Basic guidelines for using Gurobi w/ Python

Installing Gurobi in Colab

Each time you open a new Colab session, they need to install Gurobi. The installation cell looks like this:

```
!pip install gurobipy
```

After running this cell, they import the library:

```
import gurobipy as gp
from gurobipy import GRB
```

Gurobi offers a free academic license that removes size limitations on problems. Students can register at gurobi.com/academia/academic-program-and-licenses using their Northeastern email. However, for the small problems typical in coursework, the default license included with gurobipy is usually sufficient.

By the end of each session, run the following code to free the license from that workflow:

```
model.dispose()
gp.disposeDefaultEnv()
```

The Basic Structure of a Gurobi Model

Every linear programming model in Gurobi follows the same pattern, which maps directly onto the components you teach: decision variables, objective function, and constraints.

```
python
# 1. Create a new model
model = gp.Model("model_name")  # name your model

# 2. Add decision variables
x = model.addVar(name="x", lb=0)  # lb = lower bound
y = model.addVar(name="y", lb=0)

# 3. Set the objective function
model.setObjective(3*x + 5*y, GRB.MAXIMIZE)  # or GRB.MINIMIZE

# 4. Add constraints
model.addConstr(x + 2*y <= 10, "constraint_1")
model.addConstr(3*x + y <= 15, "constraint_2")

# 5. Solve the model
model.optimize()

# 6. Print results
if model.status == GRB.OPTIMAL:
   print(f"Optimal x = {x.X}")
   print(f"Optimal y = {y.X}")
   print(f"Optimal objective value = {model.ObjVal}")
```

This structure reinforces what you learn when formulating by hand—the code is essentially a direct translation of the mathematical model.

Common Variable Types and Bounds

You will encounter different variable specifications as problems grow more complex:
python
```
# Continuous variable (default), non-negative
x = model.addVar(lb=0, ub=GRB.INFINITY, vtype=GRB.CONTINUOUS, name="x")

# Variable with both lower and upper bounds
y = model.addVar(lb=0, ub=100, name="y")

# For integer programming (later in their studies)
z = model.addVar(vtype=GRB.INTEGER, name="z")
```

Adding Multiple Variables at Once

For larger problems, creating variables one at a time becomes tedious. Gurobi provides **addVars** for this:

```
# Create 5 variables: x[0], x[1], x[2], x[3], x[4]
x = model.addVars(5, lb=0, name="x")  # Structure : name-of-your-model.addVars

# Create variables indexed by a list
products = ['A', 'B', 'C']
production = model.addVars(products, lb=0, name="produce")
```

Writing Constraints with Summations

When constraints involve sums over many variables use Python's **quicksum**:

```
# import function
from gurobipy import quicksum

# Sum of all x variables must be <= 100
model.addConstr(quicksum(x[i] for i in range(5)) <= 100, "total_limit")

# Weighted sum
costs = [2, 3, 1, 4, 2]
model.addConstr(quicksum(costs[i] * x[i] for i in range(5)) <= 50, "budget")
```

Reading and Interpreting Output

After model.optimize(), you can extract various pieces of information:

```
# Check if solution is optimal
if model.status == GRB.OPTIMAL:
  # Variable values
  for v in model.getVars():
    print(f"{v.VarName} = {v.X}")

  # Objective value
  print(f"Objective = {model.ObjVal}")


  # For sensitivity analysis (connects to Module 6)
  for c in model.getConstrs():
    print(f"{c.ConstrName}: Shadow Price = {c.Pi}") #Get shadow price
```

The shadow prices (.Pi) connect directly to the sensitivity analysis you learn in Module Dual Theory, where you emphasize that "knowing how solutions change when inputs change is often more valuable than the optimal solution itself" (Final Learning Objectives).

Troubleshooting Common Issues

Model is infeasible: The constraints contradict each other. You should check their constraint formulations against their mathematical model.

Model is unbounded: The objective can increase (or decrease) infinitely. Usually means a constraint is missing or a variable lacks appropriate bounds.

Syntax errors with constraints: The most common mistake is using = instead of == for equality constraints, or forgetting that Python uses <= and >= rather than ≤ and ≥.

Variables show as 0 when they shouldn't: Check that the variable was included in the objective or constraints—a variable that doesn't appear anywhere will default to its lower bound.

A Note for Students New to Programming

If you're encountering Python for the first time, focus on understanding the *structure* rather than memorizing syntax. The code mirrors the mathematical formulation you're learning to write by hand:

- model.addVar() → defining decision variables
- model.setObjective() → writing the objective function
- model.addConstr() → adding constraints

When you get stuck, describe what you're trying to do mathematically, then translate piece by piece. And as noted in your course materials, you're encouraged to use AI tools for "debugging code" and "getting discussion points regarding mathematical concepts" (Final Syllabus, Statement on AI Use).

Data Structures

Lists: Ordered Collections

Lists store sequences of items and are accessed by position (starting from 0, not 1).

```
python
# Costs for three products
costs = [4, 6, 3]

# Access individual elements
print(costs[0])  # First element: 4
print(costs[2])  # Third element: 3


# Loop through all elements
for i in range(len(costs)):
   print(f"Product {i} costs {costs[i]}")
```

Lists work well when data has a natural numeric order, like time periods or sequential resources.

Dictionaries: Named Access

Dictionaries store key-value pairs, allowing access by meaningful names rather than positions. This is often more intuitive for optimization problems.

```
python
# Production costs by product name
costs = {'chairs': 45, 'tables': 80, 'desks': 120}

# Access by name
print(costs['chairs'])  # 45

# Loop through dictionary
for product, cost in costs.items():
   print(f"{product} costs ${cost}")
```

Dictionaries make code self-documenting. Compare costs[0] (what's product 0?) with costs['chairs'] (immediately clear).

Combining Structures for Complex Data

Real problems have multiple dimensions—products, time periods, locations, resources. Nested structures handle this naturally.

```
python
# Production capacity by plant and product
capacity = {
  'Boston': {'chairs': 100, 'tables': 50, 'desks': 30},
  'Chicago': {'chairs': 150, 'tables': 75, 'desks': 45}
}

# Access specific capacity
print(capacity['Boston']['chairs'])  # 100

# Loop through all combinations
for plant in capacity:
  for product in capacity[plant]:
    print(f"{plant} can make {capacity[plant][product]} {product}")
```

Using Data Structures with Gurobi

```
import gurobipy as gp
from gurobipy import GRB, quicksum

# Define data
products = ['chairs', 'tables', 'desks']
profit = {'chairs': 25, 'tables': 40, 'desks': 55}
labor_hours = {'chairs': 2, 'tables': 4, 'desks': 3}
wood_required = {'chairs': 3, 'tables': 6, 'desks': 4}
available_labor = 100
available_wood = 150

# Create model
model = gp.Model("production")

# Decision variables indexed by product
produce = model.addVars(products, lb=0, name="produce")

# Objective: maximize total profit
model.setObjective(
  quicksum(profit[p] * produce[p] for p in products),
  GRB.MAXIMIZE )

# Constraints
model.addConstr(
  quicksum(labor_hours[p] * produce[p] for p in products) <= available_labor,
  "labor_limit")

model.addConstr(
  quicksum(wood_required[p] * produce[p] for p in products) <= available_wood,
  "wood_limit")

# Solve
model.optimize()

# Display results
if model.status == GRB.OPTIMAL:
  print(f"\nOptimal profit: ${model.ObjVal:.2f}")
  for p in products:
    if produce[p].X > 0.001:  # Only show non-zero production
      print(f"Produce {produce[p].X:.1f} {p}")
```

Notice how the code reads almost like the mathematical formulation: "maximize the sum of profit times production for each product, subject to labor and wood constraints."

Two-Dimensional Indexing: Transportation Problems

For network problems, you need variables indexed by pairs, like shipping quantities from origins to destinations.

```
python
# Transportation problem data
origins = ['Factory1', 'Factory2']
destinations = ['Store1', 'Store2', 'Store3']
supply = {'Factory1': 100, 'Factory2': 150}
demand = {'Store1': 80, 'Store2': 90, 'Store3': 70}

# Shipping cost from each origin to each destination
cost = {
  ('Factory1', 'Store1'): 8,  ('Factory1', 'Store2'): 6,  ('Factory1', 'Store3'): 10,
  ('Factory2', 'Store1'): 9,  ('Factory2', 'Store2'): 5,  ('Factory2', 'Store3'): 7}

# Create model
model = gp.Model("transportation")

# Variables: ship[origin, destination] = amount shipped
ship = model.addVars(origins, destinations, lb=0, name="ship")

# Objective: minimize total shipping cost
model.setObjective(
  quicksum(cost[o, d] * ship[o, d] for o in origins for d in destinations),
  GRB.MINIMIZE )

# Supply constraints: can't ship more than available
for o in origins:
  model.addConstr(
    quicksum(ship[o, d] for d in destinations) <= supply[o],
    f"supply_{o}"    )

# Demand constraints: must meet each destination's demand
for d in destinations:
  model.addConstr(
    quicksum(ship[o, d] for o in origins) >= demand[d],
    f"demand_{d}"  )

model.optimize()
```

The tuple keys (origin, destination) match the subscript notation you used in mathematical formulations: x_{ij} becomes ship[i, j].

Reading Data from Files

As problems grow larger, typing data directly into code becomes impractical. You should know how to read from external sources.

From CSV files:

```
python
import pandas as pd

# Upload file in Colab: click folder icon, then upload
# Or mount Google Drive

# Read CSV into a DataFrame
data = pd.read_csv('production_data.csv')

# Convert to dictionary for Gurobi
products = data['product'].tolist()
profit = dict(zip(data['product'], data['profit']))
```

From a simple text structure in Colab:

```
# For small examples, define data as a multi-line string
raw_data = """
product,profit,labor,wood
chairs,25,2,3
tables,40,4,6
desks,55,3,4
"""

# Parse with pandas
from io import StringIO
data = pd.read_csv(StringIO(raw_data))
```

Common Pitfalls and How to Avoid Them

Mixing up indices: If products are ['A', 'B', 'C'], don't accidentally use range(3) somewhere. Stay consistent.

```
python
# Inconsistent (error-prone)
products = ['A', 'B', 'C']
for i in range(3):
  print(costs[products[i]])  # Confusing indirection

# Consistent (clearer)
for p in products:
  print(costs[p])
```

KeyError in dictionaries: Accessing a nonexistent key crash the program. Use .get() for safety:

```
python
# This crashes if 'sofas' isn't in the dictionary
price = costs['sofas']

# This returns None (or a default) instead of crashing
price = costs.get('sofas', 0)
```

Forgetting that range() excludes the endpoint: range(5) gives 0, 1, 2, 3, 4—not 0 through 5.

Quick Reference Table

Structure	Best Used For	Example Access
List	Ordered sequences, numeric indices	costs[0]
Dictionary	Named access, readable code	costs['chairs']
Tuple as key	Multi-dimensional indices	cost['Boston', 'Store1']
Set	Collections without duplicates, membership testing	'chairs' in products
DataFrame (pandas)	Tabular data from files	data['profit'].sum()

Connecting to Course Concepts

As you work through the course, you'll see how data structures mirror mathematical notation:

- A list $[c_1, c_2, c_3]$ represents a vector of coefficients
- A dictionary with tuple keys represents a matrix: a[i,j] maps to A_{ij}
- Nested loops with quicksum implement summation notation: $\sum_i \sum_j$

This connection between mathematical formulation and code implementation is central to the course goal of selecting "appropriate solution methods based on problem structure" (Final Learning Objectives). Clean data organization makes that structure visible in your code.