

Step 01 Set up the model

Try first: Write the two resource rows yourself before copying. Which row contains the coefficient 2?

RUN THIS CODE IN YOUR PYTHON SESSION

```
import numpy as np
from scipy.optimize import linprog
np.set_printoptions(precision=4, suppress=True)
A = np.array([[1., 1.], [2., 1.]])
c = np.array([3., 2.]) # Contributions, not minimization coefficients.
b = np.array([4., 5.])
def solve(c=(3., 2.), b=(4., 5.), matrix=A, bounds=(0, None)):
    result = linprog(-np.asarray(c, dtype=float),
                     A_ub=matrix, b_ub=np.asarray(b, dtype=float),
                     bounds=bounds, method="highs")
    if not result.success:
        raise RuntimeError(result.message)
    return result
print("Resource matrix:", A, sep="\n")
print("Contributions:", c)
print("Capacities:", b)
```

EXPECTED OUTPUT

```
Resource matrix:
[[1. 1.]
 [2. 1.]]
Contributions: [3. 2.]
Capacities: [4. 5.]
```

Step 02 Solve and read the baseline

Try first: Predict production, contribution and unused capacity. Run, then compare all three.

RUN THIS CODE IN YOUR PYTHON SESSION

```
base = solve()
print("success:", base.success)
print("production:", base.x)
print("minimized objective:", base.fun)
print("max contribution:", -base.fun)
print("resource slack:", base.ineqlin.residual)
print("row marginals:", base.ineqlin.marginals)
assert np.allclose(base.x, [1, 3])
assert np.isclose(-base.fun, 9)
```

EXPECTED OUTPUT

```
success: True
production: [1. 3.]
minimized objective: -9.0
max contribution: 9.0
resource slack: [0. 0.]
row marginals: [-1. -1.]
```

Step 03 Verify the plan without trusting the solver

Try first: Check the infeasible candidate (2,2). Explain why contribution 10 cannot beat the optimum.

RUN THIS CODE IN YOUR PYTHON SESSION

```
x = base.x
use = A @ x
slack = b - use
print("resource use:", use)
print("computed slack:", slack)
print("computed contribution:", c @ x)
assert np.all(x >= -1e-9)
assert np.all(slack >= -1e-9)
bad = np.array([2., 2.])
print("candidate (2,2) resource use:", A @ bad)
print("candidate feasible:", bool(np.all(A @ bad <= b)))
vertices = np.array([[0,0], [2.5,0], [1,3], [0,4]])
print("vertex contributions:", vertices @ c)
```

EXPECTED OUTPUT

```
resource use: [4. 5.]
computed slack: [0. 0.]
computed contribution: 9.0
candidate (2,2) resource use: [4. 6.]
candidate feasible: False
vertex contributions: [0.  7.5 9.  8. ]
```

Step 04 Solve the dual explicitly

Try first: Why do we negate both A transpose and c in the dual inequalities?

RUN THIS CODE IN YOUR PYTHON SESSION

```
dual = linprog(b, A_ub=-A.T, b_ub=-c,  
              bounds=(0, None), method="highs")  
assert dual.success, dual.message  
prices = dual.x  
print("dual prices:", prices)  
print("dual slack:", A.T @ prices - c)  
print("dual objective:", dual.fun)  
print("duality gap:", dual.fun + base.fun)  
assert np.all(A.T @ prices >= c - 1e-9)  
assert np.isclose(dual.fun, -base.fun)
```

EXPECTED OUTPUT

```
dual prices: [1. 1.]  
dual slack: [0. 0.]  
dual objective: 9.0  
duality gap: 0.0
```

Step 05 Check complementarity and signs

Try first: Explain why a zero product does not prove feasibility by itself.

RUN THIS CODE IN YOUR PYTHON SESSION

```
resource_slack = b - A @ base.x
dual_slack = A.T @ prices - c
products = np.r_[prices * resource_slack, base.x * dual_slack]
print("complementarity products:", products)
print("profit shadow prices:", -base.ineqlin.marginals)
print("max reduced costs:", c - A.T @ prices)
assert np.allclose(products, 0)
assert np.allclose(prices, -base.ineqlin.marginals)
assert np.allclose(c - A.T @ prices, -base.lower.marginals)
```

EXPECTED OUTPUT

```
complementarity products: [0. 0. 0. 0.]
profit shadow prices: [1. 1.]
max reduced costs: [0. 0.]
```

Step 06 Perturb one capacity at a time

Try first: Predict both production vectors before running. Why must each trial start from the original b?

RUN THIS CODE IN YOUR PYTHON SESSION

```
h = 0.1
for i in range(2):
    changed = b.copy() # Reset for each independent trial.
    changed[i] += h
    trial = solve(b=changed)
    min_slope = (trial.fun - base.fun) / h
    profit_slope = (-trial.fun + base.fun) / h
    print("resource", i+1, "production", trial.x,
          "min slope", round(min_slope, 6),
          "profit slope", round(profit_slope, 6))
    assert np.isclose(profit_slope, prices[i])
```

EXPECTED OUTPUT

```
resource 1 production [0.9 3.2] min slope -1.0
profit slope 1.0
resource 2 production [1.1 2.9] min slope -1.0
profit slope 1.0
```

Step 07 Discover the range for resource one

Try first: At $b_1=6$, calculate the old prediction before reading the actual result. Explain the discrepancy.

RUN THIS CODE IN YOUR PYTHON SESSION

```
for b1 in [2, 2.5, 4, 4.5, 5, 6]:
    r = solve(b=(b1, 5))
    old_plan = np.array([5-b1, 2*b1-5])
    old_prediction = 9 + (b1-4)
    print("b1", b1, "plan", r.x, "value", round(-r.fun, 4),
          "old plan feasible", bool(np.all(old_plan >= 0)),
          "old predicted value", old_prediction)
    expected = min(3*b1, b1+5, 10)
    assert np.isclose(-r.fun, expected)
print("At 2.5 and 5, left and right slopes differ.")
```

EXPECTED OUTPUT

```
b1 2 plan [2. 0.] value 6.0 old plan feasible
False old predicted value 7
b1 2.5 plan [ 2.5 -0. ] value 7.5 old plan
feasible True old predicted value 7.5
b1 4 plan [1. 3.] value 9.0 old plan feasible
True old predicted value 9
b1 4.5 plan [0.5 4. ] value 9.5 old plan feasible
True old predicted value 9.5
b1 5 plan [0. 5.] value 10.0 old plan feasible
True old predicted value 10
b1 6 plan [0. 5.] value 10.0 old plan feasible
False old predicted value 11
At 2.5 and 5, left and right slopes differ.
```

Step 08 Resource two and simultaneous changes

Try first: Explain why the separate allowable ranges do not justify capacities (5,4).

RUN THIS CODE IN YOUR PYTHON SESSION

```
for b2 in [3, 4, 5, 6, 8, 9]:
    r = solve(b=(4, b2))
    expected = min(2*b2, b2+4, 12)
    assert np.isclose(-r.fun, expected)
    print("b2", b2, "plan", r.x, "value", round(-r.fun, 4))
b1, b2 = 5., 4.
joint_plan = np.array([b2-b1, 2*b1-b2])
joint = solve(b=(b1,b2))
print("old joint plan:", joint_plan)
print("joint range satisfied:", b1 <= b2 <= 2*b1)
print("actual joint solution:", joint.x, "value", -joint.fun)
assert np.allclose(joint.x, [0,4])
```

EXPECTED OUTPUT

```
b2 3 plan [0. 3.] value 6.0
b2 4 plan [0. 4.] value 8.0
b2 5 plan [1. 3.] value 9.0
b2 6 plan [2. 2.] value 10.0
b2 8 plan [ 4. -0.] value 12.0
b2 9 plan [4. 0.] value 12.0
old joint plan: [-1.  6.]
joint range satisfied: False
actual joint solution: [0. 4.] value 8.0
```

Step 09 Rotate the objective

Try first: At $c_1=2$ and $c_1=4$, verify an entire edge instead of asserting one solver vector.

RUN THIS CODE IN YOUR PYTHON SESSION

```
for c1 in [1, 2, 3, 3.5, 4, 5]:
    r = solve(c=(c1,2))
    expected = 8 if c1 < 2 else c1+6 if c1 <= 4 else 2.5*c1
    assert np.isclose(-r.fun, expected)
    print("c1", c1, "plan", r.x, "value", round(-r.fun,4))
for point in [[0,4], [0.5,3.5], [1,3]]:
    assert np.all(A @ point <= b + 1e-9)
    assert np.isclose(np.dot([2,2], point), 8)
for point in [[1,3], [1.75,1.5], [2.5,0]]:
    assert np.all(A @ point <= b + 1e-9)
    assert np.isclose(np.dot([4,2], point), 10)
print("Both endpoint optimal edges verified.")
```

EXPECTED OUTPUT

```
c1 1 plan [0. 4.] value 8.0
c1 2 plan [0. 4.] value 8.0
c1 3 plan [1. 3.] value 9.0
c1 3.5 plan [1. 3.] value 9.5
c1 4 plan [1. 3.] value 10.0
c1 5 plan [2.5 0. ] value 12.5
Both endpoint optimal edges verified.
```

Step 10 Introduce a new product and reoptimize

Try first: Compute its old reduced cost. How far can $x=1-t$, $y=3$ continue?

RUN THIS CODE IN YOUR PYTHON SESSION

```
A3 = np.column_stack([A, [1.,2.]])
for ct in [2,3,4]:
    c3 = np.array([3.,2.,ct])
    r = solve(c=c3, matrix=A3)
    print("ct", ct, "old reduced cost", ct-3,
          "plan", r.x, "value", round(-r.fun,4))
    assert np.isclose(-r.fun, 10 if ct==4 else 9)
new_prices = np.array([0.,2.])
new_c = np.array([3.,2.,4.])
assert np.all(A3.T @ new_prices >= new_c - 1e-9)
for point in [[0,3,1], [0,2,1.5], [0,0,2.5]]:
    assert np.all(A3 @ point <= b + 1e-9)
    assert np.isclose(new_c @ point, b @ new_prices)
print("Alternative new-product optima certified at value 10.")
```

EXPECTED OUTPUT

```
ct 2 old reduced cost -1 plan [1. 3. 0.] value
9.0
ct 3 old reduced cost 0 plan [1. 3. 0.] value 9.0
ct 4 old reduced cost 1 plan [0.  0.  2.5] value
10.0
Alternative new-product optima certified at value
10.
```

Step 11 Interpret a lower-bound marginal

Try first: Predict the loss from forcing $x \geq 0.1$ under objective $x + 3y$.

RUN THIS CODE IN YOUR PYTHON SESSION

```
changed_objective = solve(c=(1,3))
forced = solve(c=(1,3), bounds=[(0.1,None),(0,None)])
print("baseline plan:", changed_objective.x)
print("row marginals:", changed_objective.ineqlin.marginals)
print("lower-bound marginals:", changed_objective.lower.marginals)
print("forced plan:", forced.x)
print("profit loss:", round(-changed_objective.fun + forced.fun, 6))
assert np.allclose(forced.x, [0.1,3.9])
assert np.isclose(-changed_objective.fun + forced.fun, 0.2)
```

EXPECTED OUTPUT

```
baseline plan: [0. 4.]
row marginals: [-3. -0.]
lower-bound marginals: [2. 0.]
forced plan: [0.1 3.9]
profit loss: 0.2
```

Step 12 Choose capacity by net benefit

Try first: Choose among A, B, C and no purchase before revealing the outputs.

RUN THIS CODE IN YOUR PYTHON SESSION

```
baseline_dollars = -solve().fun * 100
offers = [("None",0,0), ("A",0.5,30), ("B",1,70), ("C",2,90)]
results = []
for name,q,cost in offers:
    gross = -solve(b=(4+q,5)).fun*100 - baseline_dollars
    net = gross - cost
    results.append((net,name))
    print(name, "gross gain", round(gross,2),
          "cost", cost, "net benefit", round(net,2))
best = max(results)
print("best package:", best[1], "net benefit:", round(best[0],2))
assert best[1] == "B"
qs = np.linspace(0,2,201)
nets = np.array([-solve(b=(4+q,5)).fun*100-baseline_dollars-60*q for q in qs])
q_best = qs[np.argmax(nets)]
print("divisible capacity: q =", q_best, "net =", round(max(nets),2))
assert np.isclose(q_best,1)
```

EXPECTED OUTPUT

```
None gross gain 0.0 cost 0 net benefit 0.0
A gross gain 50.0 cost 30 net benefit 20.0
B gross gain 100.0 cost 70 net benefit 30.0
C gross gain 100.0 cost 90 net benefit 10.0
best package: B net benefit: 30.0
divisible capacity: q = 1.0 net = 40.0
```

Step 13 Diagnose failures and attempt the exercises

Try first: Run both failure cases. Then complete exercises.py before opening answers.py.

RUN THIS CODE IN YOUR PYTHON SESSION

```
infeasible = linprog([1.], A_ub=[[1.],[-1.]], b_ub=[1.,-2.],  
                    bounds=(None,None), method="highs")  
unbounded = linprog([-1.], bounds=(0,None), method="highs")  
print("infeasible success/status:", infeasible.success, infeasible.status)  
print("unbounded success/status:", unbounded.success, unbounded.status)  
assert not infeasible.success and infeasible.status == 2  
assert not unbounded.success and unbounded.status == 3  
print("Do not interpret failed solves as optimal production plans.")  
print("Next: complete the independent exercises, then check answers.py.")
```

EXPECTED OUTPUT

```
infeasible success/status: False 2  
unbounded success/status: False 3  
Do not interpret failed solves as optimal  
production plans.  
Next: complete the independent exercises, then  
check answers.py.
```