scipy.optimize.linprog is the go-to tool in Python for solving **Linear Programming** problems.

Linear programming solves problems of the following form:

$$\begin{aligned} \min_x \ & c^T x \\ \text{such that } & A_{ub} x \leq b_{ub}, \\ & A_{eq} x = b_{eq}, \\ & l \leq x \leq u, \end{aligned}$$

where x is a vector of decision variables; c, b_{ub}, b_{eq}, l, and u are vectors; and A_{ub} and A_{eq} are matrices.

Here is the most important rule to remember: **linprog always minimizes.**

If you actually want to *maximize* something, you just multiply your objective function by -1.

****
****
****

A Simple Example: The Budget Diet

Imagine you are trying to meet a daily protein goal using two foods: **Steak** and **Eggs**.

- **Eggs** cost $0.20 each and give you 6g of protein.
- **Steak** costs $2.00 per serving and gives you 30g of protein.
- **Goal:** You need at least 60g of protein today, and you want to spend the *least* amount of money possible.

Let x_0 be the number of eggs, and x_1 be the servings of steak.

1. Translating the Problem to Math

- **What we want to minimize (Cost):**

$$0.20x_0 + 2.00x_1$$

- **Our Constraint (Protein):**

$$6x_0 + 30x_1 \geq 60$$

Because linprog expects all constraints to use "less than or equal to" ($\leq$), we multiply our protein constraint by -1:

$$-6x_0 - 30x_1 \leq -60$$

2. The Python Code

Here is how you set this up and solve it using SciPy:

```
import numpy as np
from scipy.optimize import linprog

# 1. Coefficients of the objective function (the costs)
# Minimize: 0.20 * x0 + 2.00 * x1
c = [0.20, 2.00]

# 2. Left-hand side coefficients of the inequality constraints
# Notice we made them negative to flip the >= to <=
A_ub = [[-6, -30]]

# 3. Right-hand side values of the inequality constraints
b_ub = [-60]

# 4. Bounds for each variable (you can't eat negative eggs or steak!)
# None means no upper limit
x0_bounds = (0, None)
x1_bounds = (0, None)
bounds = [x0_bounds, x1_bounds]

# 5. Run the optimization
res = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

# Print the results
print("Success:", res.success)
print("Lowest Cost:", res.fun)
print("Optimal quantities (Eggs, Steak):", res.x)
```

The Core Arguments Checklist

When your problems get bigger, you just add more rows and columns to these arguments:

- **c**: A 1D array of the coefficients you want to minimize.
- **A_ub** & **b_ub**: The **U**pper **B**ound inequality matrix and vector ($A_{ub} \cdot x \leq b_{ub}$).
- **A_eq** & **b_eq**: *(Optional)* The **Eq**uality matrix and vector if you have strict equations ($A_{eq} \cdot x = b_{eq}$).
- **bounds**: A sequence of (min, max) pairs for each variable to keep them within realistic limits.
- **method='highs'**: Always include this. The HiGHS solver is the modern, fastest, and most reliable solver bundled with SciPy.

****
****
****

Another Simple Example: Crafting Custom Furniture

Let's say you run a woodshop making two products: **Tables** (x_0) and **Chairs** (x_1). You want to maximize your weekly profit.

- **Profit:** Each table brings in **$200**, and each chair brings in **$80**.
- **Resource 1 (Wood):** You only have **300 board-feet** of wood per week.
    - A table takes 30 board-feet; a chair takes 10 board-feet.
- **Resource 2 (Labor):** You have **110 hours** of shop labor available per week.
    - A table takes 5 hours to build; a chair takes 4 hours.

Formulating the Math

- **Maximize Profit:** $200x_0 + 80x_1 \implies$ **Minimize:** $-200x_0 - 80x_1$
- **Wood Constraint:** $30x_0 + 10x_1 \leq 300$
- **Labor Constraint:** $5x_0 + 4x_1 \leq 110$

Because these constraints are naturally "less than or equal to" (≤), we don't need to flip any signs on them like we did in the first diet example.

The Python Implementation

Here is how you stack multiple constraints into **A_ub** (as rows in a matrix) and **b_ub** (as a vector):

Python

```
import numpy as np
from scipy.optimize import linprog

# 1. Objective function coefficients (NEGATED for maximization)
# Maximize: 200*x0 + 80*x1  =>  Minimize: -200*x0 - 80*x1
c = [-200, -80]

# 2. Left-hand side coefficients for inequality constraints (A_ub)
# Row 0: Wood usage (30 per table, 10 per chair)
# Row 1: Labor usage (5 per table, 4 per chair)
A_ub = [
  [30, 10],  # Wood coefficients
  [ 5,  4]   # Labor coefficients
]

# 3. Right-hand side capacities (b_ub)
b_ub = [300, 110]  # Max wood, Max labor

# 4. Bounds (Can't make negative furniture)
bounds = [(0, None), (0, None)]

# 5. Solve using the modern HiGHS solver
res = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

# 6. Extract and correct the results
# Remember to negate the optimal value back to a positive profit!
max_profit = -res.fun

print("Optimization Successful:", res.success)
print(f"Maximum Weekly Profit: ${max_profit:,.2f}")
print(f"Optimal Number of Tables to make: {res.x[0]:.1f}")
print(f"Optimal Number of Chairs to make: {res.x[1]:.1f}")
```

Breaking Down the Results

If you run this code, you will get the following output:

- **Maximum Weekly Profit:** $2,200.00
- **Optimal Production:** 2.0 Tables and 24.0 Chairs.

How to Double-Check the Math Line Inline

- **Wood check:** $(30 \times 2) + (10 \times 24) = 60 + 240 = 300$ board-feet (Used 100% of available wood).
- **Labor check:** $(5 \times 2) + (4 \times 24) = 10 + 96 = 106$ hours (Used 106 out of 110 hours, leaving 4 hours of leftover capacity).

In a business scenario, a constraint like **"110 hours of shop labor available per week"** represents the total capacity of your workforce during that specific time frame. It's the hard ceiling on how much actual work can physically get done.

Instead of being a random number, this limit is usually built from a combination of standard shifts, part-time help, and operational reality.

How 110 Hours is Built (The Real-World Math)

If you are explaining this to a team, a manager, or writing a business plan, you can break that 110-hour pool down into a realistic staffing schedule. Here are two very common ways a small workshop hits exactly 110 hours:

Scenario A: The Balanced Small Team

- **1 Full-Time Maker:** Works a standard 40-hour week.
- **1 Full-Time Apprentice:** Works a standard 40-hour week.
- **1 Part-Time Assistant:** Works 30 hours a week (e.g., 6 hours a day, 5 days a week).
- **Total Capacity:** $40 + 40 + 30 = \mathbf{110}$ hours

Scenario B: The Solo Owner + Extra Hands

- **The Shop Owner:** Works a dedicated 50 hours a week.
- **2 Part-Time Workers:** Work 30 hours a week each.
- **Total Capacity:** $50 + 30 + 30 = \mathbf{110\text{ hours}}$

Why Use a Single "Labor Pool" Instead of Tracking Workers Individually?

When you plug numbers into a linear programming model like linprog, you generally group your resources into macro buckets. Here is why we treat labor as one big 110-hour pool:

- **Cross-Training:** It assumes that the people working those hours are capable of handling the tasks required to build the products (both tables and chairs).
- **Fungible Time:** If the apprentice spends 4 hours sanding a table, that frees up 4 hours for the owner to assemble a chair. As long as the skills overlap, the time is interchangeable.
- **Simplified Scheduling:** Before deciding exactly *who* works on *what* on Tuesday morning, management first needs to know what the entire shop is physically capable of producing as a collective unit.

The "Operational Efficiency" Caveat: In the real world, a manager knows that no one actually works 100% of the time they are clocked in. Clean-up, machine maintenance, bathroom breaks, and setting up tools eat up time. Therefore, that 110 hours usually represents **"direct labor capacity"**—the time safely allocated just for production after accounting for daily shop overhead.

A Quick Real-World Warning: Linear programming assumes everything is perfectly divisible. If your constraints yielded a result like 2.4 tables, linprog can't tell you to round up or down to make sense for whole objects. If you ever *strictly* need whole numbers (integers), you would add integrality=[1, 1] to your

linprog call to trigger Mixed-Integer Linear Programming (MILP).

****
****
****

Another Simple Example: Blending Problem

In linear programming, equality constraints (A_eq and b_eq) are used when something **must be exactly equal** to a specific value.

The classic real-world example of this is a **Blending Problem**. Imagine you are running a chemical plant or a fuel refinery, and you need to mix different ingredients together to create an exact batch size of a final product while keeping costs as low as possible.

The Scenario: Mixing Eco-Fuel
You need to mix a batch of exactly **100 gallons** of eco-friendly fuel. You can blend three components to make it:

- **Component A (x_0):** Costs $1.50 per gallon.
- **Component B (x_1):** Costs $2.00 per gallon.
- **Component C (x_2):** Costs $3.00 per gallon.

The Constraints

1. **Strict Equality:** The total volume must equal **exactly 100 gallons**.
$$x_0 + x_1 + x_2 = 100$$
2. **Inequality (Eco-Rating):** To meet environmental standards, the mix must contain **at least 30 gallons of Component C**.
$$x_2 \geq 30 \implies -x_2 \leq -30$$

Formulating the Math

- **Minimize Cost:** $1.50x_0 + 2.00x_1 + 3.00x_2$
- **Inequality Matrix (A_ub, b_ub):** $-1x_2 \leq -30$
- **Equality Matrix (A_eq, b_eq):** $1x_0 + 1x_1 + 1x_2 = 100$

The Python Code
Notice how we separate our inequality arguments (_ub) from our strict equality arguments (_eq):
Python

```
import numpy as np
from scipy.optimize import linprog

# 1. Costs per gallon for A, B, and C
c = [1.50, 2.00, 3.00]

# 2. Inequality Constraint (A_ub and b_ub): Component C >= 30
# Formatted as: 0*x0 + 0*x1 - 1*x2 <= -30
A_ub = [[0, 0, -1]]
b_ub = [-30]

# 3. Strict Equality Constraint (A_eq and b_eq): Total must equal 100
# Formatted as: 1*x0 + 1*x1 + 1*x2 = 100
A_eq = [[1, 1, 1]]
b_eq = [100]

# 4. Bounds (No negative gallons)
bounds = [(0, None), (0, None), (0, None)]
```

```
# 5. Solve the problem
res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')

print("Optimization Successful:", res.success)
print(f"Minimum Cost for Batch: ${res.fun:.2f}")
print(f"Gallons of Component A: {res.x[0]:.1f}")
print(f"Gallons of Component B: {res.x[1]:.1f}")
print(f"Gallons of Component C: {res.x[2]:.1f}")
```

Analyzing the Output

If you execute this code, linprog calculates the perfect cost-saving balance:

- **Total Minimum Cost:** $195.00
- **Component A:** 70.0 gallons
- **Component B:** 0.0 gallons
- **Component C:** 30.0 gallons

Why did it pick this?

The mathematical logic is flawless. It was forced by b_eq to make exactly 100 gallons, and forced by b_ub to use at least 30 gallons of the expensive Component C ($3.00). To fill the remaining 70 gallons of the batch as cheaply as possible, it completely bypassed Component B ($2.00) and bought only the cheapest option left, Component A ($1.50).

$$70 + 0 + 30 = \mathbf{100} \text{ gallons strictly}$$

Another Python Code

```
import numpy as np
from scipy.optimize import linprog

# 1. Costs per gallon for A, B, and C
c = np.array([1.50, 2.00, 3.00])

# 2. Inequality Constraint (A_ub and b_ub): Component C >= 30
# Formatted as: 0*x0 + 0*x1 - 1*x2 <= -30
A_ub = np.array([[0, 0, -1]])
b_ub = np.array([-30])

# 3. Strict Equality Constraint (A_eq and b_eq): Total must equal 100
# Formatted as: 1*x0 + 1*x1 + 1*x2 = 100
#A_eq = [1, 1, 1]
A_eq = np.array([[1, 1, 1]])
b_eq = np.array([100])

# 4. Bounds (No negative gallons)
bounds = np.array([(0, None), (0, None), (0, None)])

# 5. Solve the problem
res = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')

print("Optimization Successful:", res.success)
print(f"Minimum Cost for Batch: ${res.fun:.2f}")
print(f"Gallons of Component A: {res.x[0]:.1f}")
print(f"Gallons of Component B: {res.x[1]:.1f}")
print(f"Gallons of Component C: {res.x[2]:.1f}")

print("*******************************")
print("*******************************")
print("*******************************")
```

```
A_eq1 = np.array([1, 1, 1])
A_eq2 = np.array([[1, 1, 1]])

print("******************************")
print(len(A_eq1))
print(A_eq1.shape)
print(A_eq1.ndim)
print(A_eq1.size)
print(A_eq1.dtype)
print(type(A_eq1))

print("******************************")
print(len(A_eq2))
print(A_eq2.shape)
print(A_eq2.ndim)
print(A_eq2.size)
print(A_eq2.dtype)
print(type(A_eq2))
```