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 variables first:

1. Sets:
   - I: set of items to be packed, indexed by i
   - J: set of available boxes, indexed by j

2. Parameters:
   - w[i]: weight of item i
   - C[j]: capacity of box j
   - M: upper bound on the number of boxes that can be used

3. Decision variables:
   - x[i,j]: binary variable; 1 if item i is packed into box j, 0 otherwise
   - y[j]: binary variable; 1 if box j is used, 0 otherwise

The mathematical optimization model can be formulated as follows:

Objective function:
- Minimize the total number of boxes used
  min ∑[j ∈ J] y[j]

Constraints:

1. Each item must be packed into exactly one box:
   ∑[j ∈ J] x[i,j] = 1, ∀i ∈ I

2. The total weight of items packed into each box must not exceed its capacity:
   ∑[i ∈ I] w[i] * x[i,j] ≤ C[j] * y[j], ∀j ∈ J

3. The number of boxes used must not exceed the upper bound:
   ∑[j ∈ J] y[j] ≤ M

4. Binary variable constraints:
   x[i,j] ∈ {0,1}, ∀i ∈ I, ∀j ∈ J
   y[j] ∈ {0,1}, ∀j ∈ J

The objective function minimizes the total number of boxes used. Constraint 1 ensures that each item is packed into exac
```

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)
```

```
The objective function for the given mathematical optimization model can be formulated as follows:

Objective function:
- Minimize the total number of boxes used
  min ∑[j ∈ J] y[j]

where:
- y[j] is a binary decision variable; 1 if box j is used, 0 otherwise
- J is the set of available boxes, indexed by j

The objective function aims to minimize the sum of the binary variables y[j] over all boxes j in the set J. By minimizin
```

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)
```

```
Constraints:

1. Each item must be packed into exactly one box:
   ∑[j ∈ J] x[i,j] = 1, ∀i ∈ I

2. The total weight of items packed into each box must not exceed its capacity:
   ∑[i ∈ I] w[i] * x[i,j] ≤ C[j] * y[j], ∀j ∈ J

3. The number of boxes used must not exceed the upper bound:
   ∑[j ∈ J] y[j] ≤ M

4. Binary variable constraints:
   x[i,j] ∈ {0,1}, ∀i ∈ I, ∀j ∈ J
   y[j] ∈ {0,1}, ∀j ∈ J

Constraint 1 ensures that each item is packed into exactly one box. The sum of x[i,j] over all boxes j for each item i m

Constraint 2 guarantees that the total weight of items packed into each box does not exceed its capacity. The sum of the

Constraint 3 limits the number of boxes used to the upper bound M. The sum of the binary variables y[j] over all boxes j

Constraint 4 defines the binary nature of the decision variables x[i,j] and y[j]. These variables can only take values o
```

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.

```
response_p11_adjusted = """
Let's define the variables first:

1. Sets:
   - I: set of items to be packed, indexed by i
   - J: set of available boxes, indexed by j

2. Parameters:
   - w[i]: weight of item i
   - C[j]: capacity of box j
   - M: upper bound on the number of boxes that can be used

3. Decision variables:
   - x[i,j]: binary variable; 1 if item i is packed into box j, 0 otherwise
   - y[j]: binary variable; 1 if box j is used, 0 otherwise
"""


message2 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt2,
    max_tokens=4069,
    messages=[
        {"role": "user", "content": (response_p11_adjusted + response_p12 + response_p13)}
    ]
)

response_p2 = message2.content[0].text


# Print response
print(response_p2)
```

```
Here's a Python code using Pyomo to solve the optimization problem:

```python
from pyomo.environ import *

# Define the model
model = AbstractModel()

# Define sets
model.I = Set()  # Set of items
model.J = Set()  # Set of boxes

# Define parameters
model.w = Param(model.I, within=NonNegativeReals)  # Weight of each item
model.C = Param(model.J, within=NonNegativeReals)  # Capacity of each box
model.M = Param(within=NonNegativeIntegers)  # Upper bound on the number of boxes

# Define decision variables
model.x = Var(model.I, model.J, within=Binary)  # Binary variable: 1 if item i is packed into box j, 0 otherwise
model.y = Var(model.J, within=Binary)  # Binary variable: 1 if box j is used, 0 otherwise

# Define objective function
def obj_rule(model):
    return sum(model.y[j] for j in model.J)
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define constraints
def item_packing_rule(model, i):
    return sum(model.x[i,j] for j in model.J) == 1
model.item_packing = Constraint(model.I, rule=item_packing_rule)

def capacity_rule(model, j):
    return sum(model.w[i] * model.x[i,j] for i in model.I) <= model.C[j] * model.y[j]
model.capacity = Constraint(model.J, rule=capacity_rule)

def box_limit_rule(model):
    return sum(model.y[j] for j in model.J) <= model.M
model.box_limit = Constraint(rule=box_limit_rule)

# Load sample data (replace with your own data)
sample_data = {
    'I': {1, 2, 3, 4, 5},
    'J': {1, 2, 3},
    'w': {1: 5, 2: 3, 3: 2, 4: 4, 5: 1},
    'C': {1: 10, 2: 8, 3: 6},
    'M': 2
}

# Create an instance of the model and load data
instance = model.create_instance(data=sample_data)

# Solve the model
solver = SolverFactory('glpk')  # You can use any solver compatible with Pyomo
results = solver.solve(instance)
```
```

```
    # Print the results
    print("Objective value:", value(instance.obj))
```

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 = AbstractModel()

# Define sets
model.I = Set()  # Set of items
model.J = Set()  # Set of boxes

# Define parameters
model.w = Param(model.I, within=NonNegativeReals)  # Weight of each item
model.C = Param(model.J, within=NonNegativeReals)  # Capacity of each box
model.M = Param(within=NonNegativeIntegers)  # Upper bound on the number of boxes

# Define decision variables
model.x = Var(model.I, model.J, within=Binary)  # Binary variable: 1 if item i is packed into box j, 0 otherwise
model.y = Var(model.J, within=Binary)  # Binary variable: 1 if box j is used, 0 otherwise

# Define objective function
def obj_rule(model):
    return sum(model.y[j] for j in model.J)
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define constraints
def item_packing_rule(model, i):
    return sum(model.x[i,j] for j in model.J) == 1
model.item_packing = Constraint(model.I, rule=item_packing_rule)

def capacity_rule(model, j):
    return sum(model.w[i] * model.x[i,j] for i in model.I) <= model.C[j] * model.y[j]
model.capacity = Constraint(model.J, rule=capacity_rule)

def box_limit_rule(model):
    return sum(model.y[j] for j in model.J) <= model.M
model.box_limit = Constraint(rule=box_limit_rule)

# Load sample data (replace with your own data)
sample_data = {
    'I': {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24},
    'J': {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13},
    'w': {1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 3, 7: 4, 8: 4, 9: 4, 10: 4, 11: 4, 12: 4, 13: 5, 14: 5, 15: 5, 16: 5, 17: 5, 18:
    'C': {1: 9, 2: 9, 3: 9, 4: 9, 5: 9, 6: 9, 7: 9, 8: 9, 9: 9, 10: 9, 11: 9, 12: 9, 13: 9},
    'M': 13
}

# Create an instance of the model and load data
instance = model.create_instance(data=sample_data)

# Solve the model
solver = SolverFactory('glpk')  # You can use any solver compatible with Pyomo
results = solver.solve(instance)

# Print the results
print("Objective value:", value(instance.obj))
print("Items packed:")
for i in instance.I:
    for j in instance.J:
        if value(instance.x[i,j]) == 1:
            print(f"Item {i} is packed into box {j}")
print("Boxes used:")
for j in instance.J:
    if value(instance.y[j]) == 1:
        print(f"Box {j} is used")
```

```
ERROR:pyomo.common.numeric_types:evaluating object as numeric value: M
    (object: <class 'pyomo.core.base.param.ScalarParam'>)
Error retrieving immutable Param value (M):
        The Param value is undefined and no default value is specified.
ERROR:pyomo.core:Rule failed when generating expression for Constraint box_li
ValueError: Error retrieving immutable Param value (M):
        The Param value is undefined and no default value is specified.
ERROR:pyomo.core:Constructing component 'box_limit' from data=None failed:
    ValueError: Error retrieving immutable Param value (M):
        The Param value is undefined and no default value is specified.
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-f54de1a11aeb> in <cell line: 47>()
     45 
     46 # Create an instance of the model and load data
---> 47 instance = model.create_instance(data=sample_data)
     48 
     49 # Solve the model

13 frames

pyomo/core/expr/numeric_expr.pyx in
pyomo.core.expr.numeric_expr.NumericValue.__ge__()

/usr/local/lib/python3.10/dist-packages/pyomo/core/base/param.py in
_getitem_when_not_present(self, index)
    618             else:
    619                 idx_str = '%s' % (self.name,)
--> 620             raise ValueError(
    621                 "Error retrieving immutable Param value (%s):\n\tThe
Param "
    622                 "value is undefined and no default value is
specified." % (idx_str,)
```

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 *

# Define the model
model = AbstractModel()

# Define sets
model.I = Set()  # Set of items
model.J = Set()  # Set of boxes

# Define parameters
model.w = Param(model.I, within=NonNegativeReals)  # Weight of each item
model.C = Param(model.J, within=NonNegativeReals)  # Capacity of each box
model.M = Param(within=NonNegativeIntegers)  # Upper bound on the number of boxes

# Define decision variables
model.x = Var(model.I, model.J, within=Binary)  # Binary variable: 1 if item i is packed into box j, 0 otherwise
model.y = Var(model.J, within=Binary)  # Binary variable: 1 if box j is used, 0 otherwise

# Define objective function
def obj_rule(model):
    return sum(model.y[j] for j in model.J)
model.obj = Objective(rule=obj_rule, sense=minimize)

# Define constraints
def item_packing_rule(model, i):
    return sum(model.x[i,j] for j in model.J) == 1
model.item_packing = Constraint(model.I, rule=item_packing_rule)

def capacity_rule(model, j):
    return sum(model.w[i] * model.x[i,j] for i in model.I) <= model.C[j] * model.y[j]
model.capacity = Constraint(model.J, rule=capacity_rule)

def box_limit_rule(model):
    return sum(model.y[j] for j in model.J) <= model.M
model.box_limit = Constraint(rule=box_limit_rule)

# Load sample data (replace with your own data)
sample_data = {None: dict(
    I= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24},
    J= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13},
    w= {1: 2, 2: 2, 3: 2, 4: 2, 5: 3, 6: 3, 7: 4, 8: 4, 9: 4, 10: 4, 11: 4, 12: 4, 13: 5, 14: 5, 15: 5, 16: 5, 17: 5, 18: 5,
    C= {1: 9, 2: 9, 3: 9, 4: 9, 5: 9, 6: 9, 7: 9, 8: 9, 9: 9, 10: 9, 11: 9, 12: 9, 13: 9},
    M= {None: 13}
)}

# Create an instance of the model and load data
instance = model.create_instance(data=sample_data)

# Solve the model
solver = SolverFactory('glpk')  # You can use any solver compatible with Pyomo
results = solver.solve(instance)

# Print the results
print("Objective value:", value(instance.obj))
print("Items packed:")
for i in instance.I:
    for j in instance.J:
        if value(instance.x[i,j]) == 1:
            print(f"Item {i} is packed into box {j}")
print("Boxes used:")
for j in instance.J:
    if value(instance.y[j]) == 1:
        print(f"Box {j} is used")
```

```
WARNING:pyomo.core:Initializing ordered Set I with a fundamentally unordered data source (type: set).  This WILL potenti
WARNING:pyomo.core:Initializing ordered Set J with a fundamentally unordered data source (type: set).  This WILL potenti
Objective value: 13.0
Items packed:
```

```
bins = {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: [], 11: [], 12: [], 13: []}
for i in range (1, 25):
  for j in range(1, 14):
    if value(instance.x[i,j])> .5:
      bins[j].append(instance.w[i])

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

```
Bin Division: {1: [8], 2: [2, 7], 3: [7], 4: [4, 5], 5: [6], 6: [4, 5], 7: [4, 5], 8: [4, 5], 9: [3, 6], 10: [4, 5], 11:
```

```
Item 12 is packed into box 11
Item 13 is packed into box 7
Item 14 is packed into box 6
Item 15 is packed into box 11
Item 16 is packed into box 8
Item 17 is packed into box 10
```