item8', 'item9', 'item10', 'item11'
model.J = Set(initialize=['box1', 'box2', 'box3', 'box4', 'box5', 'box6', 'box7', 'box8', 'box9', 'box10', 'box11', 'box12',

weights = [2, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 7, 7, 8, 8]
capacity = 9

# Define the parameters
model.w = Param(model.I, initialize={model.I[i+1]: weights[i] for i in range(len(model.I))})  # Weight of items (sample data)
model.C = Param(model.J, initialize={model.J[j+1]: capacity for j in range(len(model.J))})  # Capacity of boxes (sample data)
model.M = Param(initialize=13)  # Upper bound on the number of boxes (sample data)

# Define the variables
model.x = Var(model.I, model.J, domain=Binary)  # Binary variable for item-box assignment
model.y = Var(model.J, domain=Binary)  # Binary variable for box usage

# Define the objective function
model.obj = Objective(expr=sum(model.y[j] for j in model.J), sense=minimize)
```

```
# Define the constraints
model.capacity = ConstraintList()
for j in model.J:
    model.capacity.add(sum(model.w[i] * model.x[i, j] for i in model.I) <= model.C[j] * model.y[j])

model.assignment = ConstraintList()
for i in model.I:
    model.assignment.add(sum(model.x[i, j] for j in model.J) == 1)

model.linking = ConstraintList()
for i in model.I:
    for j in model.J:
        model.linking.add(model.x[i, j] <= model.y[j])

model.upper_bound = Constraint(expr=sum(model.y[j] for j in model.J) <= model.M)

# Solve the model
solver = SolverFactory('glpk')  # You can use any solver available in Pyomo
results = solver.solve(model)

# Print the results
print("Number of boxes used:", value(model.obj))
for j in model.J:
    if value(model.y[j]) == 1:
        print("Box", j, "is used.")
        for i in model.I:
            if value(model.x[i, j]) == 1:
                print("  Item", i, "is packed into box", j)
```

```
WARNING:pyomo.core.base.set:DEPRECATED: Using __getitem__ to return a set value from its (ordered)
position is deprecated.  Please use at()  (deprecated in 6.1, will be
removed in (or after) 7.0)
(called from <ipython-input-16-0335ef172d72>:14)
WARNING:pyomo.core.base.set:DEPRECATED: Using __getitem__ to return a set value from its (ordered)
position is deprecated.  Please use at()  (deprecated in 6.1, will be
removed in (or after) 7.0)
(called from <ipython-input-16-0335ef172d72>:15)
Number of boxes used: 13.0
Box box1 is used.
  Item item23 is packed into box box1
Box box2 is used.
  Item item24 is packed into box box2
Box box3 is used.
  Item item2 is packed into box box3
  Item item21 is packed into box box3
Box box4 is used.
  Item item10 is packed into box box4
  Item item17 is packed into box box4
Box box5 is used.
  Item item7 is packed into box box5
  Item item16 is packed into box box5
Box box6 is used.
  Item item9 is packed into box box6
  Item item13 is packed into box box6
Box box7 is used.
  Item item11 is packed into box box7
  Item item15 is packed into box box7
Box box8 is used.
  Item item3 is packed into box box8
  Item item19 is packed into box box8
Box box9 is used.
  Item item5 is packed into box box9
  Item item14 is packed into box box9
Box box10 is used.
  Item item4 is packed into box box10
  Item item22 is packed into box box10
Box box11 is used.
  Item item8 is packed into box box11
  Item item12 is packed into box box11
Box box12 is used.
  Item item1 is packed into box box12
  Item item20 is packed into box box12
Box box13 is used.
  Item item6 is packed into box box13
  Item item18 is packed into box box13
```

```
bins = {'box1': [], 'box2': [], 'box3': [], 'box4': [], 'box5': [], 'box6': [], 'box7': [], 'box8': [], 'box9': [], 'box10':
for (i,j) in model.x:
  if value(model.x[i,j])> .5:
    bins[j].append(model.w[i])

print("Bin Division:", bins)
```

```
Bin Division: {'box1': [8], 'box2': [8], 'box3': [2, 7], 'box4': [4, 5], 'box5': [4, 5], 'box6': [4, 5], 'box7': [4, 5],
```

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